武汉大学 2022 —2023 学年 第一学期
《高性能地理计算》答题纸(论文)

学号：2020302131249　　姓名：马文卓

院系：遥感信息工程学院　　专业：空间信息与数字技术

总分	一	二	三	四	五					

基于 CUDA、cuDNN 加速的 BasicVSR 视频超分设计

马文卓，武汉大学遥感信息工程学院，湖北，武汉 430079

BasicVSR designbased on CUDA and cuDNN acceleration

Wenzhuo Ma
School of Remote Sensing and Information Engineering,Wuhan University
Hubei,Wuhan 430079

摘要：本文对经典的视频超分模型 BasicVSR 进行复现。通过对不同并行策略、不同并行模型的分析讨论，最终选用 CUDA、cuDNN 加速 BasicVSR 网络。实验结果表明，其加速比大约在 3.5 左右。本文还从结果、加速比、内存、网络流等方面对实验进行了全面深入的分析，解释了一些异常现象以及 cuDNN 进一步加速效果不太理想的原因。进一步的，针对加速比太低的事实，本文进行深入地分析，提出了硬件、方法、代码三个层面的优化途径。

关键词：BasicVSR、视频超分、CUDA、cuDNN、并行加速

Abstract： In this paper, the classic video super-resolution model BasicVSR is reproduced. Through the analysis and discussion of different parallel strategies and different parallel models, CUDA and cuDNN are finally selected to accelerate the BasicVSR network. Experimental results show that the acceleration ratio is about 3.5. This paper also makes a comprehensive and in-depth analysis of the experiment from the aspects of the results, acceleration ratio, memory, network flow, etc., and explains some abnormal phenomena and the reasons why cuDNN's further acceleration effect is not ideal. Further, in view of the fact that the acceleration ratio is too low, this paper makes an in-depth analysis, and puts forward three aspects of optimization, namely hardware, method and code.

Keywords：BasicVSR, Video Super Resolution, CUDA, cuDNN, Acceleration in Parallel

目录

1. 引言

1.1 前言

计算机视觉任务一直是人们研究的重点问题，因为计算机视觉任务一旦取得突破性的进展，就标志着计算机能更进一步的取代我们的眼睛，帮助我们感受普通视觉感受不到的信息。而视频恢复任务则又是计算机视觉当中的一大难点问题。现如今，文本图像视频为三大主要信息形式，相对于文本、图像而言，视频恢复由于其庞大的工作量而显得更加困难。

所幸，深度学习的快速发展成熟让视频恢复任务（其实是整个计算机视觉任务）看到了曙光。凭借着深度学习自主学习特征的特性，可以大大减少视觉任务当中的人力成本，基本由计算机取代。于是乎，以深度学习为代表的方法从众多传统方法中脱颖而出，成为了 SOTA（state of the art）。但深度学习也有缺点：深度学习网络大多包含着成千上万的参数等待着被学习，并且不断的迭代更新，这需要很庞大的计算量和计算耗时。特别针对视频而言，计算消耗更加庞大。

为了解决上述问题，除开在算法上优化流程，减少不必要的冗余计算之外，我们更应该考虑的是更加高效的加速体系。于是并行计算应运而生，其宗旨在于在并行机上讲一个计算问题分解成多个子任务分配给不同的处理器，各个处理器之间相互协同并行地执行子任务，以达到加速求解或者求解大规模应用问题的目的。一系列的并行加速模型（如：MPI、OpenMP、MapReduce 等）都被应用到海量数据的计算加速当中，一定程度上促进了大数据时代的到来。

1.2 实验目的

考虑到现实生活中对于高分辨率的视频的大量需求，以及现在视频超分计算速度慢无法满足批量视频超分的现实难题。本文从视频超分的领域出发，选择了 20 年 CVPR 的 SOTA 方法 BasicVSR 视频超分网络，使用 CUDA、cuDNN 对齐进行并行加速。并且通过大量的实验和结果性能分析，对 CUDA、cuDNN 并行化加速的内在算法、具体实施以及和其他并行化模型的对比进行深入的分析讨论。

1.3 组织结构

本文分为五个章节：引言、相关工作、方法设计、实验与结果分析、结论。引言主要是对本文的研究背景、研究目的、文章结构进行概述；相关工作是对本文涉及到的邻域的基本介绍；方法设计则主要对 BasicVSR 的网络设计进行详细介绍，同时介绍了本文基于 CUDA、cuDNN 加速的 BasicVSR 的设计思路；实验与结果分析主要是从这四个方面展开：①不同并行策略在本问题中的适用性②不同并行编程模型在本问题中的适用性③从加速比等角度分析结果和性能④进一步优化途径的阐述；结论部分主要是对实验的总结、对课程的结语、对老师的感谢。（https://github.com/gorgeousmwz/BasicVSR-based-on-CUDA-and-cuDNN.git）

2. 相关工作

2.1 视频恢复

视频恢复（Video Restoration）任务是指将低质量（Low Quility）的视频恢复成高质量（High Quility）的视频。其主要包含以下子任务：

- 视频超分（Video Super-Resolution）：由低分辨率的视频恢复高分辨率
- 视频去模糊（Video Deblurring）：对模糊视频清晰化
- 视频去噪（Video Denoising）：对噪声干扰视频去噪
- 视频去马赛克（Video Demosaicking）：恢复视频帧缺失的颜色信息

与图像恢复任务一样，视频恢复任务也是由传统的插值算法、重建算法等发展而来，由于深度学习方法的兴起，基于学习的视频恢复方法已经占据主流地位。与图像恢复不同，视频恢复除了需要利用每一帧图像的空间信息之外，还需要利用前后高相关性但未对齐的视频帧进行重建。也正是由于这样的不同，在带给视频恢复任务更大前景的同时（因为可利用的信息变多了），也带来了更大的计算量（要计算时空信息）和参数量。

现如今，大多数 SOTA（state of the art，最好的）的视频恢复算法都以 CNN 为基本单元。由于卷积神经网络的基本结构为卷积操作，卷积操作是一种重复性的计算模式，由此使用 GPU 加速网络模型应运而生。

2.2 视频超分

视频超分（Video Super Resolution，VSR），其目的是从低分辨率的视频中恢复高分辨率的帧。在 VSR 领域，从最为传统的插值方法到重建方法，随着深度学习的发展，现如今性能较好的方法大多是基于学习的方法。

视频超分起源于图像超分（Image Super Resolution）。图像超分所需要考虑的重点是如何通过邻域的像素点的信息来计算出带求像素点的信息，也就是利用空间信息。而 VSR 除开需要利用空间信息外，还要尽可能利用更多的时间信息，以达到更好的恢复效果。具体来说，VSR 需要考虑如何利用本帧视频帧邻域信息的同时，还要研究如何尽可能多的利用相邻具有相关性的帧当中有用区域的信息。

现如今，利用相邻帧信息的方法主要是分为两种类型：flow-based（基于光流的）和 flow-free（不基于光流的）。光流是指两帧视频帧之间，相同像素点的运动张量，一般是一个二维张量。Flow-based 的方法主要是通过计算相邻帧之间的光流来进行对齐，以利用相邻帧的信息；Flow-free 的方法主要是不通过计算光流，而是通过一些隐式的（如可形变卷积）方法来对齐相邻帧。

现有的 VSR 方法[1,2,3,4,5,6,7]主要可分为 sliding-window 和 recurrent。Recurrent 方法中的早期方法[8,9,10]预测低分辨率(LR)帧之间的光流，并执行空间 warping 以对齐。之后的方法求助于更复杂的隐式对齐方法。例如，TDAN[11]采用可变形卷积(DCNs)[12,13]在特征级对齐不同的帧。EDVR[14]进一步以多尺度的方式使用 DCNs，以获得更精确的对准。DUF[15]利用动态上采样滤波器隐式处理运动。有些方法采用循环框架。RSDN[16]提出了一个循环的细节结构块和一个隐藏状态自适

应模块，以增强对外观变化和错误积累的鲁棒性。RRN[17]采用具有标识跳过连接的层间残差映射，确保信息流流畅，并长时间保存纹理信息。上述的研究导致了许多新的和复杂的组件来解决 VSR 中的传播和对齐问题。BasicVSR[18]重新研究了一些组件，发现双向传播与基于简单光流的特征对齐相结合足以胜过许多最先进的方法。

2.3 CPU 和 GPU

Graphics Processing Units（GPU，图形处理器），利用处理图形任务的图形处理器来计算原本由中央处理器（CPU）处理的通用计算任务。

由于现代图形处理器强大的并行处理能力和可编程流水线，令流处理器可以处理非图形数据。特别在面对单指令流多数据流(SIMD），且数据处理的运算量远大于数据调度和传输的需要时，通用图形处理器在性能上大大超越了传统的中央处理器应用程序。

CPU 负责逻辑性强的事物处理和串行计算，GPU 则专注于执行高度线程化的并行处理任务（大规模计算任务）。GPU 并不是一个独立运行的计算平台，而需要与 CPU 协同工作，可以看成是 CPU 的协处理器，因此当我们在说 GPU 并行计算时，其实是指的基于 CPU+GPU 的异构计算架构。在异构计算架构中，GPU 与 CPU 通过 PCIe 总线连接在一起来协同工作，CPU 所在位置称为为主机端（host），而 GPU 所在位置称为设备端（device）。

