摘 要

在图像质量评估研究中，使用度量算法来评估数字图像的清晰度，从而得出一个符合人类视觉感知的度量值，是一个非常有趣和具有挑战性的问题，虽然目前已经有很多优秀的图像清晰度度量算法，但当其针对较大尺寸的图像来说，很难保证实时性。针对这个问题，本论文基于 CUDA 设计和实现了图像清晰度度量算法的并行加速方法。

首先，论文总结了图像清晰度度量相关课题的研究现状和发展趋势，给出了本文的主要工作和内容安排。接着，分析了几种经典的图像清晰度度量算法并且给出了它们的程序实现，使用测试数据对这些度量算法的有效性和实时性进行了对比测试。然后，在介绍 CUDA 编程模型之后，针对选择的图像清晰度度量算法，给出了基于 CUDA 的图像清晰度度量算法并行加速方法的基本流程，并进行了具体程序设计与实现，并对其中的并行归约核函数讨论了几种优化策略。最后，使用不同尺寸的测试图像在不同 GPU 配置的计算机上对加速比进行了测试，并分析讨论了影响图像清晰度度量算法加速比的因素。

论文针对现有图像清晰度度量算法对于大幅面图像的实时性较差的问题，基于 CUDA 设计并实现了图像清晰度度量算法的并行加速方法。实验结果表明，使用 GPU 并行加速后的度量算法对于大尺寸图像的计算时间大大缩短，实现了数十倍的加速比，取得了良好的加速效果。

关键词：图像清晰度度量　CUDA　GPU 并行计算　图像处理

ABSTRACT

In image quality assessment research, using metric algorithms to evaluate the sharpness of digital images to derive a metric value that matches human visual perception is a very interesting and challenging problem, and although there are many excellent image sharpness metric algorithms available, it is difficult to guarantee real-time performance when they are targeted for larger image sizes. To address this problem, this thesis designs and implements a parallel acceleration method for image sharpness metric algorithms based on CUDA.

First, the thesis summarizes the current research status and development trend of image sharpness metric related topics, and gives the main work and content arrangement of this thesis. Then, several classical image sharpness metric algorithms are analyzed and their program implementations are given, and the effectiveness and real-time performance of these metric algorithms are compared and tested using test data. Then, after introducing the CUDA programming model, the basic flow of the CUDA-based parallel acceleration method for image sharpness metric algorithms is presented for the selected image sharpness metric algorithms, and the specific program design and implementation are carried out, and several optimization strategies for the parallel reductive kernel functions are discussed. Finally, the acceleration ratio is tested on computers with different GPU configurations using test images of different sizes, and the factors affecting the acceleration ratio of the image sharpness metric algorithm are analyzed and discussed.

The thesis designs and implements a parallel acceleration method for image sharpness metric algorithm based on CUDA to address the problem of poor real-time performance of existing image sharpness metric algorithms for large size images. The experimental results show that the computation time of the metric algorithm is greatly reduced for large size images after using GPU parallel acceleration, and a good acceleration effect is achieved with a speedup ratio of tens of times.

Keywords: sharpness assessment CUDA GPU parallel-computing image processing

目 录

第一章 绪论

1.1 选题背景和研究意义

1.1.1 选题背景

随着互联网成为时代的主流，图像数据的数量也迅猛地增长，其中不免充斥着大量低质量的冗余图像，随着计算机计算能力的不断提高，使用计算机程序来进行图像质量的评估越来越具有优势；另一方面，由于图像质量评估在很多领域都有广泛的应用性，图像质量评估需求的一直不断增长，因此在图像研究领域中，如何度量一幅图像的质量，一直是研究的热点，每年都会出现大量的评估算法来解决该课题中的难点和不足。

在图像质量评估研究中，使用度量算法对数字图像的清晰度进行评估判断，从而得出一个符合人类视觉感知度量值，是一个非常具有趣味性和挑战性的问题，同时图像清晰度度量作为图像质量评估的一个十分重要的因素，不仅在数字图像质量评估中是至关重要的一环，而且其应用范围也十分的广泛，包括基于图像清晰度度量的自动对焦系统和图像质量增强、恢复和压缩算法等领域。

虽然，目前已经有很多非常出色的图像清晰度度量算法，这些算法在有效性方面基本都比较突出，但针对较大幅面图像时，由于计算复杂度的限制，很难保证实时性，这就导致了这些算法在有些特定的场景下，在确保算法有效性的前提下，很难满足高即时性的要求，与此同时这些清晰度度量算法当中的大多数可以使用并行计算技术来进行加速，但目前这个方向还鲜有人进行相关的研究和实现。

随着图形处理器（Graphics Processing Unit，GPU）及其相关技术的发展，其不再局限于图形处理任务，出现了一系列 GPU 并行计算架构，利用多计算核心的优势，图形处理单元上的通用并行计算现在已经被广泛应用于大数据研究、机器学习和人工智能等诸多领域。这些并行计算架构中最具有代表性的为 NVIDIA 在 2007 年推出的 CUDA（Compute Unified Device Architecture，统一计算架构）。在这些现代工具和技术的支持下，它们提供的可编程接口可以将开发人员从复杂的硬件结构中解放出来，而无需掌握复杂的 GPU 硬件结构、内存管理、指令优化和线程组织等相关知识，这样实现基于 GPU 的高性能并行计算程序变得相对容易。由于 GPU 并行编程模型采用的是 SIMD（Single Instruction Multiple Data，

单指令流多数据流）的并行技术，故 GPU 并行计算主要适用于需要重复执行相同计算的程序，而因为数字图像数据存储的结构，GPU 并行计算非常适合某些图像处理程序的加速，这会带来巨大的性能提升[1]。

1.1.2 研究意义

基于 CUDA 并行计算架构，将传统的图像清晰度度量算法进行加速，充分利用 GPU 多核高性能并行计算的优势，减少经典图像清晰度度量算法需要消耗较长计算时间的缺陷，提高度量算法针对大幅面图像的实时性，这将极大提高图像清晰度算法的时间效率，这样可以满足某些特定场景下对于图像清晰度度量算法的要求。

1.2 国内外研究现状及发展趋势

1.2.1 国内研究现状

在国内，关于图形清晰度度量算法这个问题一直是研究的热门，研究人员所提出的方案重要集中在对经典图像清晰度度量算法的改良以及针对传统方法的缺陷结合了多种度量算法来提升度量算法的可靠性和无偏性。

王勇等[2]提出了一种基于窗口模式的平方梯度函数作为图像清晰度评价函数，该方法采用五区域模式取不同权重，较好平衡了有效性和实时性。蒋婷等[3]改进了边缘检测 Sobel 算子，提出一种新的基于梯度计算的图像清晰度评价函数，该评价函数采用八方向的 Sobel 算子对图像进行评价运算，并且用标准差作为阈值来消除噪声的影响。通过测试实验表明该算法有效，且比经典的基于 Sobel 算子的评价函数灵敏度更高、抗噪能力更强。王昕[4]等利用第二层提升小波分解得到的高频和低频能量值，提出一种基于提升小波变换的清晰度评价函数。测试结果表明，该评价函数相比经典的评价函数具有实时性好、准确性高的优点。李郁峰等[5]在分析常用图像清晰度评价算法的基础上，提出了一种基于灰度差分的快速评价函数。该函数把相邻像素水平和垂直方向的差分乘积作为核函数。该算法相比常用评价算法提升了计算速度，同时具有较高的灵敏度，实时性较好。张亚涛等[6]提出了基于区域对比度的图像清晰度评价方法。在每一个区域中计算区域对比度，然后得到平均值作为图像清晰度评估值。该方法解决了单纯计算灰度差的缺点，提高了算法实用性和可靠性。朱倩等[7]将邻域相关性和梯度结合，提出了

一种新的图像清晰度评价函数。该算法使用 Sobel 算子计算出梯度后使用阈值消除噪声，再进行相关性加权，实验结果表明该算法抗噪能力和和灵敏度较强。曾海飞等[8]提出了一种改进的梯度阈值图像清晰度评价算法。该算法基于 OSTU 方法提出了新的图像边缘分割阈值的选取方法，并采用局部方差分离图像边缘点和非边缘点，最后使用四方向的 Tenengrad 梯度算法计算度量值。经过实验对比，该算法具有高灵敏度和高抗噪性的优势。

1.2.2 国外研究现状

相比于国内对于图像清晰度度量算法相关的研究，国外的研究人员除了对于经典图像清晰度度量算法的改良，提出了一些采用基于概率模型和统计学分析的图像清晰度度量方法，另外，国外研究人员对于度量算法的测试及其分析更加全面。

J. Ni 等[9]提出了一种新的基于梯度的无参考图像清晰度算法。该算法首先获得图像中的感兴趣区域，然后在选定的区域内搜索能够呈现锐度信息的边缘，然后计算出边缘过渡区宽度，然后得到边缘区域的灰度对比度，最后通过这些因素进行概率求和。实验结果表明，这种算法有效且与人类主观判断保持一致。R. Hassen 等[10]提出了一种基于复小波变换域局部相位相干性的无参考图像清晰度评估方法，使用 LIVE 数据库中的模糊数据集进行的测试表明，该方法与主观分数有很好的相关性。该方法创新点在于将图像清晰度的降低看作是局部相位相干性的损失，可以更合理地响应更加广泛的图像失真类型。K. Gu 等[11]提出一种在自回归参数空间的无参考图像清晰度度量方法，该方法认为局部估计的自回归模型系数的相似性越高意味着图像清晰度越低，而且同时考虑到了颜色对图像清晰度度量的影响。该文使用四个图像数据库的模糊数据集进行的测试结果表明，相对于主流的无参考图像清晰度评价指标，所提出的度量方法具有非常优秀的结果。

1.2.3 发展趋势

目前国内外对于图像清晰度度量算法的研究，在算法的有效性方面已经十分成熟，近些年来研究的主要方向主要集中在提升度量算法的灵敏性、实时性和抗噪性。其中，在提高度量算法的实时性方面，大多数的研究聚焦在降低算法本身的计算复杂度上，并且这个方向上提高度量算法的实时性效果不是非常明显，对于应用并行计算技术来对图像清晰度度量算法加速这个方向的研究目前还很少见。

1.3 论文完成的主要工作

本文介绍了图像清晰度度量和使用 CUDA 并行加速的相关基础知识，借助 OpenCV 实现了几种清晰度度量算法。进一步地，本文介绍了使用 CUDA 并行加速图像清晰度度量算法的主要流程和思路，讨论了关于并行归约函数的优化策略。最后通过测试数据，对比了加速前后性能对比，以及对影响加速比的因素进行了讨论分析。本文具体工作内容如下：

介绍了图像清晰度度量涉及到的相关基础知识，介绍了几种经典的图像清晰度度量算法，并借助 OpenCV 库使用 C++编程实现了这些图像清晰度度量算法，使用测试数据对其进行了测试。

