研究生课程考试成绩单

（试卷封面）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 院 系 | 计算机科学与工程 | 专业 | | 计算机科学与技术 | | |
| 学生姓名 | 卓旭 | 学号 | | 212138 | | |
| 课程名称 | 数字图像处理进展 | | | | | |
| 授课时间 | 2021年 12月至2021年12月 | | 周学时 | 3 | 学分 | 1 |
| 简  要  评  语 |  | | | | | |
| 考核论题 | 非局部均值滤波去噪算法的实现及对比实验 | | | | | |
| 总评成绩  （含平时成绩） |  | | | | | |
| 备注 |  | | | | | |

任课教师签名：

日期：

注：1. 以论文或大作业为考核方式的课程必须填此表，综合考试可不填。“简要评语”栏缺填无效。

2. 任课教师填写后与试卷一起送院系研究生秘书处。

3. 学位课总评成绩以百分制计分。

**（本页空白以备双面打印）**

**非局部均值滤波去噪算法的实现及对比实验**

——《数字图像处理进展》课程报告

212138 卓旭

**摘要**

非局部均值（Non-Local Means，NLM）算法是一种简单有效的图像去噪算法。其通过待去噪像素周围邻域与图像其他像素邻域的相似度度量加权，综合得出去噪结果。本报告主要研究了NLM算法，主要工作内容和目的有：

* 再次讨论了NLM算法的原理，加深了理解；
* 使用C++实现了CPU上执行的NLM算法，说明了实际实现中的问题；
* 使用CUDA实现了GPU上执行的并行加速的NLM算法，借此机会入门了CUDA编程；
* 介绍了使用积分图对NLM算法进行优化的策略；
* 将NLM算法与传统方法：高斯低通滤波和深度学习方法：DnCNN，在多种类型的噪声和图像上进行了对比实验。

本报告的所有实验相关代码后续均开源到<https://github.com/z0gSh1u/nlm-cuda>。

**一、图像去噪问题与NLM算法的原理**

图像的噪声是一种常见的图像降质原因，因此图像去噪问题是图像处理领域一直在研究的基础问题。常用的降质过程描述是：

其中是观测到的实际图像，是不含噪声的图像，是加性噪声。

常用的高斯低通、均值滤波等去噪方法是基于噪声主要占据图像频谱的高频部分的事实，削减高频，从而降低噪声水平。NLM算法则基于另一种进路：待去噪像素的一定邻域，应能在图像其他位置找到相似邻域，借助它们的信息可以更好地恢复待去噪像素。

NLM算法的具体做法的表达式如下：

其中是待去噪图像，是像素对待去噪像素的贡献权重，表示以为中心的一定窗口，且。原论文提供的权重的计算方式如下：

是归一化配分：

其中以以为中心的一定邻域，是控制去噪平滑程度的参数，是邻域内像素的欧氏距离和的平方。

NLM算法涉及到的“窗口”较多，本文的命名与记号为：将较小的“窗口”称为领域（Patch），将较大的“窗口”称为窗口（Window）。待去噪像素的邻域称为*SourcePatch*；比较像素的邻域称为*ComparePatch*；和拥有一样的半径，即边长为；搜索窗口称为*SearchWindow*，半径记为。在中滑动。示意图如下：

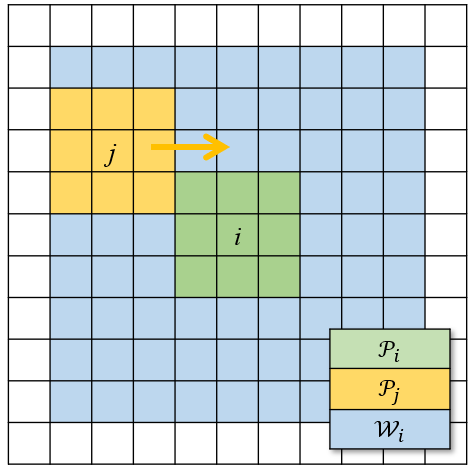


图 1 – NLM算法示意图

完成上述计算流程后，即可得到由一系列比较像素加权得到的去噪后像素的值。

**二、NLM算法的C++ CPU实现**

我首先使用C++实现了无特殊优化的NLM算法。本部分代码在nlm-cpu/nlm-cpu.cpp。本实验编写的程序处理目标均为单通道、像素8 bit深的单通道RAW图像，RAW图像内按小端序、行扫描顺序保存各像素值，无其他数据。图像的宽高、输入输出路径、NLM算法相关参数需要由配置文件提供（见NLMConfig.json），执行程序时将配置文件路径作为命令行参数。