如下图所示，CPU 由大量用于缓存的芯片和少量算术逻辑单元（ALU）组成；而 GPU 由少量用于缓存的芯片和大量用于计算的 ALU 组成。由于硬件结构的差异导致 GPU 相对于 CPU 更适用于高并行性、大规模数据密集型、可预测的计算模式。

得益于 GPU 的硬件结构，其非常适合对于 SIMD（Single Instruction Mutiple Data）模型进行**数据并行计算**，即相同的代码在不同的线程上同时计算。

图 1:CPU 和 GPU 结构示意图

2.4 CUDA 和 cuDNN

2.4.1 CUDA 介绍

Compute Unfied Device Architecture（CUDA）[19]是由 NVIDIA 公司于 2006 年发布的一种新的操作 GPU 的硬件软件架构，是建立在 NVIDIA 的 GPU 上的一个

通用并行计算平台和编程模型，它提供了 GPU 编程的简易接口，基于 CUDA 编程可以构建基于 GPU 计算的应用程序，利用 GPU 的并行计算引擎来更加高效地解决比较复杂的计算难题。它将 GPU 视作一个数据并行计算设备，而且无需把这些计算映射到图形 API。操作系统的多任务机制可以同时管理 CUDA 访问 GPU 和图形程序的运行库，其计算特性支持利用 CUDA 直观地编写 GPU 核心程序。

（1）CUDA 的编程模型

CUDA 的架构中引入了主机端（host）和设备（device）的概念。CUDA 程序中既包含 host 程序，又包含 device 程序。同时，host 与 device 之间可以进行通信，这样它们之间可以进行数据拷贝。

- **主机(Host)**：将 CPU 及系统的内存（内存条）称为主机。
- **设备(Device)**：将 GPU 及 GPU 本身的显示内存称为设备。
- **动态随机存取存储器(DRAM)**：Dynamic Random Access Memory，最为常见的系统内存。DRAM 只能将数据保持很短的时间。为了保持数据，DRAM 使用电容存储，所以必须隔一段时间刷新（refresh）一次，如果存储单元没有被刷新，存储的信息就会丢失。（关机就会丢失数据）

典型的 CUDA 程序的执行流程如下：

1>分配 host 内存，并进行数据初始化；

2>分配 device 内存，并从 host 将数据拷贝到 device 上；

3>调用 CUDA 的核函数在 device 上完成指定的运算；

4>将 device 上的运算结果拷贝到 host 上；

5>释放 device 和 host 上分配的内存；

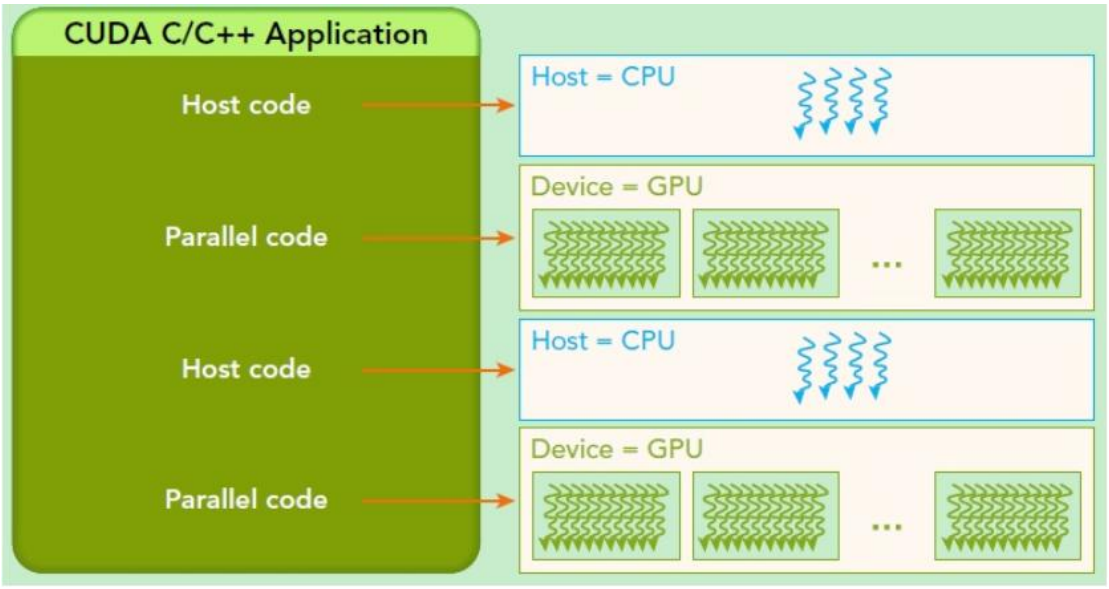

图 2：CUDA 编程模型

（2）CUDA 的执行模型

CUDA 执行流程中最重要的一个过程是调用 CUDA 的核函数 kernel 来执行并行计算。在 CUDA 程序构架中，主机端代码部分在 CPU 上执行；当遇到数据并行处理的部分，CUDA 就会将程序编译成 GPU 能执行的程序，并传送到 GPU，这个程序在 CUDA 里称做核（kernel）。

- **网格 grid**：kernel 在 device 上执行时，实际上是启动很多线程，一个 kernel 所启动的所有线程称为一个网格（grid），同一个网格上的线程共享相同的全局内存空间。grid 是线程结构的第一层次。

• **线程块 block**：网格又可以分为很多线程块（block），一个 block 里面包含很多线程。各 block 是并行执行的，block 间无法通信，也没有执行顺序。
• **线程 thread**：一个 CUDA 的并行程序会被以许多个 threads 来执行。数个 threads 会被群组成一个 block，同一个 block 中的 threads 可以同步，也可以通过 shared memory 通信。
• **线程束 warp**：GPU 执行程序时的调度单位，SM 的基本执行单元。目前在 CUDA 架构中，warp 是一个包含 32 个线程的集合，这个线程集合被“编织在一起”并且“步调一致”的形式执行。同一个 warp 中的每个线程都将以不同数据资源执行相同的指令，这就是所谓 SIMT 架构(Single-Instruction, Multiple-Thread，单指令多线程)。

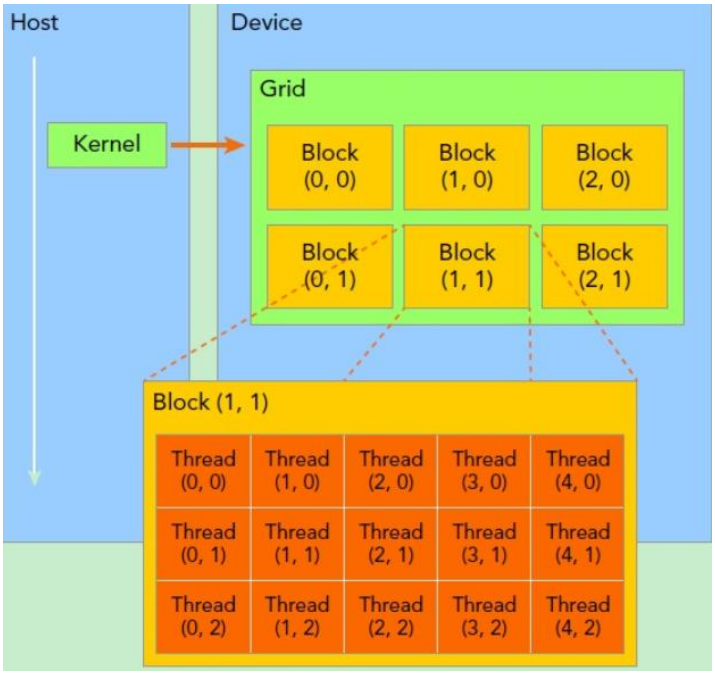

图 3：CUDA 的执行模型

（3）CUDA 的存储模型

• **SP**：最基本的处理单元，streaming processor，也称为 CUDA core。最后具体的指令和任务都是在 SP 上处理的。GPU 进行并行计算，也就是很多个 SP 同时做处理。
• **SM**：GPU 硬件的一个核心组件是流式多处理器（Streaming Multiprocessor）。SM 的核心组件包括 CUDA 核心、共享内存、寄存器等。SM 可以并发地执行数百个线程。一个 block 上的线程是放在同一个流式多处理器（SM）上的，因而，一个 SM 的有限存储器资源制约了每个 block 的线程数量。

一个 kernel 实际会启动很多线程，这些线程是逻辑上并行的，但是网格和线程块只是逻辑划分，SM 才是执行的物理层，在物理层并不一定同时并发。SM 要为每个 block 分配 shared memory，而也要为每个 warp 中的线程分配独立的寄存器。所以 SM 的配置会影响其所支持的线程块和线程束并发数量。所以 kernel 的 grid 和 block 的配置不同，性能会出现差异。还有，由于 SM 的基本执行单元是包含 32 个线程的 warp，所以 block 大小一般要设置为 32 的倍数。