首先对图形处理单元上的通用计算作了简单的介绍，重点介绍了 CUDA 并行计算架构的编程模型，最后阐述了本文图像清晰度度量算法加速的主要思路。

详细说明了本文使用 CUDA 并行编程的加速图像清晰度度量的具体实现，其中着重讨论了关于并行归约函数的优化策略。

为了更方便的测试本文中的度量算法，将实现的度量算法集成到了一个简单的图形用户界面，将本文选择加速的图像清晰度度量算法使用各种测试数据在不同图像显卡配置的设备上进行了对比测试，根据结果分析了影响算法加速比的各种因素。

1.4 论文的内容安排

本文章节安排共有六个章节，具体如下：

第一章 绪论：这一章节作为论文开头，介绍了本课题的选题背景和研究意义，然后分别说明了关于清晰度度量的国内外研究现状，接着阐述了当前本研究的发展趋势，在本章节最后简要说明了本文完成的主要工作和论文的内容安排。

第二章 图像清晰度度量：该章节首先介绍了图像清晰度度量相关的基础概念，其次介绍了几种经典的图像清晰度度量算法并分析了它们的原理，最后借助开源计算机视觉程序库 OpenCV 使用 C++编码实现了这些度量算法，并对其进行了测试。

第三章 使用 CUDA 加速图像清晰度度量算法：这一章节开头介绍了 CUDA 编程模型，然后选择了适合使用 CUDA 加速的图像清晰度度量算法，最后给出了 CUDA 加速度量算法的主要思路和流程。

第四章 加速方法的具体实现及其优化：首先给出了图像清晰度度量算法使用 CUDA 加速的具体实现，然后针对其中的并行归约函数进行了分析并且讨论了一些优化策略。

第五章 测试结果及其分析：将本文中的度量算法进行了封装，并提供了简单的图形用户界面，然后借助实现的图形用户界面对算法使用测试样例进行了测试，验证其有效性和加速比，并且使用不同的测试图像在不同 GPU 配置上的计算机上进行对比测试，依据测试结果分析了影响加速方法加速比的因素。

第六章 总结与展望：在最后这一章节，首先对本文的工作做了总结，接着基于本文工作的不足之处进行了讨论，对未来的工作做了展望。

第二章 图像清晰度度量

2.1 图像清晰度概念

图像清晰度指的是图像上的每个细部纹理和其边界的清晰程度，其与图像的模糊度是两个相反的概念。一般认为，图像的清晰度是人类视觉对图像清晰程度的主观判断，没有明确的衡量标准。

2.2 图像清晰度度量概念

在图像的采集、传输和存储等过程中，因为种种原因，不可避免地会导致图像的清晰度降低。比如说在图像的采集阶段，由于相机的对焦问题或者快门速度问题导致采集到的图像模糊程度较大；又比如说在这个数据爆炸的时代，互联网上的图像传输和存储变得十分频繁，很多网站和应用都会采用压缩算法来压缩图像的体积来减轻服务器带宽占用和数据库存储压力，在这个过程中一张图像会被反复压缩很多次，导致图像失真严重、图像清晰度大大降低。当一幅图像的清晰度过低，换而言之也就是图像模糊程度过高时，从图像中识别和获取信息就变得异常困难，很多时候我们总是希望只关注清晰度较高的图像，所以就需要去量化图像的模糊程度，即图像清晰度的度量，从而挑选出清晰度相对更高的图像。图像清晰度度量任务的目的就是针对一组尺寸内容相同且尺寸相同的图像进行打分，反映出其清晰程度差异，打分结果要求与人类视觉感受相一致。

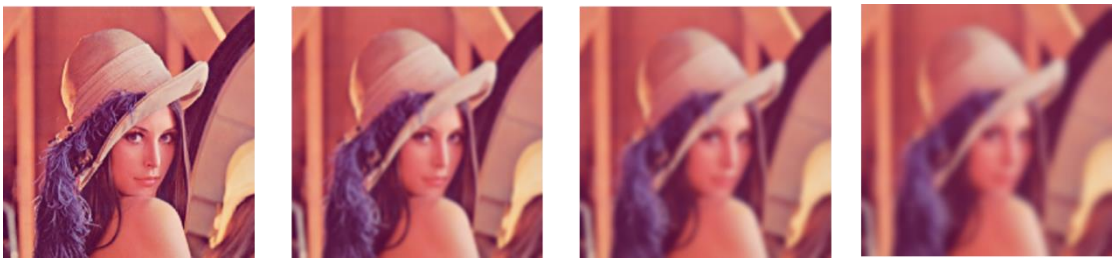

图 2.1 不同清晰度的图像

如图 2.1 所示，这四张图像的内容和尺寸都完全一致，但这四张图像从左到右清晰程度依次降低，一个正确有效的图像清晰度度量算法应该能反映它们的清晰程度，即算法给出的度量值打分也应该依次递减。

2.3 经典的图像清晰度度量算法

2.3.1 方差评价函数

一般来说，清晰度较高的图像，其像素灰度波动程度要比清晰度低的图像更加剧烈，方差描述的是随机变量的离散程度，即随机变量到其期望的距离，故可以用方差函数来评估一幅图像的清晰程度。其可以表示为：

$$F = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}[g(x,y)-\mu]^2 \qquad \text{式(2-1)}$$

其中：

$$\mu = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}g(x,y) \qquad \text{式(2-2)}$$

式(2-2)中，$g(x,y)$ 代表图像矩阵在 (x,y) 处的灰度值，M 为图像高度，N 为宽度，μ 为图像灰度值的平均值，F 为结果，即该图像的清晰度度量值。

2.3.2 Roberts 评价函数

Roberts 评价函数是基于 Roberts 算子的图像清晰度度量函数，Roberts 算子是一种用于边缘检测的最简单的算子，该算子又叫一阶交叉微分算子，将相邻像素正副对角线方向灰度值作差分，从而提取出这两个方向的梯度。Roberts 算子如式(2-3)所示：

$$k_x = \begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix}, k_y = \begin{bmatrix} 0 & -1 \\ +1 & 0 \end{bmatrix} \qquad \text{式(2-3)}$$

清晰度较高的图像具有更加明显的边缘，在边缘上，灰度值的变化是比较明显的，表达变化的方法是求导，在离散域中为差分。因此可以使用边缘检测算子提取出图像的梯度，然后将梯度的模或其平方和进行求和，得到的结果就能反映出图像的清晰程度[12]。

Roberts 评价函数就是使用 Roberts 算子对灰度图像矩阵进行卷积计算提取出正负 45° 方向的梯度，然后将这两个方向的梯度的模值进行累加，最后再除以图像宽度和高度的乘积，如式(2-4)所示：

$$F = \frac{1}{MN}\sum_{x=1}^{M-2}\sum_{y=1}^{N-2}(G_x + G_y) \qquad \text{式(2-4)}$$

其中：

$$G_x = |f(x,y) * k_x|, G_y = |f(x,y) * k_y| \quad \text{式(2-5)}$$

2.3.3 Tenengrad 评价函数

与 Roberts 评价函数类似，Tenengrad 评价函数也是通过提取图像的梯度来度量图像清晰度，但不同的是 Tenengrad 评价函数使用了 Sobel 算子。Sobel 算子同样也是离散的一阶微分算子，分别用两个奇数大小（一般为3）的内核，在水平和垂直方向对灰度图像矩阵进行卷积计算，从而提取出梯度。Sobel 算子如下所示：

$$k_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, k_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \quad \text{式(2-6)}$$

提取出水平和垂直两个方向的梯度之后，Tenengrad 评价函数将他们的平方和进行累加，最后同样除以图像宽度和高度的乘积，结果作为度量值，如式(2-7)所示：

$$F = \frac{1}{MN} \sum_{x=1}^{M-2} \sum_{y=1}^{N-2} [G_x^2(x,y) + G_y^2(x,y)] \quad \text{式(2-7)}$$

2.3.4 拉普拉斯评价函数

前面的 Roberts 算子和 Sobel 算子都是一阶微分算子，而拉普拉斯评价函数采用的是二阶微分算子来提取梯度，在二维空间上，拉普拉斯算子的定义为：

$$\Delta = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \quad \text{式(2-8)}$$

在离散域中，

$$\begin{aligned} \frac{\partial^2 f}{\partial x^2} &= f(x+1,y) + f(x-1,y) - 2f(x,y) \\ \frac{\partial^2 f}{\partial y^2} &= f(x,y+1) + f(x,y-1) - 2f(x,y) \end{aligned} \quad \text{式(2-9)}$$

因为水平和垂直这两个方向的梯度值的符号可能异号，这样它们相加会相互抵消，所以分别对其取绝对值再求和然后在整幅图像上进行累加，即：

$$F = \frac{1}{MN} \sum_{x=1}^{M-2} \sum_{y=1}^{N-2} (\left|\frac{\partial^2 f}{\partial x^2}\right| + \left|\frac{\partial^2 f}{\partial y^2}\right|) \quad \text{式(2-10)}$$

2.3.5 灰度差分评价函数

和基于梯度提取的图像清晰度量函数一样，灰度差分评价函数将相邻像素灰度值作差分然后求和：

$$F=\sum_{x=0}^{M-2}\sum_{y=0}^{N-2}[|f(x,y)-f(x+1,y)|+|f(x,y)-f(x,y+1)|] \quad \text{式(2-11)}$$

该评价函数相对来说计算复杂度较低，有着较好的实时性。

2.3.6 灰度差分乘积评价函数

灰度差分评价函数虽然能够反映图像的清晰程度而且实时性较好，但其灵敏性不够高，导致不同清晰度的图像给出的度量值差异较小，所以针对这个缺点提出了灰度差分乘积评价函数[5]：

$$F=\sum_{x=0}^{M-2}\sum_{y=0}^{N-2}|f(x,y)-f(x+1,y)||f(x,y)-f(x,y+1)| \quad \text{式(2-12)}$$

2.3.7 最大最小灰度梯度评价函数

$$F=\sum_{x}\sum_{y}[\max(f)-\min(f)] \quad \text{式(2-13)}$$

式中 $\max(f)$ 和 $\min(f)$ 分别表示 (i,j) 处的邻域内灰度值的最大值和最小值。

最大最小灰度梯度函数原理与基于边缘检测算子的评价算法类似，将局部灰度变化的剧烈程度进行累加来反映图像的清晰程度。

2.3.8 信息熵评价函数

信息熵是用来度量随机变量的不确定性程度的大小的统计量，也就是说信息熵反映了数据的混乱程度。对清晰程度较高的图像来说，其灰度值分布差异较大，即混乱程度较高；而清晰程度较低图像则一般拥有较为均匀的灰度分布，混乱程度较低。因此，可以使用图像灰度的信息熵来度量图像的清晰程度，其计算方法如式(2-14)所示：

$$F=-\sum_{k=0}^{255}p_k\log_2 p_k \quad \text{式(2-14)}$$

其中 p_k 表示灰度值 k 在整幅图像中的概率。

2.4 经典图像清晰度度量算法的实现

2.4.1 图像灰度化

由于这些度量函数都是在灰度图的基础上进行的，因此首先给出采用的图像灰度化的方法及其实现。

在提取图像清晰度特征的时候，不需要关注图像的色彩信息，所以使用灰度化方法来将图像从 RGB 色彩空间转换到 GRAY 色彩空间[13]。灰度化的目标就是输入一幅3通道的彩色图像，输出为单通道的灰度图，如图 2.2 所示。