首先，读入原图像，并将其转换为float型数据格式以便后续运算。常见做法将各像素值除以255归一化到[0, 1]后进行运算。但经尝试，本程序中保留在[0, 255]区间上能取得更好的处理效果，猜测可能原因是归一化后计算距离、权重时存在浮点误差累积的问题，归一化操作见uint8ImageToFloat函数。

与均值滤波等操作类似的，由于原图周围一圈的像素无法作为中心放下*SourcePatch*，为使得它们也得到去噪处理，需要先将原图向外补出一圈（上下左右各像素）。本文采用镜像对称法填补，比起补黑边效果要好。相关函数是padSourceImageSymmetric，示意图如下，带绿色边框的是原图，红色边框的是四个翅膀的填补，蓝色边框的是四角填补（依据行方向的翅膀）：



图 2 – 镜像对称法填补图像示意图

接下来，创建与原图等大的目标图像（结果图像），对其进行所谓的“反向映射”：遍历其中各个像素，按照第一节的NLM算法具体过程计算该像素的处理后值，最终即得到去噪后的图像。相关函数为NLMDenoise。其中有一些实现细节有必要说明：

① 由图1可知某时刻会有，此时未归一化权重。该权重远远大于其他比较像素得到的权重，这将导致原始像素被极大保留，去噪无法进行。本文的处理方式是暂时跳过，最后将时的权重设置为其他权重中的最大值。

② 在计算权重项中邻域内像素的欧氏距离和的平方时，将其除以了邻域尺寸，以控制指数的数量级对的稳定性。

计算完成后，将结果图像重新拉伸到[0, 255]灰度级并量化到8 bit位深，相关函数为floatImageToUint8。

使用CPU上运行的NLM算法对注入了零均值、标准差为的高斯噪声的512×512的Lena灰度图像进行去噪处理，参数为，，。结果图如下：

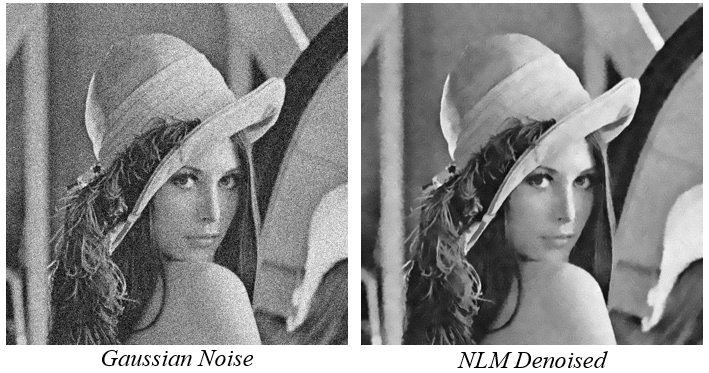


图 3 – NLM算法的处理结果示意图

在我的计算机（i5 7300HQ）上运行耗时为112秒，可见在没有进行多线程并行等优化的情况下，NLM算法的时间开销较大。

**三、NLM算法的CUDA GPU实现**

不难发现，NLM算法中，不同像素之间的去噪过程没有相关性，具备较好的并行执行可能性，而GPU的通用计算能力正适合处理此类任务。NVIDIA公司的CUDA技术允许我们使用类似于C++语言的方式进行CPU-GPU混合编程。本部分的相关代码在nlm-cuda/nlm-cuda.cu。

首先，对CUDA的架构和其中的基本概念进行简单说明：

* 宿主机被称为Host，GPU设备被称为Device，二者之间使用cudaMemcpy函数互传数据；
* 在Device上运行的函数需使用\_\_global\_\_标记，被称为Kernel函数；
* CUDA软件架构从小到大由Thread、Block和Grid组成，它们之间的组成关系如下图所示；
* Kernel函数会被映射到Thread上执行，可同时运行的最大Thread数量由GPU的计算能力（CC）决定。暂时无法调度上GPU运行的Block则串行排队。