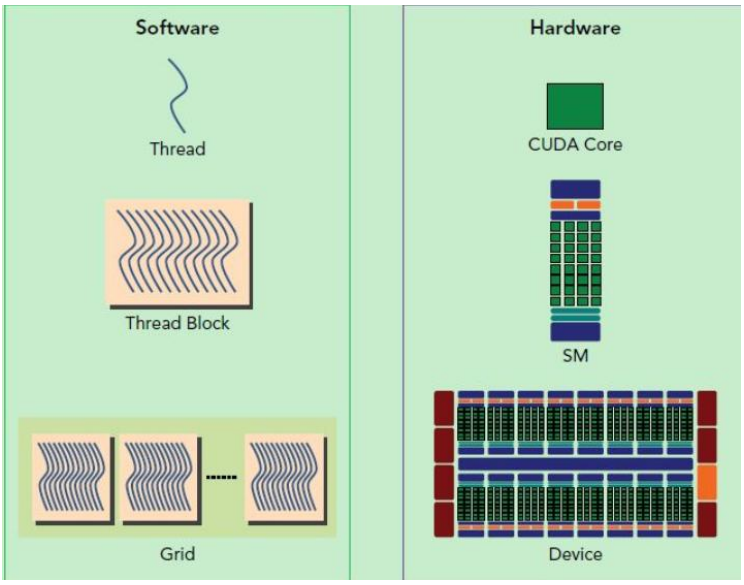

图 4：CUDA 存储模型

2.4.2 cuDNN 介绍

cuDNN（CUDA Deep Neural Network）是 CUDA 的深度神经网络库，用于深度神经网络的 GPU 加速基元库。cuDNN 为标准例程提供了高度优化的实现，例如向前和向后卷积，池化，规范化和激活层。

全球的深度学习研究人员和框架开发人员都依赖 cuDNN 来实现高性能 GPU 加速。它使他们可以专注于训练神经网络和开发软件应用程序，而不必花时间在底层 GPU 性能调整上。cuDNN 的加快广泛使用的深度学习框架，包括 Caffe2，Chainer，Keras，MATLAB，MxNet，PyTorch 和 TensorFlow。

3. 方法设计

3.1 BasicVSR

《BasicVSR:The Search for Essential Components in Video Super-Resolution and Beyond》[18]是 2021 年 CVPR 中的文章。该篇文章提出了一个轻量且高表现性能的视频超分 framework——BasicVSR。BasicVSR 改进了传统 VSR 结构中的 propagation 和 alignment 部分，分别提出了一个双向视频流的循环结构以及基于 flow-based 的 feature-wise 对齐方法。此外，在 BasicVSR 的基础上，作者进一步对 propagation 和 aggregation 进行优化，产生了一个更高表现性能的 VSR 结构——IconVSR。

3.1.1 BasicVSR 组成

作者将视频超分任务分为四个部分：Propagation（传播）、Alignment（对齐）、Aggregation（聚合）、Upsampling（上采样）。

•Propagation：它决定了 VSR 如何去利用视频序列的信息，它可以将所有的 VSR

分为 Sliding-Window 和 Recurrent 两类

- **Alignment**：时间和空间上的关于内容对齐
- **Aggregation**：特征信息的聚合，或者可以说就是 Fusion，它旨在将对齐后的连续帧进行时间和空间上的特征融合
- **Upsampling**：上采样层，将融合后的特征信息转变成 HR 层级的信息

BasicVSR 对 Propagation 和 Alignment 做了新的设计，而对于 Aggregation 和 Upsampling 则利用之前的 VSR 的方法。具体而言，BasicVSR 的 Propagation 使用了 Bidirectional(双向)循环机制，分为前向分支和后向分支，将整个输入序列的所有信息都加入到后续的对齐中；而对齐子网络使用 flow-based 方法，但对齐是 feature-wise 的，即使用光流估计，但是对齐是做在 feature map 上的；融合使用最基本的 concat(或者说就是 Early fusion)；上采样使用 ESPCN 提出的 PixelShuffle，即亚像素卷积层。BasicVSR 这种结构在性能和速度上都取得了很大的突破，证明了 BasicVSR 的可行性和轻量性。

BasicVSR 的具体网络结构如下：（其中 F 表示视频帧特征；U 表示上采样；S 表示光流估计；W 表示运动补偿；R 表示残差块）

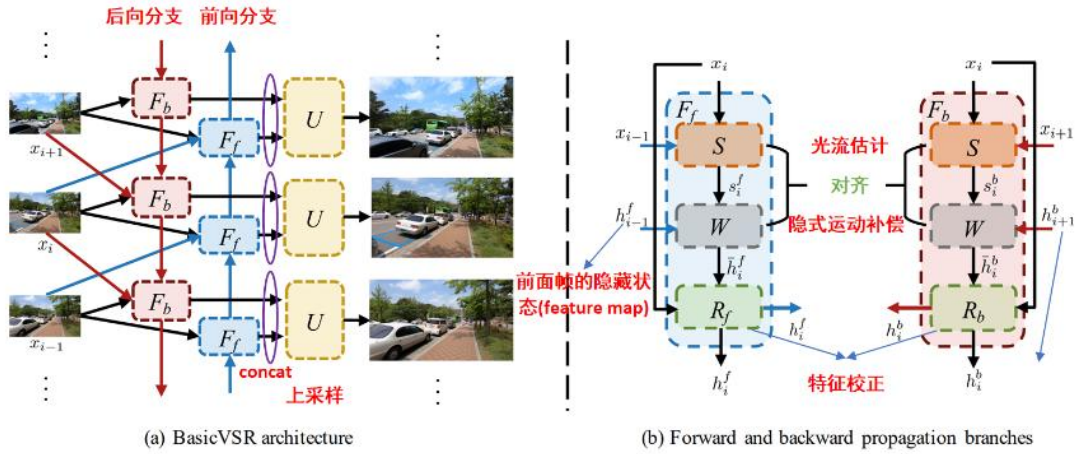

图 5：BasicVSR 网络结构

3.1.2 BasicVSR 结果

本节内容主要是展现 BasicVSR 的效果，图片均来自于论文《BasicVSR:The Search for Essential Components in Video Super-Resolution and Beyond》。

我们可以看到无论是在视觉效果上还是数据(峰值信噪比 PSNR)上，BasicVSR 都取得了最好的结果。（IconVSR 是 BasicVSR 的进阶版，所以效果更好，在此处不予讨论）

	Params (M)	Runtime (ms)	BI degradation			BD degradation		
			REDS4 [23]	Vimeo-90K-T [33]	Vid4 [21]	UDM10 [34]	Vimeo-90K-T [33]	Vid4 [21]
Bicubic	-	-	26.14/0.7292	31.32/0.8684	23.78/0.6347	28.47/0.8253	31.30/0.8687	21.80/0.5246
VESPCN [1]	-	-	-	-	25.35/0.7557	-	-	-
SPMC [29]	-	-	-	-	25.88/0.7752	-	-	-
TOFlow [33]	-	-	27.98/0.7990	33.08/0.9054	25.89/0.7651	36.26/0.9438	34.62/0.9212	-
FRVSR [25]	5.1	137	-	-	-	37.09/0.9522	35.64/0.9319	26.69/0.8103
DUF [16]	5.8	974	28.63/0.8251	-	-	38.48/0.9605	36.87/0.9447	27.38/0.8329
RBPN [9]	12.2	1507	30.09/0.8590	37.07/0.9435	27.12/0.8180	38.66/0.9596	37.20/0.9458	-
EDVR-M [32]	3.3	118	30.53/0.8699	37.09/0.9446	27.10/0.8186	39.40/0.9663	37.33/0.9484	27.45/0.8406
EDVR [32]	20.6	378	31.09/0.8800	**37.61/0.9489**	27.35/0.8264	39.89/0.9686	37.81/0.9523	27.85/0.8503
PFNL [34]	3.0	295	29.63/0.8502	36.14/0.9363	26.73/0.8029	38.74/0.9627	-	27.16/0.8355
MuCAN [20]	-	-	30.88/0.8750	37.32/0.9465	-	-	-	-
TGA [13]	5.8	-	-	-	-	-	37.59/0.9516	27.63/0.8423
RLSP [8]	4.2	49	-	-	-	38.48/0.9606	36.49/0.9403	27.48/0.8388
RSDN [12]	6.2	94	-	-	-	39.35/0.9653	37.23/0.9471	27.92/0.8505
RRN [14]	3.4	45	-	-	-	38.96/0.9644	-	27.69/0.8488
BasicVSR (ours)	6.3	63	31.42/0.8909	37.18/0.9450	27.24/0.8251	39.96/**0.9694**	37.53/0.9498	27.96/0.8553
IconVSR (ours)	8.7	70	**31.67/0.8948**	37.47/0.9476	**27.39/0.8279**	**40.03/0.9694**	**37.84/0.9524**	**28.04/0.8570**