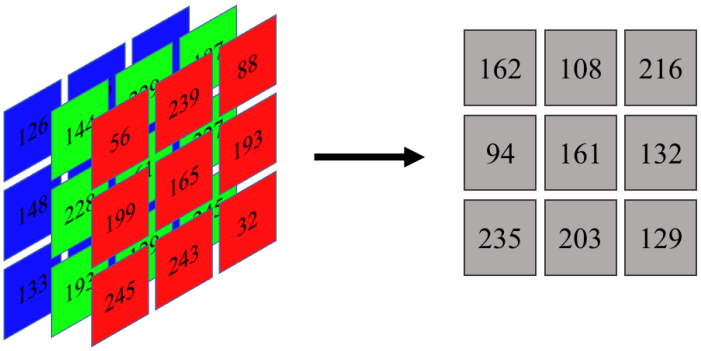

图 2.2 图像灰度化

常用的图像灰度化方法一般有两种：

平均值法：取三个色彩分量的平均值以作为灰度；

加权平均法：该方法考虑到人眼对与不同色彩分量的敏感程度，取不同的权值对其加权平均。

本文采用了加权平均法来进行图像灰度化，具体如式(2-15)：

$$Grayscale = 0.114B + 0.587G + 0.299R \qquad \text{式(2-15)}$$

具体算法实现如下：

Algorithm 1 灰度化函数

Initializations:

rows ← src.rows() 图像高度

cols ← src.cols() 图像宽度

Iteration:

```
1: for i ← 0 to rows-1 do
2:  for j ← 0 to cols-1 do
3:   dst[i][j] ← 0.114f * src[i][j][0] + 0.587f * src[i][j][1] + 0.299f * src[i][j][2]
4:  end for
```

5: **end for**

下面给出前面介绍的几种经典图像清晰度度量算法的具体实现。

2.4.2 方差评价函数的实现

Algorithm 2 方差函数

```
1: rows←src.rows() 图像高度
2: cols←src.cols() 图像宽度
3: mean←0.0 灰度平均值初始化
4: res←0.0 结果初始化
5: for i←0 to rows-1 do
6:  for j←0 to cols-1 do
7:   mean←mean + grayImage[i][j]
8:  end for
9: end for
10: for i←0 to rows-1 do
11:  for j←0 to cols-1 do
12:   temp←grayImage[i][j]-mean
13:   res←res + temp * temp
14:  end for
15: end for
16: res←res / (cols * rows)
```

2.4.3 Roberts 评价函数的实现

Algorithm 3 Roberts 评价函数

```
1: rows←src.rows() 图像高度
2: cols←src.cols() 图像宽度
3: res←0.0 结果初始化
4: for i←0 to rows-2 do
5:  for j←0 to cols-2 do
6:   rx←grayImage[i+1][j+1] – grayImage[i][j]
7:   ry←grayImage[i+1][j] – grayImage[i][j+1]
```

8: res←res + abs(rx) + abs(ry)

9: **end for**

10: **end for**

11: res←res / (cols * rows)

2.4.4 Tenengrad 评价函数的实现

Algorithm 4 Tenengrad 评价函数

1: rows←src.rows() 图像高度

2: cols←src.cols() 图像宽度

3: res←0.0 结果初始化

4: **for** i←1 to rows-2 **do**

5: **for** j←1 to cols-2 **do**

6: dx←-grayImage[i-1][j-1] – 2 * grayImage[i][j-1] - grayImage[i+1][j-1] + grayImage[i-1][j+1] + 2 * grayImage[i][j+1] + grayImage[i+1][j+1]

7: dy←-grayImage[i-1][j-1] - 2 * grayImage[i-1][j] - grayImage[i-1][j+1] + grayImage[i+1][j-1] + 2 * grayImage[i+1][y] + grayImage[i+1][j+1]

8: res←res + dx * dx+ dy * dy

9: **end for**

10: **end for**

11: res←res / (cols * rows)

2.4.5 拉普拉斯评价函数的实现

Algorithm 5 拉普拉斯评价函数

1: rows←src.rows() 图像高度

2: cols←src.cols() 图像宽度

3: res←0.0 结果初始化

4: **for** i←1 to rows-2 **do**

5: **for** j←1 to cols-2 **do**

6: dx←grayImage[i][j+1] + grayImage[i][j-1] – 2 * grayImage[i][j]

7: dy←grayImage[i+1][j] + grayImage[i-1][j] – 2 * grayImage[i][j]

8: res←res + abs(dx) + abs(dy)

9: **end for**

10: **end for**

11: res←res / (cols * rows)

2.4.6 灰度差分评价函数的实现

Algorithm 6 灰度差分评价函数

1: rows←src.rows() 图像高度

2: cols←src.cols() 图像宽度

3: res←0.0 结果初始化

4: **for** i←0 to rows-2 **do**

5: **for** j←0 to cols-2 **do**

6: dx←grayImage[i][j] - grayImage[i][j+1]

7: dy←grayImage[i][j] - grayImage[i+1][j]

8: res←res + abs(dx) + abs(dy)

9: **end for**

10: **end for**

11: res←res / (cols * rows)

2.4.7 灰度差分乘积评价函数的实现

Algorithm 7 灰度差分乘积评价函数

1: rows←src.rows() 图像高度

2: cols←src.cols() 图像宽度

3: res←0.0 结果初始化

4: **for** i←0 to rows-2 **do**

5: **for** j←0 to cols-2 **do**

6: dx←grayImage[i][j] - grayImage[i][j+1]

7: dy←grayImage[i][j] - grayImage[i+1][j]

8: res←res + abs(dx * dy)

9: **end for**

10: **end for**

11: res←res / (cols * rows)

2.4.8 最大最小灰度梯度评价函数的实现

Algorithm 8 最大最小灰度梯度评价函数

```
1: rows←src.rows() 图像高度
2: cols←src.cols() 图像宽度
3: res←0.0 结果初始化
4: for x←1 to cols-2 do
5:  for y←1 to rows-2 do
6:   max_val←0
7:   min_val←255
8:   for x←-1 to 1 do
9:    for y←-1 to 1 do
10:    max_val←max{max_val, grayImage[y+i][x+j]}
11:    min_val←min{min_val, grayImage[y+i][x+j]}
12:   end for
13:    end for
14:   end for
15:  res←res + max_val - min_val
16:  end for
17: end for
18: res←res / (cols * rows)
```

2.4.9 信息熵评价函数的实现

Algorithm 9 信息熵评价函数

```
1: rows←src.rows() 图像高度
2: cols←src.cols() 图像宽度
3: res←0.0 结果初始化
4: p[256]←{0.0} 概率初始化
5: for x←0 to cols-1 do
6:  for y←0 to rows-1 do
7:   val←grayImage[y][x]
8:   p[val]←p[val] + 1
```

```
9:  end for
10: end for
11: for u←0 to 255 do
12:  p[i]←p[i] / (rows * cols)
13: end for
14: for i←0 to 255 do
15:  if p[i] != 0
16:   res←p[val] / (rows * cols)
17:  end if
18: end for
19: res←res / (cols * rows)
```

2.5 经典图像清晰度度量算法的测试

2.5.1 有效性测试

使用了二十张清晰度不同的图像，这二十张图像从编号 1~20 其清晰度先递增后递减，对前面介绍的八种经典的图像清晰度度量算法进行测试，将其度量值进行最大最小归一化之后（如式 2-16），得到如图 2.3 所示的结果。

$$x' = \frac{x - X_{min}}{X_{max} - X_{min}} \qquad \text{式(2-16)}$$

其中 x 是原始数据值，X_{max}、X_{min} 分别是这组数据中的最大值和最小值，x' 是归一化后的结果。

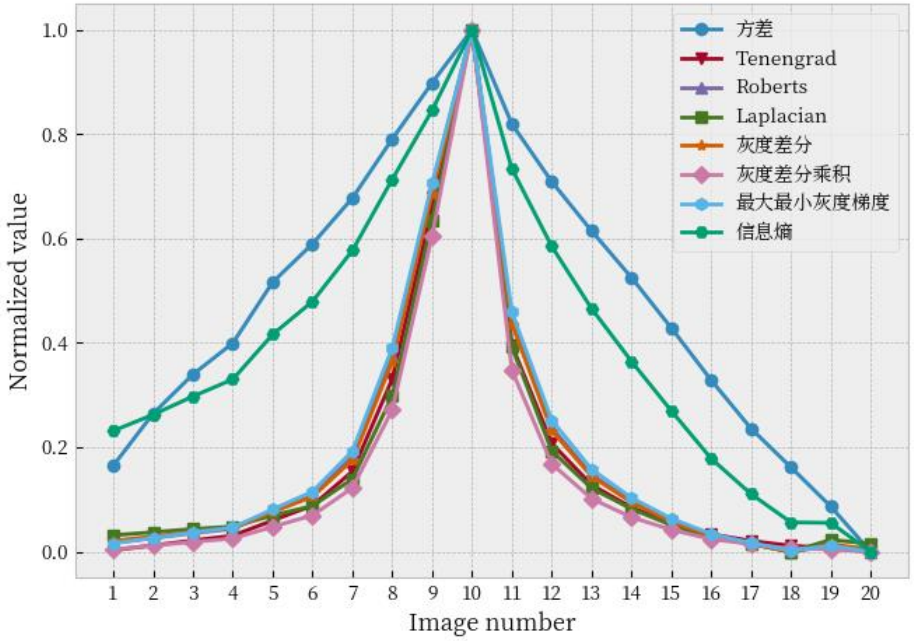

图 2.3 经典图像清晰度度量算法有效性测试结果

如图 2.3 所示，折线图横坐标分别是二十幅图像的编号，纵坐标为经过最大最小归一化之后各种度量算法的度量值。从折线图中，可以看出这八种经典的图像清晰度度量算法基本都正确反映出了图像的清晰程度。其中方差评价函数和信息熵评价函数与其他评价函数曲线走势差异较大，其他评价函数单峰性较好，经过分析这是由于除了这两种度量算法之外，其他六种度量算法都是基于灰度差分的原理。

2.5.2 实时性测试

使用六幅不同大小的图像，其尺寸分别为 256×256 、 512×512 、 1024×1024 、 2048×2048 、 4096×4096 和 8192×8192 ，对上述八种经典图像清晰度度量算法进行了测试，得到结果如图 2.4 所示。