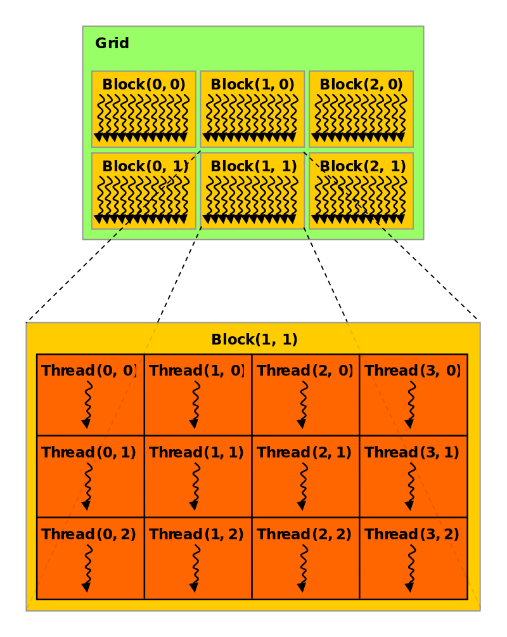


图 4 – CUDA软件架构示意图 [https://en.wikipedia.org/wiki/Thread\_block\_(CUDA\_programming)]

在图像处理领域，我们通常希望一个Thread对应一个像素的处理。因此，可按如下方式组织Thread与Block：

|  |
| --- |
| dim3 BlockThread(16, 16); // 每个Block安排16×16个Thread  // 总Block数如下计算，以确保每个像素都分配到Thread，padH和padW为宽高  dim3 GridBlock((padH + BlockThread.x - 1) / BlockThread.x,  (padW + BlockThread.y - 1) / BlockThread.y);  // Kernel函数调用时需传入并发配置信息  NLMDenoise<<<GridBlock, BlockThread>>>(/\* 其他参数 \*/); |

如此，每个Thread运行Kernel函数时即可按如下方式获知自己处理的像素坐标：

|  |
| --- |
| // 通过本Thread的Block号和Thread号即可求得映射到的像素坐标  int i = (blockIdx.x \* blockDim.x) + threadIdx.x;  int j = (blockIdx.y \* blockDim.y) + threadIdx.y;  // 由于求Block数时向上取整，可能出现多余映射，要排除  if (i >= srcH || j >= srcW) return; |

对于更细节的CUDA编程相关知识超出了本报告的讨论范围，此处略去不述。

除了NLMDenoise被改写为CUDA Kernel函数外，实现的其他环节与CPU版一致。执行前，图像被拷贝到显存中；执行完成后，拷贝回了内存。同样对第二节中的Lena图按相同配置进行NLM算法处理，获得了相同的处理结果，但执行时间降低到140毫秒，仅为CPU执行的0.125%。

需要说明的是，当我们使用了一些第三方库时，Debug与Release编译模式下得到的程序执行效率会有很大差异。但二、三节我实现的NLM算法没有依赖任何第三方库，均为原生代码编写，且计时部分仅为NLMDenoise函数，故尽管编译模式为Debug，比较仍然是正确和公平的。

**四、NLM算法的积分图优化法**

本节，将介绍一种使用积分图像（Integral Image，前缀和图像）对NLM算法进行原理层次的加速的优化策略。该方法没有在前面的程序中实现，故仅做简要介绍。

考虑二、三节实现的NLM算法的时间开销，若待处理图像尺寸为，则时间复杂度为：

（对每个像素点在每个*SearchWindow*内求每组Patch之间的权重）。

积分图像的基本思想是，对每个像素点，提前求出组成的矩形的像素和。则对于下示意图，可方便地求得任一矩形区域*ABCD*的像素和：

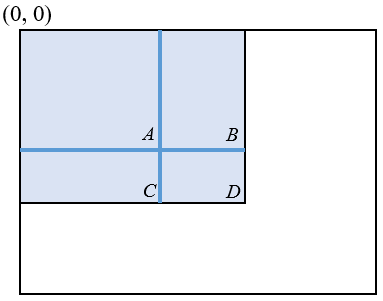


图 5 – 积分图像示意图

倘若我们按照下式提前构造一组关于“距离”的积分图像：

其中像素为积分图变量；像素为求和变量；表示落在原点与确定的矩形区域内（表示偏序小于）；为平移向量，取遍张成的二维空间的整数点。

那么有：

其中，。

这是一个常量时间的操作，而积分图的构建可以在时间递推得到。综上，NLM的时间复杂度优化为：

可以看到，效率的提升是很显著的。但是积分图的递推构建步骤不适合并行完成，故比起全程并行的CUDA实现，实际能快多少还有待实验。

**五、NLM算法与其他去噪方法的对比实验**

本节，我将实现的NLM算法与另外两种去噪算法：传统的高斯低通滤波器和深度学习的DnCNN，在三种性质的图像：Lena灰度图像、医学CT图像、彩色大尺寸自然图像上，注入了三种类型的噪声：高斯噪声、泊松噪声、椒盐噪声，对比了三种算法各自的去噪性能表现。进一步地，分析了NLM算法的参数调节的纲领性准则，以及几种去噪算法各自的优缺点。