图 6：BasicVSR 性能对比

3.2 基于 CUDA、cuDNN 加速的 BasicVSR

本文主要是在 Pytorch 框架下调用 CUDA 和 cuDNN 库，下面主要从代码层面介绍基于 CUDA、cuDNN 加速的 BasicVSR 模型。

3.2.1 输入模块

虽然 BasicVSR 主要用于视频的超分（恢复），但是图像也可看做单帧的视频，所以在本次试验中我分别设计了针对三种不同形式（随机模拟的视频、图像、视频）的输入的输入函数：input_lrs,input_image,input_video。

随机模拟视频是指通过 torch.randn 生成（n，t，c，h，w）形式的五维矩阵来代表视频帧。因为视频的也是这样表示的，其具体结构如下：

- n：每次输入的视频数目
- t：每个视频的帧数
- c：视频帧通道数
- h：视频帧高度
- w：视频帧宽度

```
lrs = torch.randn(5, 12, 3, 64, 64)
input_lrs(lrs,model,device,True)
```

图片的输入格式为（n，1，c，h，w），主要是通过 pillow 库读取之后转换为 tensor 格式。

```
def input_image(model,device,is_use_cuda=False):
    '''
        Input low quilty image, and return high quilty image
        Args:
            model: BasicVSR net
            device: server device
            is_use_cuda: use GPU or not (default: False)
    '''
    lrs=Image.open("/home/mawenzhuo/BasicVSR/images/ant.png")
    tran_totensor=transforms.ToTensor()
    lrs=tran_totensor(lrs)
    c,h,w=lrs.size()
    lrs=torch.reshape(lrs,[1,1,c,h,w])
    if is_use_cuda&(device==torch.device("cuda")):#使用cuda，且cuda可用
        lrs=lrs.to(device)
        model=model.to(device)
    rlt = model(lrs)
    print('HQ image size:',rlt.size())
    rlt=torch.reshape(rlt,[3,h*4,w*4])
    tran_PIL=transforms.ToPILImage()
    rlt=tran_PIL(rlt)
    rlt.save("/home/mawenzhuo/BasicVSR/images/ant_r.png")
```

视频主要是需要使用 opencv 进行一个帧分离（将视频拆分成多帧图像），计算完成后合并成图片。

```
#将视频转为一帧帧图片
cap = cv2.VideoCapture("/home/mawenzhuo/BasicVSR/frames/video.mp4")
fps = cap.get(cv2.CAP_PROP_FPS)  # 获取帧率
width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))  # 获取宽度
height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))  # 获取高度
suc = cap.isOpened()  # 是否成功打开
frame_count = 0
frames=[]
while suc:
    suc, frame = cap.read()
    if suc==False:
        break
    cv2.imwrite('/home/mawenzhuo/BasicVSR/frames/frame{}.png'.format(frame_count), frame)
    frames.append(frame)
    frame_count += 1
cap.release()
```

```
#将视频帧拼接成视频
rlt=rlt.permute([0,1,4,3,2]).detach().cpu().numpy().astype('uint8')
f = cv2.VideoWriter_fourcc(*'mp4v')
videoWriter = cv2.VideoWriter('/home/mawenzhuo/BasicVSR/frames/video_r.mp4',f,fps,(rlt.shape[2],rlt.shape[3]
for i in range(frame_count):
    frame=rlt[0,i,:,:,:]
    videoWriter.write(frame)
videoWriter.release()
```

3.2.2 CUDA、cuDNN 并行加速模块

本次实验中主要使用 pytorch 框架下 CUDA、cuDNN 调用 GPU 对程序进行并行加速。由于显存的原因，在大多数情况下我使用第 3 台 GPU 进行实验。

```
if is_use_cuda&(device==torch.device("cuda")):#使用cuda，且cuda可用
    lrs=lrs.to(device)
    model=model.to(device)
```

```
# torch.backends.cudnn.enabled = True
# torch.backends.cudnn.benchmark = True
start=time.time()
torch.cuda.set_device(2)
device=torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BasicVSR(spynet_path="/home/mawenzhuo/BasicVSR/spynet_weight.pth")#
lrs = torch.randn(5, 12, 3, 64, 64)
input_lrs(lrs,model,device,True)
# input_image(model,device,True)
# input_vedio(model,device,True)
end=time.time()
print('The total time consumption is {} s'.format(end-start))
```

3.2.3 BasicVSR 模块

本模块仅展示了关键代码部分，由于代码较多，详细代码可见附件（附详细注释）。

1>Propagation 部分

```
# Backward Propagation 后向分支
rlt = []#后向对齐特征序列
feat_prop = lrs.new_zeros(n, self.num_feat, h, w)#传播特征
for i in range(t-1, -1, -1):#逆向遍历每一帧
    curr_lr = lrs[:, i, :, :, :]#当前帧
    if i < t-1:
        flow = backward_flow[:, i, :, :, :]# flow estimation module（获取下一帧的光流）
        feat_prop = flow_warp(feat_prop, flow.permute(0, 2, 3, 1))# spatial warping module

    feat_prop = torch.cat([curr_lr, feat_prop], dim=1)#与当前帧融合
    feat_prop = self.backward_resblocks(feat_prop)# residual blocks
    rlt.append(feat_prop)#加入特征序列
rlt = rlt[::-1]#倒序
```

2>Alignment 部分

```
flow = forward_flow[:, i-1, :, :, :]# flow estimation module（获取上一帧的光流）
feat_prop = flow_warp(feat_prop, flow.permute(0, 2, 3, 1))# spatial warping module
```

3>Aggregation 部分

```
##前后向分支对齐特征融合
cat_feat = torch.cat([rlt[i], feat_prop], dim=1)
sr_rlt = self.lrelu(self.concate(cat_feat))
```

4>Upsampling 部分

```
#上采样
sr_rlt = self.lrelu(self.up1(sr_rlt))
sr_rlt = self.lrelu(self.up2(sr_rlt))
sr_rlt = self.lrelu(self.conv_hr(sr_rlt))
sr_rlt = self.conv_last(sr_rlt)
```