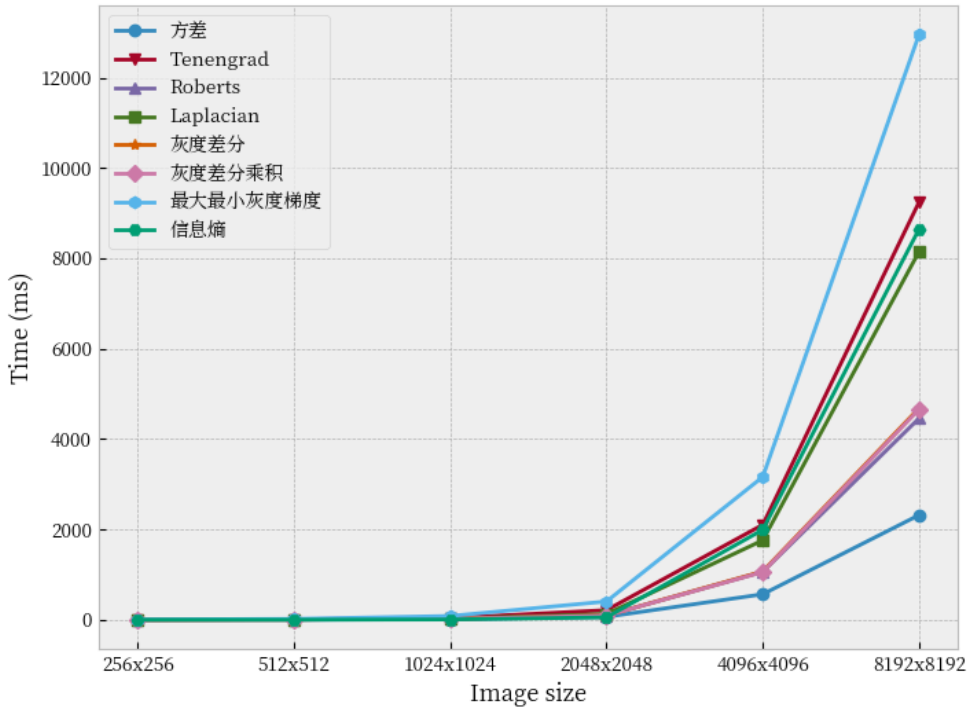

图 2.4 经典图像清晰度度量算法实时性测试结果

如图 2.4 所示，折线图横坐标为图像尺寸大小，纵坐标为算法运行耗时（单位为毫秒），从中可以看出，由于这八种图像清晰度度量算法的时间复杂度都是在 $O(mn)$（其中 m 、 n 分别为图像的高度和宽度），因此随着图像尺寸的增大，其运行耗时接近二次函数趋势上升。其中最大最小灰度梯度评价函数因为其复杂度常数较大，所以耗时较多；方差评价函数运算操作较为简单，故其增长较为平缓。

总的来说，这写经典图像清晰度度量算法针对较大幅面的图像时，其运行耗时是相对比较大的，即算法的实时性较差，这样就使得这些经典的图像清晰度度量算法在某些需要高实时性的场景下无法满足要求。

第三章 使用 CUDA 加速图像清晰度度量算法

3.1 CUDA 介绍

CPU（Central Processing Unit，中央处理器）虽然很强大，但针对需要大规模并行计算的场景，CPU 就不再适合，一方面因为 CPU 上的 ALU（Arithmetic Logic Unit，算术逻辑单元）数量相对较少，无法实现真正的高并行计算；另一方面由于 CPU 中的线程比较重，操作系统对其进行调度是一个成本很昂贵的过程。与此相比，GPU（Graphics Processing Unit，图形处理器）拥有更多轻量级的 ALU 且其设计目的也不同，比较适合对大量数据进行并行操作。在 GPU 发展之初，GPU 只是被用来加速图形渲染的性能，如今，有了 GPU 并行计算架构的帮助，使用 GPU 加速科学和工程计算变得更加容易。

CUDA（Compute Unified Device Architecture，统一计算架构）是由图形处理器半导体公司 NVIDIA 提出的一种基于 NVIDIA GPU 设备的通用并行计算架构。在某些语境下，CUDA 也是基于该架构的一种异构编程语言，所谓异构，就是指一套代码可以运行在主机（Host）与设备（Device）两种平台（设备在这里指 GPU），其是 C++语言的扩展，NVIDIA 基于 LLVM 开发了针对其专用的编译器 NVCC（Nvidia CUDA Compiler），NVCC 首先将主机代码和设备代码分开，之后将主机代码使用对应平台编译器进行编译，并且将设备代码进行进一步地编译，最后使用链接器将它们连同 CUDA RUNTIME 库链接到一起生成可执行文件。CUDA 提供的 API 不要求开发者对 GPU 有深入的了解就可以利用 GPU 实现高性能的并行计算程序。

3.2 CUDA 编程模型

3.2.1 核函数

在 CUDA 中，调用核函数时其将会在 GPU 上被若干个 CUDA 线程并行执行，即 SIMD 模型（Single Instruction Multiple Data，单指令流多数据流），这区别与普通的函数调用只会在 CPU 上执行一次。声明一个核函数需要在函数声明处返回值前面加上__global__关键字（具体如表 3.1），而且核函数的返回值必须为 void 类型。

表 3.1 CUDA 函数类型

函数类型	调用和执行	声明关键字
核函数	主机上调用，设备上执行	__global__
设备函数	设备上调用，设备上执行	__device__
主机函数	主机上调用，主机上执行	__host__（可以省略）

3.2.2 CUDA 线程组织

CUDA 使用网格（Grid）和线程块（Block）组织线程，在启动内核函数时在“<<<…>>>”中传入网格大小和线程块大小，类型可以为整型或者三分量向量(dim3)类型。在每个内核函数中，都有内置变量 blockIdx 、blockDim 和 threadIdx，可以通过来获取当前线程索引。例如，网格大小为(4,3,1)、线程块大小为(5,4,3)的线程组织结构如图 3.1 所示。

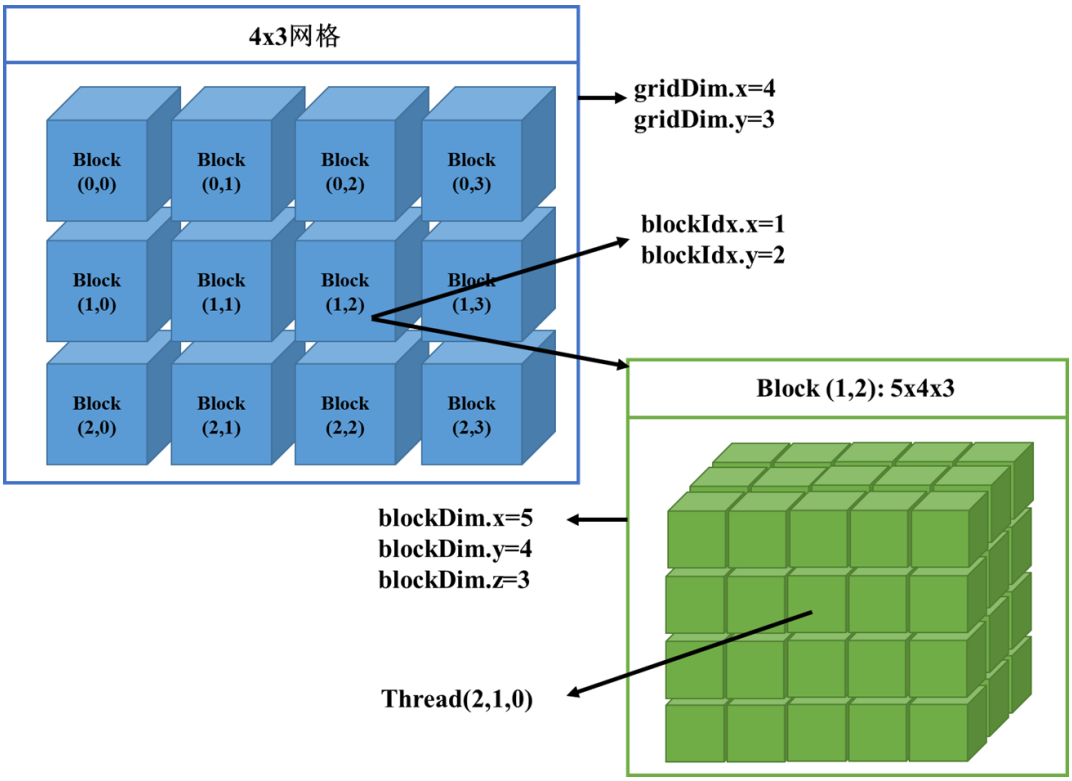

图 3.1 CUDA 线程组织结构

如图 3.1，网格是二维的，其 x 和 y 方向的维度分别为4和3，即 gridDim.x 和 gridDim.y 分别为4和3；共有12个线程块，每个线程块是三维的，x、y 和 z 三个方向的维度分别为5、4、3，那么每个线程块就有5×4×3＝60个线程。

3.2.3 CUDA 内存体系

和 CPU 内存一样，GPU 中的内存也是分级设计的，从而减小内存访问延迟、提升核函数运行效率。CUDA 中的内存分了六个级别，它们分别是全局内存、常量内存、纹理内存、寄存器内存、局部内存和共享内存，它们的组织关系如图 3.2 所示，图中箭头表示可读写方向[14]。

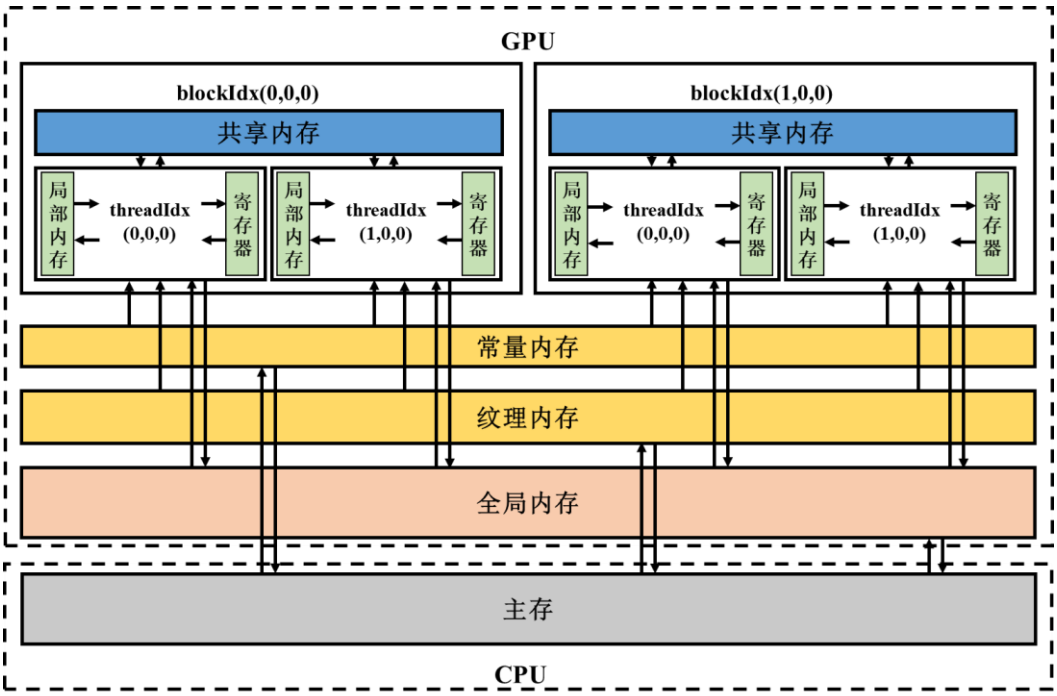

图 3.2 CUDA 内存体系

全局内存（Global Memory），顾名思义，其可以被核函数中所有的线程访问但其是 CUDA 内存体系中访问速度最慢的。全局内存分为静态全局内存和动态分配的全局内存，后者主要用作在主机和设备之间传输数据：在核函数调用之前，首先使用 cudaMalloc 在设备上动态开辟全局内存，然后调用 cudaMemcpy(d_m, h_m, cudaMemcpyHostToDevice)将主机上的数据拷贝至动态分配的设备全局内存上；待核函数运行完毕之后同样调用 cudaMemcpy(h_m, d_m, cudaMemcpyDeviceToHost)将运算结果从设备内存拷贝回主机内存，最后使用 cudaFree 函数释放掉设备上开辟的全局内存。