首先，对本节用于对比NLM的两种去噪算法进行简单介绍。

**高斯低通滤波器**

基于噪声主要占据图像频谱的高频部分的事实，削减高频可以降低噪声水平。高斯低通滤波器的传递函数是高斯函数。零均值的二维高斯函数表达式为：

其中是高斯分布的标准差。这是一个连续函数，在进行卷积时需要将其离散化成一定尺寸的卷积核，如下图所示：

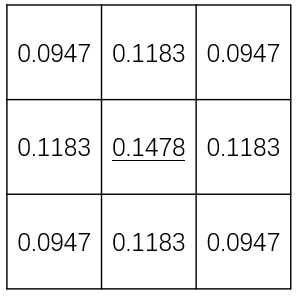


图 6 – 一个3×3的σ=1.5时的归一化的高斯核

随后使用该核对图像进行卷积操作即可。

本报告中的高斯低通滤波使用ImageJ软件自带的功能实现，其只需参数，核的半径是按照下式自动计算的：

其中为精度控制参数，默认取0.01。按该式计算的卷积核尺寸能保留绝大部分的高斯函数能量。

**DnCNN**

DnCNN是我们课上讲过的一种经典的基于深度学习的图像去噪模型。其网络结构如下图所示：

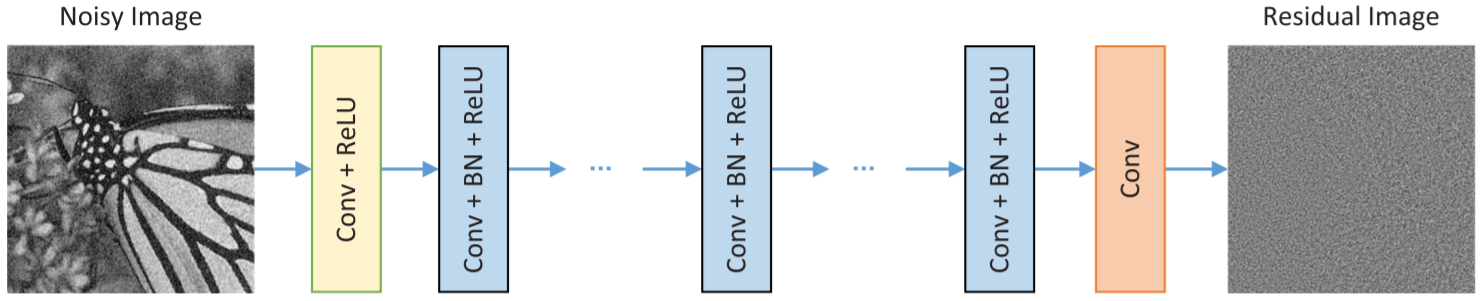


图 7 – DnCNN的网络结构

相比于直接学习出去噪后的图像，DnCNN选择学习其中的噪声，然后将噪声从原图中减去以实现去噪。这是因为待去噪图像的景物千变万化，而噪声模式相对单一，更好学习。本报告中的DnCNN相关实验使用官方的MATLAB代码和预训练模型完成。

接下来，介绍本节的三幅实验对象图像。这些图像均可在experiment/GroundTruth和experiment/Noisy目录下找到。

**Lena灰度图像**

图像尺寸为512×512。选择该图像的动机是，这是一幅常用的数字图像处理基准图像，其中的发丝等细节也能较好地考察去噪算法的性能。



图 8 – Lena灰度图像

**医学CT图像**

图像尺寸为512×512，像素8 bit位深，可视为加窗后图像。选择该图像的动机是，比起自然图像，医学图像对细节的保留要求更高，且性质与自然图像很不同，如不存在景深等。