4. 实验与结果分析

4.1 实验环境和硬件条件

- 操作系统：Linux
- 编程语言：Python
- 编程环境：Python 3.7.13（conda 虚拟环境）
- CPU：15 个 CPU；Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz；4 核；

```
processor       : 15
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz
stepping        : 1
microcode       : 0xb000040
cpu MHz         : 1200.130
cache size      : 15360 KB
physical id     : 1
siblings        : 8
core id         : 3
cpu cores       : 4
apicid          : 23
initial apicid  : 23
fpu             : yes
fpu_exception   : yes
cpuid level     : 20
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall n
x pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx
 smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dno
wprefetch cpuid_fault epb cat_l3 cdp_l3 invpcid_single pti intel_ppin ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase ts
c_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdt_a rdseed adx smap intel_pt xsaveopt cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dth
erm ida arat pln pts md_clear flush_l1d
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs taa itlb_multihit mmio_stale_data
bogomips        : 7002.26
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

图 7：CPU 信息

- GPU：4 个 GPU；最高支持 CUDA11.4；显存如下
- 运行框架：Pytorch 1.10.1；torchvision 0.11.2
- 并行框架：cudatoolkit 11.3.1（CUDA 和 cuDNN 均使用 pytorch 自带库进行调用）
- 第三方库：opencv、pillow、time

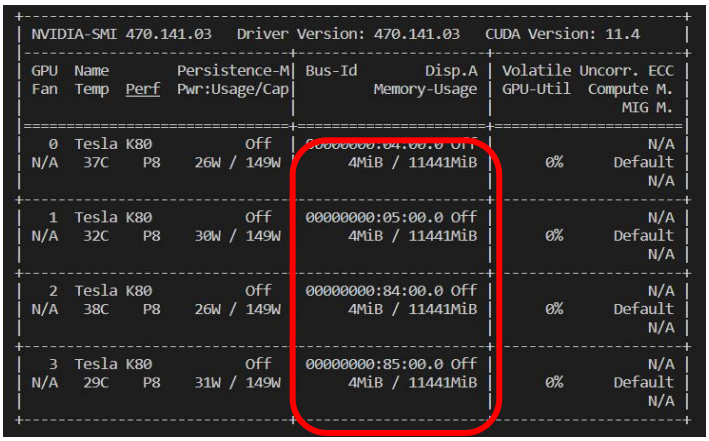

图 8：GPU 信息

4.2 不同并行策略适用性分析

常用的并行策略主要有数据并行、任务并行、数据流并行等，本节内容主要是分别分析各种常见的并行策略在本实验中的适用性，以解释本次实验选用**数据并行的并行策略**的原因。

（1）**数据并行**：将输入数据划分为一些子数据及，分发到个处理器上执行相同的操作。即以数据为中心，通过将数据集进行风格后在不同计算单元内并行执行相同的计算操作来达到提高性能的效果。这要求待处理的数据具有平等的特性，即几乎没有需要特殊处理的数据。其结构如下图所示。

本次实验当中，由于计算操作几乎为卷积运算，而最为庞大的是输入数据集（成千上万的视频），这恰好符合数据并行的要求：每组数据的操作相同、数据庞大好划分。事实上，几乎所有的深度学习任务都是基于数据并行的，因为以数据集为单位的输入数据非常适合划分为子数据集。

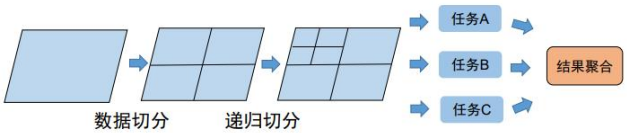

图 9：数据并行结构

（2）**任务并行**：也称之为功能并行或控制并行，侧重于将一个完整的任务分解为一些具有不同功能的子任务，调度到不同的处理器上并行执行。即以任务为中心，通过讲一个任务分成许多子任务在不同计算单元内并行执行。任务并行要求子任务之间一般互不相关，可以各自执行自己的操作。其结构如下图所示。

本次试验当中，虽然 BasicVSR 可以将其流程分为 Propagation、Alignment、Aggregation、Upsampling 这四个部分，但是这四个子任务相互关联密切，不能够独立执行，所以任务并行的方法并不适用于 BasicVSR 的加速。

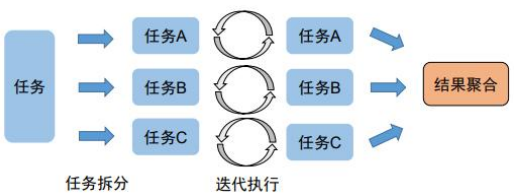

图 10：任务并行结构

（3）数据流并行：一个子任务的输出是另外一个子任务的输入，子任务之间构成一个流水线，虽然在流水线上多个阶段可以同时操作，但操作的是不同的数据，此时构成一类特殊的任务并行，即为数据流并行。

本实验中似乎四个子任务满足上述数据流并行的定义，但是仔细分析BasicVSR的网络就可以发现，Propagation、Alignment、Aggregation、Upsampling这四个部分并不是线性串联的，也就是说并不是上一个任务的输出就是这一个子任务的输入。在代码实现过程当中，自四个部分是相互交错的。具体的，数据被输入之后，在前向Propagation分支当中计算光流Alignment，然后将对其后的信息传输到后向Propagation当中在此进行光流Alignment，在后向Propagation中的最后一步对前后向Propagation的信息进行Aggregation，在整个模型的最后再进行Upsampling。所以，数据流并行并不适用于本实验。

4.3 不同并行编程模型适用性分析

常用的并行编程模型有OpenMP、MPI、MapReduce、GPU等，本节主要是分别介绍常用的并行模型并且讨论其在本实验中的适用性。

4.3.1 共享存储系统并行编程 OpenMP

OpenMP（Open Multi-Processing）是一种开放的面向共享内存模型的多线程并行应用程序编程接口。适用于单机、多核环境下的并行化编程，具有良好的可移植性，方便易用。

OpenMP执行模型采用Fork-Join形式：程序由主线程控制，在需要进行并行计算时，派生出子线程执行并行任务，并行任务结束后再回到主线程。其结构如下图所示。

事实上，OpenMP是可以用于BasicVSR一类的网络模型加速的，只不过由于OpenMP仅支持C++和Fortran语言，而本人使用python语言下的pytorch框架实现BasicVSR，导致使用OpenMP加速比较麻烦。

图 11：OpenMP模型结构

4.3.2 分布存储系统并行编程 MPI

MPI（Message Passing Interface）是一个消息传递编程模型库，其提供的接口方便C语言和Fortan77调用。在MPI中，用户必须显式地发送和接受消息来实现处理机间的数据交换，每个并行进程均有自己独立的地址空间，相互之间的访问不能直接进行，必须通过显式的消息传递实现。其编程模型如下图所示。

经过调研发现，MPI也是可以适用于深度学习框架下的并行加速，例如柏涛涛等基于深度神经网络与MPI并行计算的人脸识别算法研究[20]。这里我们也是由于语言的调用限制，没有选择MPI的并行加速模型。

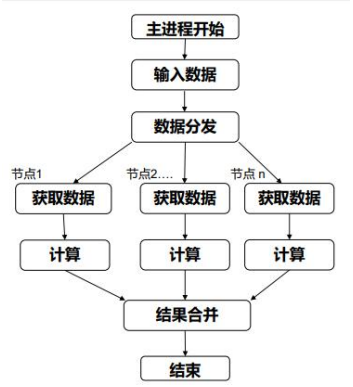

图 12：MPI 模型结构

4.3.3 基于 MapReduce 的分布式并行计算

MapReduce 是一种计算模式，用于大规模数据集的并行计算。其主要由三部分组成：编程模型、数据处理引擎、运行环境。其计算模式如下图。

• 编程模型：MapReduce 编程模型将问题抽象成 Map 和 Reduce 两个阶段

•数据处理引擎：由 MapTask 和 ReduceTask 组成，分别负责 Map 阶段逻辑和 Reduce 阶段罗的处理

• 运行环境：由一个 JobTracker 和若干个 TaskTracker 两类服务组成，其中 JobTracker 负责资源管理和作业控制，而 TaskTracker 负责接收来自 JobTracker 的命令并执行

显而易见的，MapReduce 模式也能应用于深度学习框架之下，例如庞雪等基于 MapReduce 的深度学习混合模型文本分类研究[21]。事实上，基本常用的并行模型都能使用到深度学习网络当中去，我们所需要考虑的是基于实验项目其使用的便捷性和性能好坏。也是从这两个方面的考虑，没有选择 MapReduce。

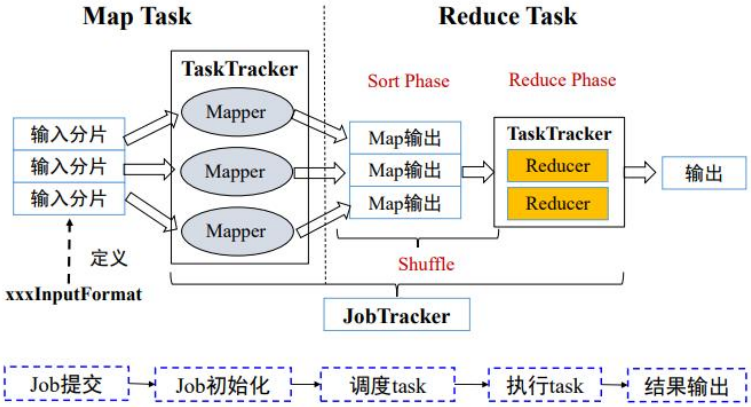

图 13：MapReduce 模型结构

4.3.4 通用计算 GPU

关于 CPU 和 GPU 的介绍在第二章中已经详细介绍了，这里主要探讨为什么本实验选用 GPU 加速的原因，以及选用 CUDA 和 cuDNN 库的原因。

CPU 是中央处理单元，GPU 是图形处理单元，GPU 由上千个流处理器(core)作为运算器。执行采用单指令多线程(SIMT)模式。相比于单核 CPU（向量机）流水线式的串行操作，虽然 gpu 单个 core 计算能力很弱，但是通过大量线程进行同时计算，在数据量很大是会活动较为可观的加速效果。

具体到 cnn，利用 gpu 加速主要是在 conv（卷积）过程上。conv 过程同理可

以像以上的向量加法一样通过 cuda 实现并行化。具体的方法很多，不过最好的还是利用 fft（快速傅里叶变换）进行快速卷积。NVIDIA 提供了 cufft 库实现 fft，复数乘法则可以使用 cublas 库里的对应的 level3 的 cublasCgemm 函数。

GPU 加速的基本准则就是“人多力量大”。CNN 说到底主要问题就是计算量大，但是却可以比较有效的拆分成并行问题。随便拿一个层的 filter 来举例子，假设某一层有 n 个 filter，每一个需要对上一层输入过来的 map 进行卷积操作。那么，这个卷积操作并不需要按照线性的流程去做，每个滤波器互相之间并不影响，可以大家同时做，然后大家生成了 n 张新的谱之后再继续接下来的操作。既然可以并行，那么同一时间处理单元越多，理论上速度优势就会越大。所以，处理问题就变得很简单粗暴，就像 NV 那样，暴力增加显卡单元数（当然，显卡的架构、内部数据的传输速率、算法的优化等等也都很重要）。

GPU 主要是针对图形显示及渲染等技术的出众，而其中的根本是因为处理矩阵算法能力的强大，刚好 CNN 中涉及大量的卷积，也就是矩阵乘法等，所以在这方面具有优势。

回归到 BasicVSR，其本质就是 CNN，只不过为了增加视频恢复的效果，增加了一些其他的模块。在 BasicVSR 中，无论是在 Propagation、Alignment 模块还是在 Aggregation、Upsampling 模块，都有大量的卷积操作，或者说卷及操作是该算法的基础单元。正是因为这样的特性，使用 GPU 加速 BasicVSR 算法就合情合理了。

本文考虑到并行模型的成熟性和方便性以及和硬件的契合性，选用 CUDA 库和 cuDNN 库对 BasicVSR 并行加速。

4.4 结果、性能分析

4.4.1 结果分析

我们以如下图尺寸的视频进行输入，得到如下结果：