常量内存（Constant Memory）是只读不可写的，使用__constant__关键字定义，同样可以在所有的线程里访问，但其访问速度比去全局内存略高。

纹理内存（Texture Memory）和常量内存类似，一般也是只读不可写的，而且其读写性能也与常量内存相当。其设计目的主要是为了提高 GPU 对 3D 图形渲

染的性能。

寄存器内存（Register Memory）是针对每个线程内的内存，其生命周期当前线程执行的周期。和 CPU 上的寄存器一样，寄存器内存访问速度在 CUDA 中是最快的，但寄存器的数量也是有限的。

局部内存（Local Memory）作为寄存器内存的补充，也是在当前线程内可见，但由于其硬件实现层面是全局内存的一部分，因此其访问性能也比较低。

共享内存（Shared Memory）是线程块内共享的内存，其有着仅次于寄存器内存的访问速度。在某些场景下，利用其线程块内共享的特性，可以用来避免对全局内存的频繁访问。

3.3 使用 CUDA 加速图像清晰度度量算法

3.3.1 度量算法选择

根据本文第二章对经典图像清晰度度量算法的测试以及 CUDA 加速程序的特点，选择了比较有代表性且较为适合用 CUDA 加速的 Tenengrad 评价函数、拉普拉斯评价函数和灰度差分乘积评价函数来使用 CUDA 对其进行加速。

3.3.2 度量算法加速基本流程

选择的这三种度量算法其流程主要是：首先对输入图像进行灰度化，然后针对每一个像素求其度量值，最后将每个像素求得的度量值累加求和，得到最终度量结果。对应的，使用 CUDA 进行加速也分为这三个流程，其流程图如图 3.3 所示。

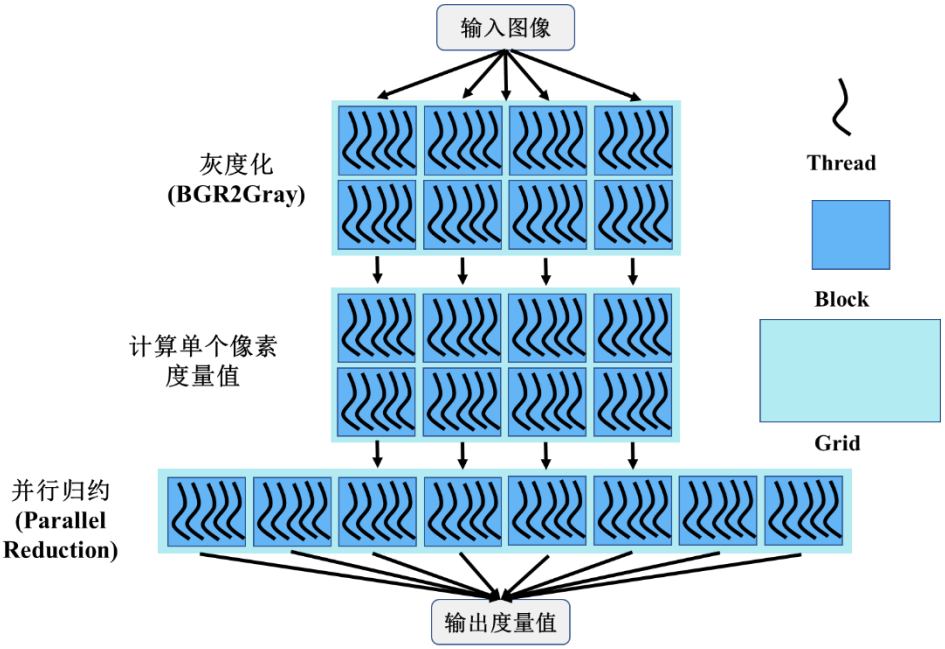

图 3.3 图像清晰度度量算法加速基本流程

对应图 3.3 这三个流程，设计了灰度化核函数、计算单个像素度量值核函数（Tenengrad 评价函数、拉普拉斯评价函数和灰度差分乘积评价函数三种）和并行归约核函数。

首先借助 OpenCV 输入图像，得到图像矩阵，因为该矩阵是二维结构，所以在组织线程结构时，同样将网格和线程块设为二维结构，方便后续核函数的实现。然后将图像矩阵拷贝到 CUDA 全局内存，并将其作为参数传入灰度化核函数，之后得到灰度化之后的图像矩阵。再将得到的灰度图像矩阵传入计算单个像素度量值核函数得到单个像素度量之矩阵。最后再将该矩阵传入并行归约核函数对其累加求和得到结果，这里注意并行归约核函数的网格和线程块组织要变成一维，方便实现归约算法。

第四章 加速方法的具体实现及其优化

4.1 灰度化核函数具体实现

灰度化函数使用 CUDA 加速较为简单，每个 CUDA 线程对应一个像素使用加权平均法去执行灰度化操作。网格和线程块组织为二维结构，其中线程块大小为$(16, 16, 1)$，即16×16，这样可计算出网格大小应为$(\left\lceil \frac{cols}{16} \right\rceil, \left\lceil \frac{rows}{16} \right\rceil, 1)$，即图像宽度、高度除以16上取整。

具体实现如下：

Kernel 1 灰度化核函数

```
template <typename T_in, typename T_out>
__global__ void BGR2Gray(const cv::cuda::PtrStep<T_in> src,
                         cv::cuda::PtrStep<T_out> dst, size_t cols,
                         size_t rows) {
    auto x = blockIdx.x * blockDim.x + threadIdx.x;
    auto y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < cols && y < rows) {
        auto cell = src(y, x);
        dst(y, x) = (T_out)0.114f * cell.x + 0.587f * cell.y + 0.299f * cell.z;
    }
}
```

该核函数输入参数为一个指向 CUDA 全局内存中的彩色三通道图像矩阵的指针 src，输出参数为指向 CUDA 全局内存中的单通道图像矩阵的指针 dst。

内置变量 blockIdx 是当前线程所在线程块的索引，blockIdx.x 是 x 方向上的索引，blockIdx.y 是 y 方向上的索引；blockDim 是线程块的维度大小，blockDim.x、blockDim.y 分别是是 x 方向和 y 方向线程块的维度，在这里它们都是16。所以使用 blockIdx.x * blockDim.x + threadIdx.x 和 blockIdx.y * blockDim.y + threadIdx.y 就可以计算出当前线程应该对应的像素坐标位置。

由于使用 OpenCV 中的 imread 函数得到的彩色图像其通道顺序为 BGR，对应每个像素 xyz 分量，故灰度值为$0.114x+0.587y+0.299z$。

4.2 计算单个像素度量值核函数具体实现

计算单个像素度量值核函数的线程组织和灰度化核函数一致，也是一个 CUDA 线程对应一个像素分别使用这三种度量算法计算度量值。它们的输入参数为灰度化核函数的输出结果，即指向 CUDA 全局内存存储灰度值矩阵的指针；输出参数为单个像素度量值矩阵。

4.2.1 Tenengrad 评价核函数实现

Kernel 2 Tenengrad 评价核函数

```
template <typename T_in, typename T_out>
__global__ void tenengradKernel(const cuda::PtrStep<T_in> src,
                                cuda::PtrStep<T_out> dst, size_t cols,
                                size_t rows) {
  auto x = threadIdx.x + blockDim.x * blockIdx.x;
  auto y = threadIdx.y + blockDim.y * blockIdx.y;
  T_out dx, dy;
  if (x > 0 && x < cols - 1 && y > 0 && y < rows - 1) {
    dx = (-1 * src(y - 1, x - 1)) + (-2 * src(y, x - 1)) +
      (-1 * src(y + 1, x - 1)) + (1 * src(y - 1, x + 1)) +
      (2 * src(y, x + 1)) + (1 * src(y + 1, x + 1));
    dy = (-1 * src(y - 1, x - 1)) + (-2 * src(y - 1, x)) +
      (-1 * src(y - 1, x + 1)) + (1 * src(y + 1, x - 1)) +
      (2 * src(y + 1, x)) + (1 * src(y + 1, x + 1));
    dst(y, x) = ((dx * dx) + (dy * dy));
  }
}
```

如 Kernel 2 所示，由于 src 所指向的矩阵存储在 CUDA 全局内存中，所以每个像素所对应的线程可以直接访问其需要用到的像素信息，在此处每个线程访问其对应像素位置周围的八个像素，计算出度量值，存储在输出参数 dst 所指向的矩阵。

4.2.2 拉普拉斯评价核函数实现

Kernel 3 拉普拉斯评价核函数

```
template <typename T_in, typename T_out>
__global__ void LaplacianKernel(const cuda::PtrStep<T_in> src,
                                cuda::PtrStep<T_out> dst, size_t cols,
                                size_t rows) {
  auto x = threadIdx.x + blockDim.x * blockIdx.x;
  auto y = threadIdx.y + blockDim.y * blockIdx.y;
  T_out lx, ly;
  if (x > 0 && x < cols - 1 && y > 0 && y < rows - 1) {
    lx = CudaMath::abs<T_out>(src(y, x + 1) + src(y, x - 1) - 2 * src(y, x));
    ly = CudaMath::abs<T_out>(src(y + 1, x) + src(y - 1, x) - 2 * src(y, x));
    dst(y, x) = lx + ly;
  }
}
```

4.2.3 灰度差分乘积评价核函数实现

Kernel 4 灰度差分乘积核函数

```
template <typename T_in, typename T_out>
__global__ void SDMKernel(const cuda::PtrStep<T_in> src,
                          cuda::PtrStep<T_out> dst, size_t cols, size_t rows) {
  auto x = threadIdx.x + blockDim.x * blockIdx.x;
  auto y = threadIdx.y + blockDim.y * blockIdx.y;
T_out dx, dy;
  if (x < cols - 1 && y < rows - 1) {
    dx = src(y, x) - src(y, x + 1);
    dy = src(y, x) - src(y + 1, x);
    dst(y, x) = CudaMath::abs<T_out>(dx * dy);
  }
}
```

4.3 并行归约核函数实现

4.3.1 折半归约法

并行计算一个数组的和，为了解决这个问题，一般使用折半归约法（Binary Reduction）。类似于归并排序的 Bottom-Up 实现，使用迭代的方法，首先将每一对相邻元素相加，每一对元素相加是独立的，因此可以使用并行计算，然后在其基础上并行求得所有相邻四个元素的和，以此类推，最后得到所有元素之和。例如，对于一个长度为16的数组，使用折半归约法并行求和，其算法流程如图 4.1。