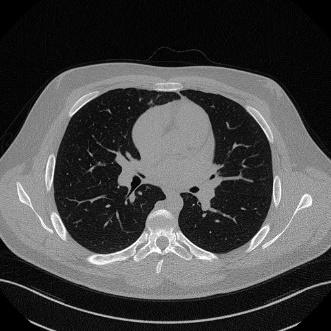


图 9 – 医学CT图像

**彩色大尺寸自然图像**

图像尺寸为2040×1356，为三通道彩色图像。选择该图像的动机是，考察各算法在彩色图像去噪上的效果，以及大尺寸图像上执行的性能。



图 10 – 彩色大尺寸自然图像

接下来，介绍本节注入的三种噪声。

**高斯噪声**

高斯噪声是符合高斯函数的噪声。本实验中使用ImageJ软件注入，均值为0，标准差为25。特定高斯噪声的产生可使用如下公式进行：

其中（服从标准正态分布），，而标准正态分布可从均匀分布中借助Box-Muller算法产生。

**泊松噪声**

泊松噪声是符合泊松分布的噪声。泊松分布的概率分布为：

其中被称为平均发生次数。严格来说，泊松噪声与像素值有关，不是加性噪声，不符合第一节提出的降质模型。

本实验中使用ImageJ软件的poisson-noise插件注入泊松噪声。其内部构造的随机过程用代码表示如下：

|  |
| --- |
| int poissonValue(float pixVal) { // pixVal为像素值  double L = exp(-pixVal);  int k = 0; double p = 1;  do {  k++; p \*= rand(); // rand函数返回[0, 1]之间的随机数  } while (p >= L);  return (k - 1);  } |

**椒盐噪声**

椒盐噪声表现为白点（盐噪声，最高灰度）或黑点（胡椒噪声，最低灰度）。本实验中使用ImageJ软件注入，密度约为16×16的区域内有13个噪点。

下面是去噪算法执行时的具体参数说明：

|  |  |
| --- | --- |
| 方法 | 参数 |
| NLM |  |
| 高斯低通滤波 |  |
| DnCNN |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **图像：Lena灰度图像**（PSNR/SSIM） | | | |
|  | NLM | 高斯低通滤波 | DnCNN |
| 高斯噪声 | 26.32/0.74 | 29.02/0.76 | **32.25/0.87** |
| 泊松噪声 | 23.42/0.84 | 30.68/0.85 | **35.77/0.92** |
| 椒盐噪声 | 23.31/0.55 | **27.95/0.72** | 20.81/0.42 |
| **图像：医学CT图像**（PSNR/SSIM） | | | |
|  | NLM | 高斯低通滤波 | DnCNN |
| 高斯噪声 | **26.47/0.56** | 23.90/0.47 | 24.98/0.50 |
| 泊松噪声 | 29.82/0.75 | 27.15/0.66 | **30.58/0.79** |
| 椒盐噪声 | 19.56/0.39 | **24.36/0.48** | 18.86/0.40 |
| **图像：彩色大尺寸自然图像**（PSNR） | | | |
|  | NLM | 高斯低通滤波 | DnCNN |
| 高斯噪声 | 27.79 | 26.45 | **28.81** |
| 泊松噪声 | 31.81 | 28.82 | **32.80** |
| 椒盐噪声 | 19.61 | 26.70 | **28.54** |

讨论TODO

下面给出实验的定性结果：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **图像：Lena灰度图像** | | | | | | |
| **原图** | | **高斯噪声图** | | **泊松噪声图** | | **椒盐噪声图** |
|  | |  | |  | |  |
|  | **NLM** | | **高斯低通滤波** | | **DnCNN** | |
| **高斯噪声** |  | |  | |  | |
| **泊松噪声** |  | |  | |  | |
| **椒盐噪声** |  | |  | |  | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **图像：医学CT图像** | | | | | | |
| **原图** | | **高斯噪声图** | | **泊松噪声图** | | **椒盐噪声图** |
|  | |  | |  | |  |
|  | **NLM** | | **高斯低通滤波** | | **DnCNN** | |
| **高斯噪声** |  | |  | |  | |
| **泊松噪声** |  | |  | |  | |
| **椒盐噪声** |  | |  | |  | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **图像：彩色大尺寸自然图像** | | | | | | |
| **原图** | | **高斯噪声图** | | **泊松噪声图** | | **椒盐噪声图** |
|  | |  | |  | |  |
|  | **NLM** | | **高斯低通滤波** | | **DnCNN** | |
| **高斯噪声** |  | |  | |  | |
| **泊松噪声** |  | |  | |  | |
| **椒盐噪声** |  | |  | |  | |

**六、总结与体会**

**附录**