```
lrs = torch.randn(1, 60, 3, 64, 64)
input_lrs(lrs,model,device,True)
```

- 使用 cpu 计算结果如下：

```
torch.Size([1, 60, 3, 256, 256])
The total time consumption is 14.414279699325562 s
```

- 使用 GPU 加速解算结果如下：

```
torch.Size([1, 60, 3, 256, 256])
The total time consumption is 6.09595251083374 s
```

- 使用 cuDNN 进一步加速计算结果如下：

```
torch.Size([1, 60, 3, 256, 256])
The total time consumption is 7.335965394973755 s
```

从上面的结果我们可以看到，使用 cuda 调用 GPU 进行加速运算是有效的！**加速比为 2.071**。同时，使用 cuDNN 库进行进一步加速，虽然其相对于仅仅使用 CPU 的性能更好一些，加速比达到了 1.964。但是不难发现，使用 cuDNN 进一步加速模型居然比仅仅使用 CUDA 调用 GPU 的效果差。对此我做了更加深入的探讨：

如下图所示，这是我对同一尺寸（1,40,3,64,64）的视频分别使用 CPU、CUDA、

cuDNN 各计算 10 次得到的耗时曲线图。x 轴为模型计算的次数，y 轴为时间消耗。

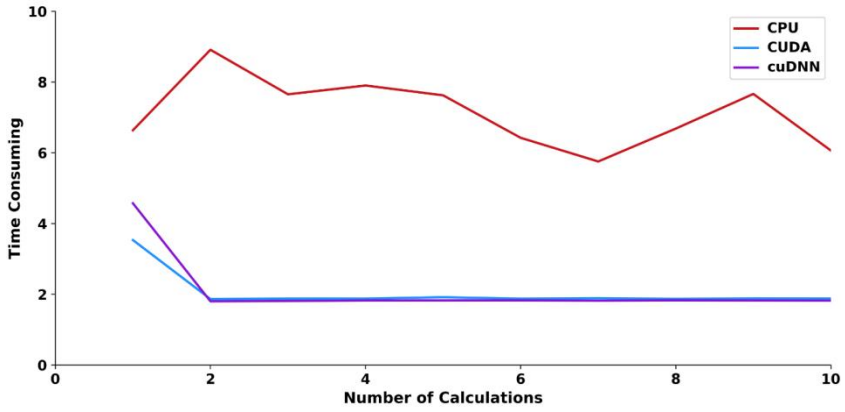

图 14：相同视频的重复计算耗时曲线

非常有趣的事情发生了，我们根据这各实验，可以得出以下结论（我还做了其他相关的实验对其进行验证，最终结果得到了证实）：

①不难看出，尽管模型的参数、输入数据均没有发生变化，但是使用 CPU 进行计算时耗时波动很大。这是因为 CPU 除了需要处理运行时的计算，其还需要兼顾电脑其他的任何活动的计算。因此，由于每次运行时电脑内存分布情况不一，所以导致其耗时不一。相反，GPU 由于主要处理图形相关的大数据量计算，一般情况下 GPU 的调用比较少，所以每次运行模型时其 GPU 情况基本相差不大，导致了其耗时稳定。

②根据上面的结果异常，我在思考为什么 cuDNN 在 CUDA 的基础上进一步加速的效果居然没有单纯使用 CUDA 效果好？这个实验给我们了很好的解释。事实上，无论是 CUDA 调用 GPU 还是再加上 cuDNN 去进行进一步优化，在程序开始的时候都需要加载一些全局变量以及显卡驱动，所以相对于后面计算时更高的加速比，第一次计算模型的时候会有额外的耗时。但是为什么 cuDNN 的额外耗时更长呢？这是因为 cuDNN 库进一步加速在第一次计算时会搜索全局网络来寻找最高效的计算路径，以达到加速的目的，但是很显然这个搜索的过程会耗费一段时间，所以这也是为什么其额外耗时大于 CUDA 的原因。

③比较容易发现，其实对于该视频的计算而言，增加了 cuDNN 进行加速后，其实时间消耗也就短了平均 0.03s 左右，这是为什么呢？根据调研，我总结了以下两点可能的原因：

1>数据量太少，无法体现明显的区别。

2>由于我使用的是 pytorch 框架下 CUDA、cuDNN，其中可能已经经过一些“不为人知”的优化，导致无法在性能上体现出差异。

4.4.2 加速比分析

为了分析其加速比，我做了这样的实验：**不断增加数据量（每次增加 3 帧视频帧），并且每个数据都进行三种方法的运算（CPU、CUDA、cuDNN）**，最终计算他们的耗时、加速比，得到如下曲线。

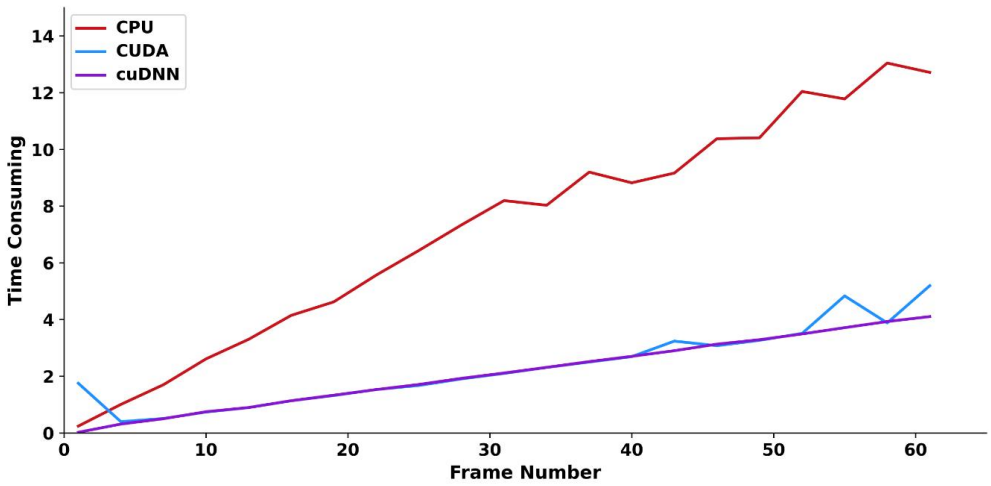

图 15：不同视频计算耗时曲线

由上图可知：

①CUDA 和 cuDNN 都有着比较好的加速结果，当数据量较大时，其可以节省接近 2/3 的耗时

②当数据量较少时，如上图中视频尺寸为（1,1,3,64,64）时，使用 CUDA 加速还没有仅使用 CPU 进行计算快。这是因为在数据量太小时，计算的耗时比数据从 Host 传输到 Device 的耗时以及配置 GPU 耗时都要短，导致单纯使用 CPU 计算更快。

③为什么这里使用 cuDNN 进一步加速时，第一次计算耗时特别短？这是因为在做实验的时候是先进行 CUDA 加速试验，然后接着进行 cuDNN 进一步加速。所以在之前 CUDA 配置好之后，cuDNN 不需要额外耗时去进行配置，只需要计算就可以了。

下图分别绘制出单独使用 CUDA 和加上 cuDNN 两种方法的加速比随数据量增加的变化曲线。

①整体来讲（除开第一个数据的异常）两者的加速比是比较接近的，都在 3.5 左右，加速效果很好

②在数据量比较大的时候，cuDNN 进一步加速带了了更好的加速效果

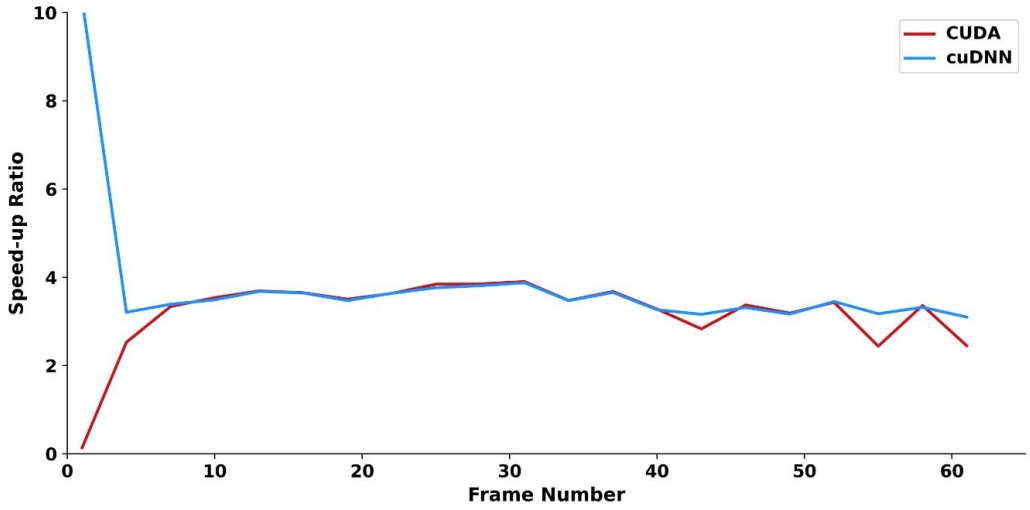

图 16：不同视频计算加速比

4.4.3 内存分析

为了更好地理解模型运行过程中的计算模式，我使用 memory_profiler 库对模型进行了内存分析。（下图和下表为视频尺寸为[1,40,3,64,64]作为输入得到的内存曲线和具体内存数据）。

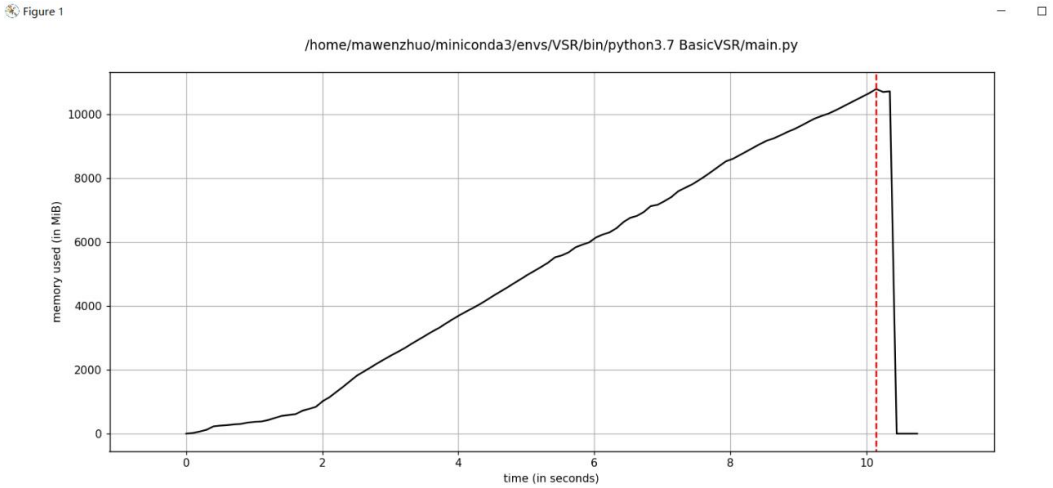

图 17：内存消耗曲线

由上图可以看到，除了最开始内存增加较为平缓之外，后续过程中内存的消耗增加比较固定。这说明除开开始的数据准备阶段，后续的运行模块中每个部分的计算量分布是比较平均的（事实上，根据代码分析发现，每个模块中其实就是对输入的数据特征序列进行卷积操作，所以内存开销变化不大）。

85	774.9 MiB	394.1 MiB	1		forward_flow, backward_flow = self.comp_flow(lrs)
86	774.9 MiB	0.0 MiB	1		end=time.time()
87	774.9 MiB	0.0 MiB	1	Alignment	time_consming.append(end-start)
88					
89					# forward_flow = rearrange(forward_flow, 'n t c h w -> t n h w c').contiguous()
90					# backward_flow = rearrange(backward_flow, 'n t c h w -> t n h w c').contiguous()
91					# lrs = rearrange(lrs, 'n t c h w -> t n c h w').contiguous()
92					
93					# Backward Propagation 后向分支
94	774.9 MiB	0.0 MiB	1		start=time.time()
95	774.9 MiB	0.0 MiB	1		rlt = []#后向对齐特征序列
96	774.9 MiB	0.0 MiB	1		feat_prop = lrs.new_zeros(n, self.num_feat, h, w)#传播特征
97	5270.3 MiB	0.0 MiB	41		for i in range(t-1, -1, -1):#逆向遍历每一帧
98	5151.4 MiB	0.0 MiB	40		curr_lr = lrs[:, i, :, :, :]#当前帧
99	5151.4 MiB	0.0 MiB	40		if i < t-1:
100	5151.4 MiB	0.0 MiB	39		flow = backward_flow[:, i, :, :, :]# flow estimation module (获取下一帧的光流)
101	5151.4 MiB	19.7 MiB	39		feat_prop = flow_warp(feat_prop, flow.permute(0, 2, 3, 1))# spatial warping module
102					