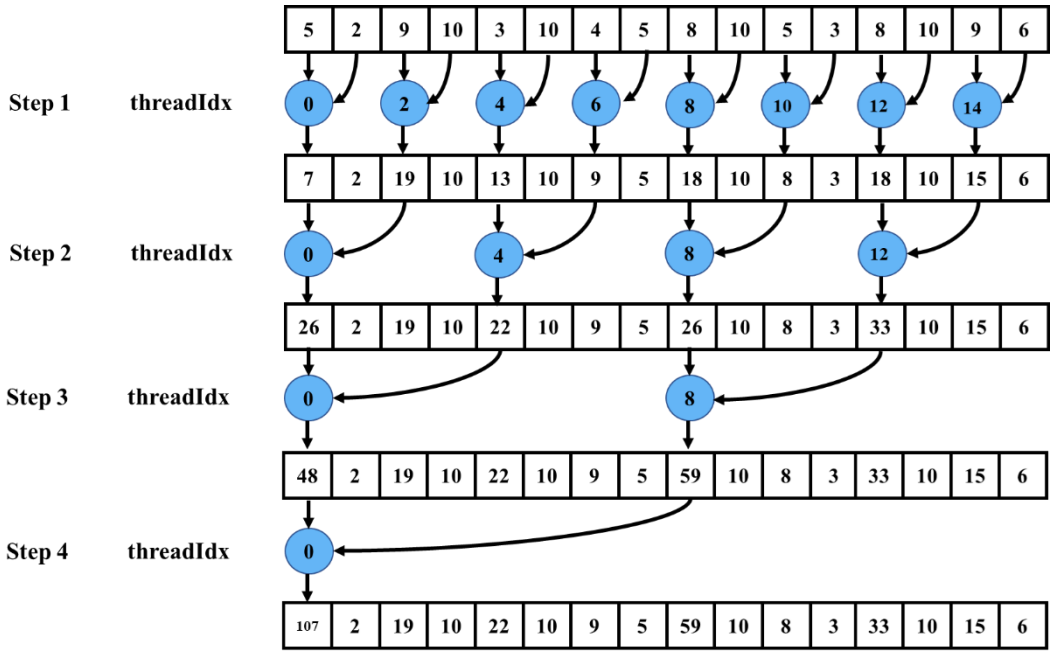

图 4.1 折半归约法

4.3.2 并行归约核函数实现

由于折半归约法要求线程组织是一维的，所以将前两个流程的网格和线程块组织一维化，即线程块大小为256，由此计算出网格大小为图像大小除以线程块大小向上取整$\left\lceil \frac{cols \times rows}{256} \right\rceil$。

从图 4.1 可以看出，如果不考虑同步问题，并行归约就会产生数据竞争问题，例如：假设在第二步归约时，2 号线程将9+10的结果19还没有写入内存的话，对于 0 号线程来说，其看到的 2 号位置的元素还是9，这样就产生了错误。因此要实现并行的折半归约算法，就要考虑同步。然而 CUDA 没有提供全局同步，这是因为要实现全局的同步硬件成本是十分昂贵的。但是 CUDA 支持线程块内所有

线程的同步，所以利用这一点，先在每个线程块内，执行折半归约法求出线程块内所有元素的和，然后使用原子操作，将所有线程块求得的结果再累加起来，这样就得到了正确的结果。

并行归约算法需要对内存进行频繁的读写操作，因此如果直接再全局内存上进行操作的话，会产生较大延迟。对此，可以利用共享内存，以此来提高核函数的效率。

综上所述，初步实现了并行归约核函数，如 Kernel 5 所示。

Kernel 5 并行归约核函数 1

```
1: template <typename T_in, typename T_out>
2: __global__ void reduce(const cv::cuda::PtrStep<T_in> src, T_out *dst,
3:                        size_t cols) {
4:   __shared__ T_out s_data[256];
5:   auto tid = threadIdx.x;
6:   auto i = threadIdx.x + blockDim.x * blockIdx.x;
7:   auto y = i / cols;
8:   auto x = i % cols;
9:   s_data[tid] = src(y, x);
10:  __syncthreads();
11:  for (size_t s = 1; s < blockDim.x; s *= 2) {
12:      if (tid % (2 * s) == 0) {
13:          s_data[tid] += s_data[tid + s];
14:      }
15:      __syncthreads();
16:  }
17:  if (tid == 0) {
18:      atomicAdd(dst, s_data[0]);
19:  }
20: }
```

如 Kernel 5 所示，输入参数 src 为指向单个像素度量值矩阵的指针（第二个流程的结果），而输出参数为指向最终结果的指针。

__shared__关键字用来声明共享内存中的数组 s_data，其大小和线程块大小一致，即 256 。这里要注意虽然每个线程都会执行该核函数，但对于共享内存只会

在各个线程块内定义一次，并且对于一个线程块来说，其内部的所有线程都可以访问到该线程块对应的共享内存。

因为线程块是一维的，threadIdx.x 就是当前线程在其所在线程块的索引，threadIdx.x + blockDim.x * blockIdx.x 为当前线程在所有线程中的索引，将其除以图像宽度可以计算出该线程对应的图像像素坐标。

将当前线程所对应坐标的值赋值到对应的共享内存的位置后，需要调用__syncthread()来确保同一线程块内其他线程也执行到当前位置，来防止后续操作产生数据竞争。

在线程块内进行归约，归约的步长从 2 开始，每次倍增，直到线程块维度的一半。同样的，在每次赋值之后，下一步归约之前，也要使用__syncthread()来进行同步。

当线程块内归约完成之后，即线程块内所有元素的和被存储在该线程块的共享内存的 0 号位置后，在该线程块的 0 号线程使用原子操作加法 atomicAdd 将线程块内的和累加到输出结果上。

4.3.3 优化 1——解决线程束分支发散问题

在硬件实现上，CUDA 的线程块是由线程束（Warp）组成的，线程束是 GPU 多处理器（Multiprocessor）执行的基本单元，目前 CUDA 版本规定了线程束的大小为 32 。例如线程块大小为 256 时，其被分为 8 个线程束，这些线程数可能在若干个多处理器上运行。在同一线程束内的所有线程在同一时刻针对不同的数据执行同样的指令，即单指令多线程（Single Instruction Multiple Thread）的模式。这就意味着，如果对于同一线程束中的不同线程，其对应的数据不同，导致了它们进入了不同的条件分支中，而它们又不能同时执行不一样的指令，这就出现了线程束发散。例如：有一 if-else 分支，对于满足条件 condition 执行 A 指令，否则执行 B 指令。那么对于同一线程束中的线程，其中对应数据满足 condition 条件的并行执行 A 指令，而其他不满足的线程则需要等待，同样的，待其执行完之后，不满足 condition 条件并行执行 B 指令，满足的线程则需要等待，这样就导致了线程执行时间的浪费。因此，在编写 CUDA 核函数时，需要尽量避免出现线程束发散，尽可能地使同一线程束中的线程进入相同的分支，这样可以有效提高核函数的执行效率。

在前文实现的并行归约核函数 Kernel 5 中的第 11~16 行，for 循环中的 if 分支导致了线程束分支发散，对此可以对其作等价变换来减轻该问题所带来的性能损失，具体如 Kernel 6 所示。

Kernel 6 并行归约核函数 2

```
…
11:   for (size_t s = 1; s < blockDim.x; s *= 2) {
12:       auto index = 2 * s * tid;
13:       if (index < blockDim.x) {
14:           s_data[index] += s_data[index + s];
15:       }
16:       __syncthreads();
17:   }
…
```

在 Kernel 6 中，虽然说 if 分支依然存在，但执行该分支的线程是连续的。比如说，对于大小为64 的线程块（两个线程束的大小），在 Kernel 中第一次归约时，即 s = 1 时，线程块内前32 个线程都会进入该 if 分支，后32 个线程不会进入，由于这两部分线程各自在不同的线程束中，所以也就不会出现线程束分支发散问题；相反，在 Kernel 5 中，同一线程束中的线程总是一半进入 if 分支，另一半不进入 if 分支，导致了线程束发散。

另一方面，相比于 Kernel 5 的分支条件判断中的取模运算具有较大的延迟，Kernel 6 的实现避免了取模运算，这样也提升了核函数的效率。

4.3.4 优化 2——避免共享内存 bank 冲突

为了使得内存访问速度更快，在并行归约核函数中使用了共享内存，在现在的 CUDA 版本中，共享内存被分为32 个 bank，每个 bank 大小为 4 个字节。一般情况下，在同一时间内同一个 bank 只允许一个线程束中的一个线程进行访问，因此同一个线程束中不同线程对同一个 bank 进行访问被叫做 bank 冲突，在编写 CUDA 核函数时，需要尽可能地避免 bank 冲突来提高效率。

Kernel 5 和 Kernel 6 中的归约操作实现如图 4.1 所示，可以看出在前两步归约时由于其步长小于4，而且这些线程都在同一线程束内，所以存在 bank 冲突，更重要的是，前两步归约正好是对共享内存数据访问最频繁的，这意味着这些访问

操作将不能并行的执行，大大降低了核函数效率。为此，可以把每次归约的步长逆序，如图 4.2 所示。

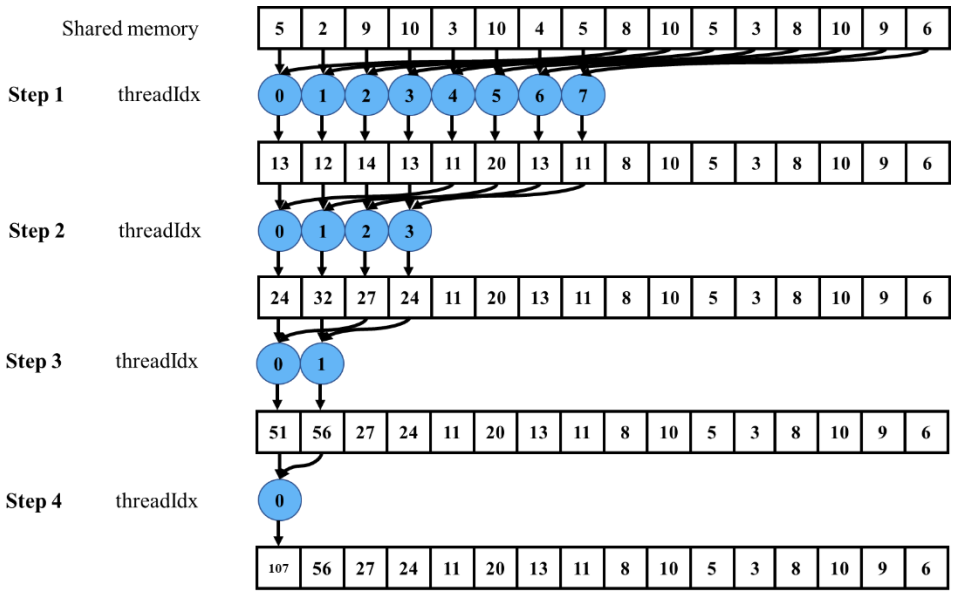

图 4.2 消除了 bank 冲突的并行归约算法

如图 4.2，在归约的前两步，也就是访问共享内存最频繁的过程，其中并没有 bank 冲突的发生，这意味着每个线程对共享内存的访问操作可以并行地执行。核函数地具体如 Kernel 7 所示。