103	5152.4 MiB	39.9 MiB	40		feat_prop = torch.cat([curr_lr, feat_prop], dim=1)#与当前帧融合
104	5270.3 MiB	4435.6 MiB	40		feat_prop = self.backward_resblocks(feat_prop)# residual blocks
105	5270.3 MiB	0.2 MiB	40		rlt.append(feat_prop)#加入特征序列
106	5270.3 MiB	0.0 MiB	1	Propagation	rlt = rlt[::-1]#倒序
107	5270.3 MiB	0.0 MiB	1		end=time.time()
108	5270.3 MiB	0.0 MiB	1		time_consming.append(end-start)
109					
110					# Forward Propagation 前向分支
111	5270.3 MiB	0.0 MiB	1		start=time.time()
112	5271.7 MiB	1.4 MiB	1		feat_prop = torch.zeros_like(feat_prop)#传播特征
113	10775.7 MiB	0.0 MiB	41		for i in range(0, t):#正向遍历每一帧
114	10661.0 MiB	0.0 MiB	40		curr_lr = lrs[:, i, :, :, :]#当前帧
115	10661.0 MiB	0.0 MiB	40		if i > 0:
116	10661.0 MiB	0.3 MiB	39		flow = forward_flow[:, i-1, :, :, :]# flow estimation module (获取上一帧的光流)
117	10661.0 MiB	2.4 MiB	39	Aggregation	feat_prop = flow_warp(feat_prop, flow.permute(0, 2, 3, 1))# spatial warping module
118					
119	10661.0 MiB	1.3 MiB	40		feat_prop = torch.cat([curr_lr, feat_prop], dim=1)#与当前帧融合
120	10715.1 MiB	1901.6 MiB	40		feat_prop = self.forward_resblocks(feat_prop)# residual blocks
121					
122					# Fusion and Upsampling
123					##前后向分支对齐特征融合
124	10717.2 MiB	82.7 MiB	40		cat_feat = torch.cat([rlt[i], feat_prop], dim=1)
125	10719.0 MiB	84.1 MiB	40		sr_rlt = self.lrelu(self.concate(cat_feat))
126					#上采样
127	10727.2 MiB	365.9 MiB	40		sr_rlt = self.lrelu(self.up1(sr_rlt))
128	10759.2 MiB	1664.8 MiB	40		sr_rlt = self.lrelu(self.up2(sr_rlt))
129	10774.9 MiB	1186.8 MiB	40	Upsampling	sr_rlt = self.lrelu(self.conv_hr(sr_rlt))
130	10774.9 MiB	209.1 MiB	40		sr_rlt = self.conv_last(sr_rlt)

图 18：内存消耗分布

通过上图可以看到每一个过程消耗的内存具体数据，我们可以发现内存主要是消耗在需要大量卷积的操作之上。事实上，我发现在整个模型的计算过程当中，消耗内存最多的不是计算过程，而是数据的传输。从 Host 将大量数据传输到 Device 需要占用很大的资源。

4.4.4 网络流分析

为了更好的了解网络中每个模块的计算速度，我对 BasicVSR 中的额四个模块进行了耗时统计（如下表）。每种方法各统计三次，取平均值得到如下饼状图。

表 1：分模块耗时统计表

方法\耗时\模块	Alignment	Propagation	Aggregation	Upsampling
CPU	3.012	7.469	0.661	3.309
CPU	3.161	7.231	0.716	3.299
CPU	2.998	7.615	0.659	3.413
CUDA	1.032	3.244	0.492	1.398
CUDA	1.106	3.236	0.491	1.301
CUDA	0.973	3.369	0.517	1.299
cuDNN	1.001	2.167	0.423	1.297
cuDNN	1.097	2.291	0.399	1.266
cuDNN	1.121	2.089	0.441	1.296

通过分析可以得出以下结论：

①在 Alignment、Propagation、Aggregation、Upsampling 四个模块中，Propagation 模块耗时最长，计算量最大。

②使用 CUDA 加速，其对于每个模块的加速比较平均

③对比饼状图和表格，可以发现：cuDNN 进一步加速的地方主要在 Propagation 模块（图中可以看到，加上 cuDNN 后 Propagation 模块权重下降）。

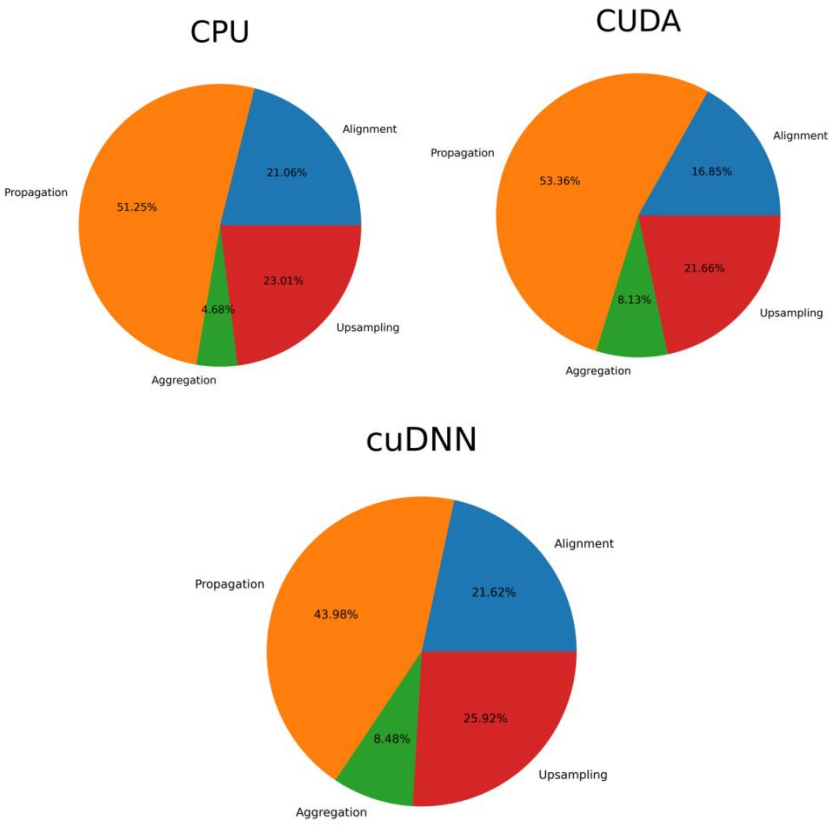

图 19：模块耗时占比

4.5 优化途径分析

通过上面的实验分析，其实可以发现当前 BasicVSR 加速实验存在的问题：加速比太小，一般而言使用 GPU 进行加速的加速比不会只在 3-4 左右。由此我进行了比较深入的分析，跟踪了整个代码的内存和时间损耗在各个阶段的占比（详细见 4.4.3 和 4.4.4），得出以下优化途径。

①硬件优化：虽然本次实验的硬件设备设备非常优越（实验室的服务器），但是由于资源分配的限制，导致个人所能使用的显存非常少，虽然有四块 GPU 但是本次实验只使用了一个 GPU 的一小部分显存有（资源限制）。后期可以使用多块 GPU 同时进行并行计算，以从硬件方面优化，提高加速比。

②方法优化：

1>本次实验主要采用的是数据并行策略。虽然数据流并行并不适用于本实验（原因见 4.2),但根据 3.1 当中 BasicVSR 的结构,我们会发现虽然由于 Propagation、Alignment、Aggregation 三个模块的交错导致纯粹的数据流并行无法使用，但是由于 Upsampling 模块和前面三个模块是线性串接的，因此可以使用一种数据并行+数据流并行的混合并行策略来进行优化。具体而言，将输入数据集划分为多个子数据集通过 GPU 进行数据并行，同时将 Propagation、Alignment、Aggregation 作为一个整体，将其和 Upsampling 组成一个数据流，使用数据流并行策略进一步加速。

2>通过内存分析和时间损耗分析（详细见 4.4.3 和 4.4.4），我们会发现其实大部分的内存资源都消耗在从 Host 到 Device 的数据传输上面。如何将这部分的时间消耗或者内存损耗降下来，是使用 GPU 加速的重中之重。当然其实如何降低通信和数据传输的资源消耗是整个并行计算的难题。

③代码优化：由于本次实验是本人复现的 BasicVSR 代码，导致很多模块写的并不如源代码高效，甚至导致了很多冗余计算。同时，这样不太模块化的代码给我在并行加速时带来了困扰，例如，有些预处理部分的卷积运算无法放进 GPU 当中并行加速，我认为这是本次实验加速比比较低的主要原因。优化途径即为重构代码，使代码更加结构化，以便于整个模型的并行加速。

5. 结论

5.1 实验结论

本文对经典的视频超分模型 BasicVSR 进行复现。通过对不同并行策略、不同并行模型的分析讨论，最终选用 CUDA、cuDNN 加速 BasicVSR 网络。实验结果表明，其加速比大约在 3.5 左右。本文还从结果、加速比、内存、网络流等方面对实验进行了全面深入的分析,解释了一些异常现象以及 cuDNN 进一步加速效果不太理想的原因。进一步的，针对加速比太低的事实，本文进行深入地分析，提出了硬件、方法、代码三个层面的优化途径。

5.2 课程结语

经过一个学期的学习，《高性能地理计算》也进入了尾声阶段。这门课程主要讲授了地理计算领域的并行计算基础、地理计算算法、高性能空间数据处处等方面的内容。通过本课程的学习，一个基本的高性能地理计算的框架雏形已然搭建，剩下的只需要后期在不断深入学习当中不断填充即可。随着大数据时代的来临，无论什么邻域，高性能计算永远是避不开的话题。如何以更少的代价去完成更多的任务是本门课程的目标，同时也会大数据时代下每一位数据工作者所需要攻克的难题。

致谢

时光荏苒，岁月如梭。一转眼已到课程结束之际，在这里特别感谢乐鹏老师以及各位研究生学长的细心教导，是你们的倾囊相授让我在学业的道路上更进一步。

参考文献

[1] Yan Huang, Wei Wang, and Liang Wang. Bidirectional recurrent convolutional networks for multi-frame super-resolution. In NeurIPS, 2015. 2, 4

[2] Ce Liu and Deqing Sun. On bayesian adaptive video super-resolution. TPAMI, 2014. 2, 6, 7, 10, 13

[3] Hiroyuki Takeda, Peyman Milanfar, Matan Protter, and Michael Elad. Super-resolution without explicit subpixel motion estimation. TIP, 18(9):1958–1975, 2009. 2

[4] Peng Yi, Zhongyuan Wang, Kui Jiang, Junjun Jiang, and Jiayi Ma. Progressive fusion video super-resolution network via exploiting non-local spatio-temporal correlations. In ICCV, 2019. 1, 2, 6, 7, 10, 13

[5] Wenbo Li, Xin Tao, Taian Guo, Lu Qi, Jiangbo Lu, and Jiaya Jia. MuCAN: Multi-correspondence aggregation network for video super-resolution. In ECCV, 2020. 2, 6, 7

[6] Takashi Isobe, Xu Jia, Shuhang Gu, Songjiang Li, Shengjin Wang, and Qi Tian. Video super-resolution with recurrent structure-detail network. In ECCV, 2020. 2, 3, 4, 6, 7, 10

[7] Takashi Isobe, Songjiang Li, Xu Jia, Shanxin Yuan, Gregory Slabaugh, Chunjing Xu, Ya-Li Li, Shengjin Wang, and Qi Tian. Video super-resolution with temporal group attention. In CVPR, 2020. 2, 3, 6, 7

[8] Jose Caballero, Christian Ledig, Aitken Andrew, Acosta Alejandro, Johannes Totz, Zehan Wang, and Wenzhe Shi. Realtime video super-resolution with spatio-temporal networks and motion compensation. In CVPR, 2017. 2, 6, 7

[9] Wei-Sheng Lai, Jia-Bin Huang, Narendra Ahuja, and Ming Hsuan Yang. Deep laplacian pyramid networks for fast and accurate super-resolution. In CVPR, pages 5835–5843, 2017. 6, 10

[10] Tianfan Xue, Baian Chen, Jiajun Wu, Donglai Wei, and William T Freeman. Video enhancement with task-oriented flow. IJCV, 2019. 2, 4, 5, 6, 7, 10, 12

[11] Yapeng Tian, Yulun Zhang, Yun Fu, and Chenliang Xu. TDAN: Temporally deformable alignment network for video super-resolution. In CVPR, 2020. 2

[12]Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In ICCV, 2017. 2

[13]Xizhou Zhu, Han Hu, Stephen Lin, and Jifeng Dai. Deformable convnets v2: More deformable, better results. In CVPR, 2019. 2

[14]Xintao Wang, Kelvin C.K. Chan, Ke Yu, Chao Dong, and Chen Change Loy. EDVR: Video restoration with enhanced deformable convolutional networks. In CVPRW, 2019. 1, 2, 3, 6, 7, 10

[15]Younghyun Jo, Seoung Wug Oh, Jaeyeon Kang, and Seon Joo Kim. Deep video super-resolution network using dynamic upsampling filters without explicit motion compensation. In CVPR, 2018. 2, 6, 7

[16]Takashi Isobe, Xu Jia, Shuhang Gu, Songjiang Li, Shengjin Wang, and Qi Tian. Video super-resolution with recurrent structure-detail network. In ECCV, 2020. 2, 3, 4, 6, 7, 10

[17]akashi Isobe, Fang Zhu, and Shengjin Wang. Revisiting temporal modeling for video super-resolution. In BMVC, 2020. 2, 3, 4, 6, 7

[18]Kelvin C.K. Chan, Xintao Wang, Ke Yu, Chao Dong, and Chen Change Loy. BasicVSR: The search for essential components in video super-resolution and beyond. In CVPR, 2021. 1, 2, 3, 5, 8, 10

[19]乐鹏.高性能地理计算.武汉大学.2021.6

[20]柏涛涛.基于深度神经网络与 MPI 并行计算的人脸识别算法研究[J].西安文理学院学报(自然科学版),2020,23(02):62-67.

[21]庞雪. 基于 MapReduce 的深度学习混合模型文本分类研究[D].齐鲁工业大学,2019.