Kernel 7 并行归约核函数 3

```
…
11:   for (auto s = blockDim.x >> 1; s > 0; s >>= 1) {
12:       if (tid < s) {
13:           s_data[tid] += s_data[tid + s];
14:       }
15:       __syncthreads();
16:   }
…
```

4.3.5 优化 3——避免线程浪费

对于 Kernel 7 的实现，可以看到归约的步长从线程块大小的一半开始，因此其实线程块中后半部分的线程没有进入 if 分支，也就是说这部分线程没有进行任何操作，这造成了线程的浪费。

事实上，不需要每个像素对应一个线程，因此可以将线程块的数量减少一半，换句话说，就是线程块的大小不变，网格的大小变为原来的一半，即

$\left\lceil \frac{cols \times rows}{256 \times 2} \right\rceil$，并且在核函数内给共享内存赋值的时候，就完成第一次归约操作，如 Kernel 8 所示。

Kernel 8 并行归约核函数 4

```
…
9:  s_data[tid] = src(y, x) + src(ny, nx);
…
```

4.3.6 优化 4——展开最后一个线程束

本文在 4.3.3 介绍了 CUDA 中的线程束，并且说明了在同一时刻同一线程束中的线程执行一样的指令，因此也无需考虑同步问题。而在之前的核函数实现中，对于步长小于等于32的情况，同样调用了__syncthread()来等待线程块内所有线程同步，这是没有必要的。因此，可以将步长小于等于32的情况在循环外展开且不调用同步函数，如 Kernel 9 所示。

Kernel 9 并行归约核函数 5

```
…
   for (size_t s = blockDim.x >> 1; s > 32; s >>= 1) {
      if (tid < s) {
         s_data[tid] += s_data[tid + s];
      }
      __syncthreads();
   }
   if (tid < 32) {
      s_data[tid] += s_data[tid + 32];
      s_data[tid] += s_data[tid + 16];
      s_data[tid] += s_data[tid + 8];
      s_data[tid] += s_data[tid + 4];
      s_data[tid] += s_data[tid + 2];
      s_data[tid] += s_data[tid + 1];
   }
…
```

第五章 测试结果及其分析

为了方便测试，本文将实现的图像清晰度度量算法进行了封装，基于图形用户界面 C++库 Elements 设计了简单的图形界面来进行度量算法加速前后的对比测试，程序界面如图 5.1 所示。

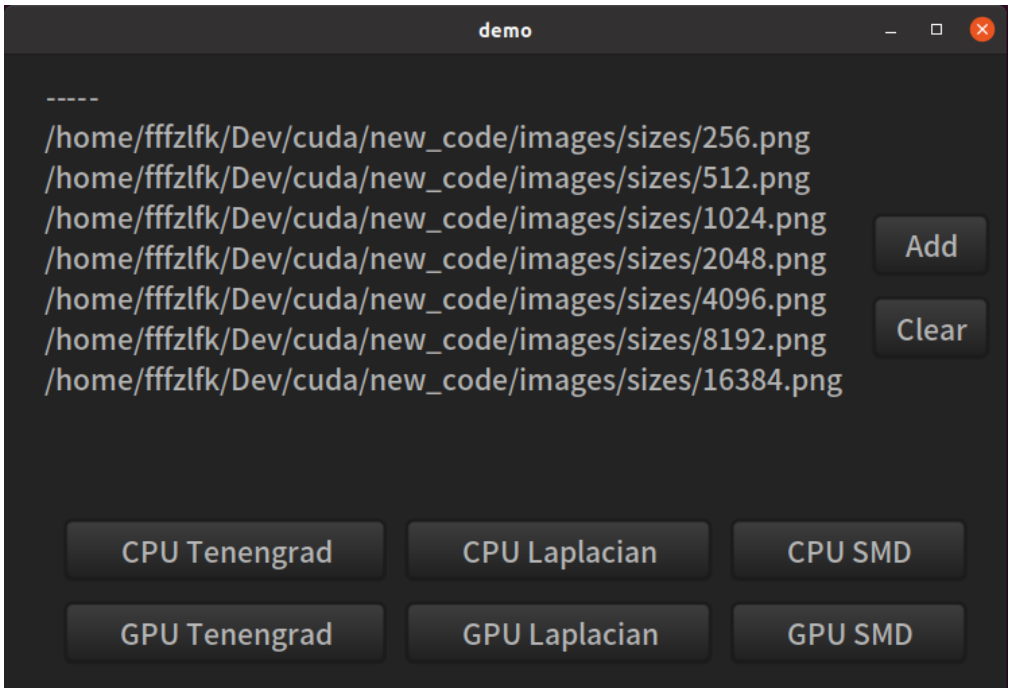

图 5.1 程序主界面

如图 5.1，点击 Add 按钮可以从文件管理器添加图像，添加完成后可点击相应按钮，选择加速前后的不同图像度量算法，给出度量值和运行算法运行时间。

5.1 有效性测试结果

为了验证度量算法的有效性，使用 LIVE 模糊数据集[15]中的测试样例分别对加速前后的度量算法进行了测试。如图 5.2 所示，使用的五张测试图像编号为 Image 1~5，它们是使用不同大小的高斯内核进行模糊处理后的图像，其模糊程度按编号依次递增，即清晰度依次递增。

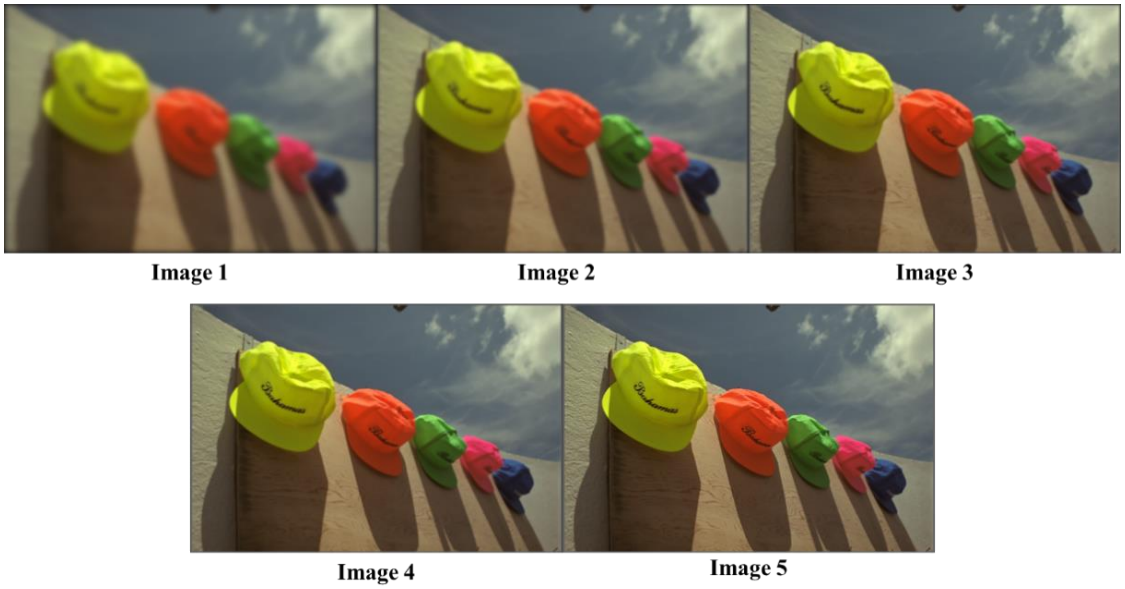

图 5.2 LIVE 数据集测试数据

测试结果表 5.1 所示。

表 5.1 加速后的度量算法有效性测试结果

图像编号	Image 1	Image 2	Image 3	Image 4	Image 5
LIVE 数据集参考值	4.17	6.92	8.29	8.84	9.32
Tenengrad（未加速）	257.33	522.41	929.39	929.39	2112.14
Tenengrad（加速后）	257.33	522.41	929.39	929.39	2112.14
拉普拉斯（未加速）	0.85	1.03	1.53	2.32	4.52
拉普拉斯（加速后）	0.85	1.03	1.53	2.32	4.52
灰度差分乘积（未加速）	0.83	1.81	3.67	6.05	12.28
灰度差分乘积（加速后）	0.83	1.81	3.67	6.05	12.28

如表 5.1 所示，实现的这三种图像清晰度度量算法给出的度量值和 LIVE 数据集的度量值走势基本一致，能正确反映图像的清晰程度；另一方面，加速前后的度量算法所给出度量值完全一致，这说明本文设计的图像清晰度度量算法加速方法是有效的。

5.2 实时性测试结果

使用了一组不同大小的图像，在两台不同的配置计算机上进行了对加速前后的三种评价函数进行了测试，两台计算机硬件配置如表 5.2 所示。

表 5.2 测试平台配置对比

		计算机 1	计算机 2
CPU	型号	Intel® Core™ i5-8250U	Intel® Xeon® Silver 4210
	主频	1.60GHz	2.20GHz
GPU	型号	NVIDIA GeForce MX150	NVIDIA Quadro RTX 5000
	设备计算能力	6.1	7.5
	全局内存大小	1.96MB	15.74MB
	主频	1.34GHz	1.81GHz
	内存总线位宽	64bit	256bit
	L2 缓存大小	524288B	4194304B

未使用 CUDA 加速的图像清晰度度量算法其运行时间主要由 CPU 主频决定，而使用 CUDA 加速后的度量算法的性能则取决于显卡的计算能力。NVIDIA 官方针对支持 CUDA 的 GPU，给出了它们的设备计算能力，如表 5.2 所示，计算机 1 的 GPU 设备计算能力为 6.1，而计算机 2 则达到了 7.5，属于性能十分出色的 GPU。本文使用这两台配置不同的计算机对度量算法进行了对比测试。

5.2.1 Tenengrad 评价函数

将加速前后的 Tenengrad 度量算法使用大小不同的测试图像在两台计算机上运行了 10 次，得到了平均运行时间，如表 5.3 所示（时间单位为毫秒）。

表 5.3 Tenengrad 度量算法加速前后实时性测试结果

图像尺寸	256x256	512x512	1024x1024	2048x2048	4096x4096	8192x8192
计算机 1 未加速	2.60	10.67	48.06	233.48	2171.10	9473.61
计算机 1 加速后	3.78	4.28	5.46	10.59	30.48	110.44
计算机 2 未加速	3.30	13.31	56.47	298.09	1648.44	8898.44
计算机 2 加速后	5.61	6.01	6.40	8.85	18.24	54.69

5.2.2 拉普拉斯评价函数

与 Tenengrad 评价函数一样，使用相同的测试数据在不同的计算机上对拉普拉斯度量算法进行测试，所得到的平均运行时间如表 5.4。

表 5.4 拉普拉斯度量算法加速前后实时性测试结果

图像尺寸	256x256	512x512	1024x1024	2048x2048	4096x4096	8192x8192
计算机 1 未加速	1.52	6.15	27.37	150.57	1892.01	8766.60
计算机 1 加速后	3.74	4.21	5.40	10.52	30.47	109.97
计算机 2 未加速	2.19	7.63	31.42	182.38	1149.63	6986.47
计算机 2 加速后	5.24	6.12	6.47	9.05	18.10	55.64

5.2.3 灰度差分乘积评价函数

灰度差分乘积评价函数测试结果如表 5.5 所示。

表 5.5 灰度差分乘积度量算法加速前后实时性测试结果

图像尺寸	256x256	512x512	1024x1024	2048x2048	4096x4096	8192x8192
计算机 1 未加速	1.29	5.36	25.22	134.06	1241.83	5306.69
计算机 1 加速后	3.78	4.30	5.38	10.32	29.64	106.89
计算机 2 未加速	1.60	6.37	29.07	152.71	888.39	4810.80
计算机 2 加速后	5.57	5.89	6.52	8.75	18.29	54.77

5.3 影响度量算法加速比的因素

在描述并行程序的性能时，通常使用加速比来衡量。一般来说，加速比定义为：

$$S_p = \frac{T_1}{T_p} \qquad \text{式(5-1)}$$

其中 T_1 为程序顺序执行所花费的时间，而 T_p 则是使用并行技术加速后程序所消耗的时间。

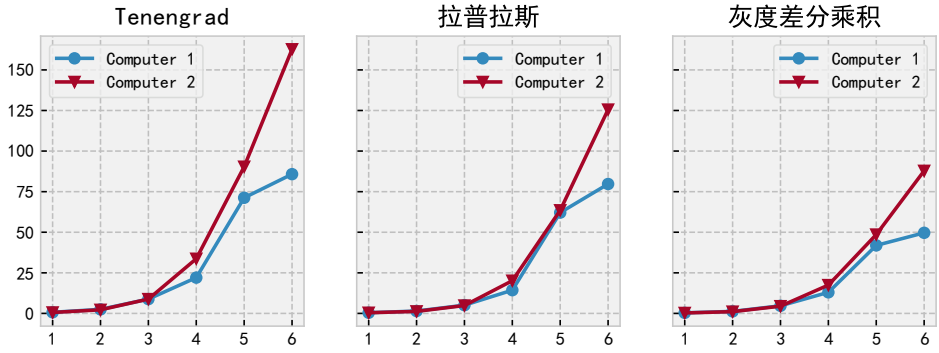

图 5.3 度量算法加速比

三种度量算法在不同计算机上针对不同大小图像的加速比如图 5.3 所示。

5.3.1 图像尺寸大小

如图 5.3，折线图的横坐标是测试图像的编号，它们的尺寸分别为256×256、512×512、1024×1024、2048×2048、4096×4096和8192×8192，纵坐标是度量算法的加速比值。可以看到，使用 CUDA 加速后的图像清晰度度量算法在针对较小尺寸的图像时效果较差，甚至对于256×256的图像其加速比小于1；而当输入图像的尺寸较大时，三种度量算法加速效果都十分显著。

随着图像尺寸的增长，度量算法的加速比也一直增大，主要原因是本文设计的 CUDA 加速方法的基本思路是针对图像的每一个像素使用一个 CUDA 线程去进行计算。在理想情况下，加速后的程序的加速比和图像像素数量成正比，虽然在实际执行过程中受到硬件方面的限制，但这也有很大的参考价值。

5.3.2 度量算法计算复杂度

此处度量算法计算复杂度指的是度量算法本身在计算度量值时操作的成本，本文中三种度量算法的加速方法第一个流程和第三个流程完全一致，因此计算复杂度的差异主要体现在计算单个像素度量值的过程，即加速方法的第二个流程。

在本文 3.2 计算单个像素度量值核函数具体实现中可以看到，Tenengrad 评价函数在三种度量算法中计算复杂度最高，其需要访问12次全局内存并进行8次乘法操作；拉普拉斯评价函数计算复杂度次之，需要访问 6 次全局内存和 2 次乘法操作；灰度差分乘积评价函数计算复杂度最低，其只需 4 次访问全局内存操作和1次乘法操作。

没有使用 CUDA 加速度量算法由于其在 CPU 单核心上顺序执行，所以不同计算复杂度的度量算法所消耗的时间差异较大。相比于未加速的度量算法，由于 CUDA 并行计算的优势，内存访问指令和计算指令被分摊到各个 GPU 核心上，因此度量算法的计算复杂度差异被强大的并行能力所稀释，这样导致加速后的度量算法所消耗的时间差别不大。

综上，计算复杂度较高的度量算法加速前执行所花费的时间较多，即加速比的分子较大，而其分母则和计算复杂度较低的度量算法差别不大，因此导致了计算复杂度较高的度量算法的加速比相比计算复杂较低的度量算法要高出一些。

5.3.3 GPU 性能

从图 5.3 中可以看到，两台计算机在针对大幅面的图像时，加速比差异比较

明显。当输入图像的尺寸达到8192×8192时，三种度量算法的加速比在计算机 2 上的加速比接近计算机 1 的两倍，而且从折线图曲线的趋势可以看出，计算机 1 随着图像尺寸的增大，其加速比曲线增长趋势放缓，而计算机 2 的加速比曲线则一直较为陡峭。这说明不同设备计算能力的 GPU，对 CUDA 加速程序的性能影响是巨大的。

CUDA 工具包提供了一个用来分析 CUDA 程序执行情况的工具 nvprof，使用 nvprof 可以收集在 CPU 和 GPU 上 CUDA 相关活动的执行时间线，给出内存传输、核函数执行和 CUDA API 调用等事件所消耗的时间和其他相关信息。

使用8192×8192大小的图像对加速后的 Tenengrad 评价函数在两台计算机上进行测试，并使用 nvprof 查看其执行情况，得到结果如表 5.6 所示。

表 5.6 加速后的 Tenengrad 度量算法执行情况

	计算机 1		计算机 2	
	Time	Time(%)	Time	Time(%)
CUDA memcpy HtoD	64.559ms	56.33%	49.040ms	93.39%
Tenengrad 核函数	20.889ms	18.23%	1.3707ms	2.61%
并行归约核函数	14.771ms	12.89%	0.96781ms	1.84%
灰度化核函数	14.396ms	12.56%	1.1314ms	2.15%

结合表 5.2 所示的两台计算机 GPU 的配置情况，从表 5.6 可以看出，有着更高性能 GPU 的计算机 2 的核函数执行耗时相比 GPU 配置较低的计算机 1 大大缩短，核函数执行效率提升了接近20倍。同时，对于大尺寸的图像数据，加速后的程序从主机将数据拷贝至设备全局内存这个过程耗时占了整个程序执行耗时的很大一部分，即使计算机 2 的 GPU 的内存总线位宽为512 bit，相比计算机 1 的 GPU 的64 bit 宽的内存总线提升了很大，但从主机到设备内存数据拷贝效率提升相比核函数执行效率的提升也不太明显。

综上，设备计算能力更强的 GPU 能够充分发挥出 CUDA 并行加速的优势，给加速后的程序带来更高的加速比，但这其中从主机内存拷贝数据到设备内存这个过程的代价是十分昂贵的，在设计 CUDA 程序对算法进行加速时，要尽可能地减少这种内存拷贝操作。

第六章 总结与展望

6.1 总结

本文聚焦于图像清晰度度量算法，分析和实现了经典的图像清晰度度量算法，在其基础上，基于 CUDA 设计和实现了针对大幅面图像的图像清晰度度量算法的加速方法。

首先，通过查阅国内外文献和其他资料，本文介绍了图像清晰度度量相关课题的研究现状和发展趋势，分析了几种经典的图像清晰度度量算法，并借助 OpenCV 库使用 C++编程实现了这些算法，之后对其进行了测试和对比。

其次，介绍了使用 CUDA 加速程序所涉及到的相关知识，并在前文工作的基础上选择了适合 CUDA 加速的度量算法，设计和实现了基于 CUDA 的图像清晰度度量算法加速方法，并且针对其中的并行归约核函数讨论了一些优化策略。

最后，本文验证了度量算法加速方法的正确性，并且对实现的度量算法进行了加速前后的对比测试。根据测试结果，加速后的度量算法针对大幅面图像时的实时性大大提高。除此之外，本文通过分析测试数据，讨论了影响度量算法加速比的因素。

6.2 未来工作展望

本文的工作是建立在前人研究的基础之上，将经典的图像清晰度度量算法使用 CUDA 进行并行加速，仅提升了度量算法针对大幅面图像时的实时性，未涉及到度量算法的无偏性和抗噪性等方面的研究，今后的研究工作可以兼顾度量算法各个方面的性质来开展。另一方面，本文设计的基于 CUDA 的图像清晰度度量算法加速方法可以进行进一步地优化，而且可以对影响算法加速比的因素可以进行更深层次的讨论。

致 谢

首先，我要感谢导师对本文工作的悉心指导。在论文开题阶段，李老师精确的指明了本研究工作的要点，为本文的完成打下了基础；在实现过程中，为笔者提供了大量的参考资料，解决了本文工作中很多的难点和重点；在最后的论文撰写阶段，也提供了很多建设性的意见。

其次，春晖寸草，我最要感谢的是自己的父母，他们辛劳的付出养育了我，给了我强大的勇气能够面对一次又一次的挫折与挑战。

另外，我还要感谢师兄对我实验过程中的耐心指导，不厌其烦地帮助我调试服务器搭建环境。

最后，我要感谢母校对我的栽培，愿母校发展日新月异，培养出更多更高层次的人才。

参考文献

[1] Fredj A H, Malek J. Real time ultrasound image denoising using NVIDIA CUDA[C]//2016 2nd International Conference on Advanced Technologies for Signal and Image Processing (ATSIP). IEEE, 2016: 136-140.

[2] 王勇,谭毅华,田金文.一种新的图像清晰度评价函数[J].武汉理工大学学报,2007,(03):124-126.

[3] 蒋婷,谭跃刚,刘泉.基于 SOBEL 算子的图像清晰度评价函数研究[J].计算机与数字工程,2008,(08):129-131+191.

[4] 王昕,刘畅.基于提升小波变换的图像清晰度评价算法[J].东北师大学报(自然科学版),2009,41(04):52-57.

[5] Yufeng L I , Niannian C , Jiacheng Z , et al. Fast and high sensitivity focusing evaluation function 一种快速高灵敏度聚焦评价函数*[J]. 计算机应用研究, 2010, 27(4):1534-1536.

[6] 张亚涛,吉书鹏,王强等.基于区域对比度的图像清晰度评价算法[J].应用光学,2012,33(02):293-299.

[7] 朱倩,姜威,贲晛烨等.梯度与相关性结合的自动聚焦算法[J].光学技术,2016,42(04):329-332.

[8] 曾海飞,韩昌佩,李凯等.改进的梯度阈值图像清晰度评价算法[J].激光与光电子学进展,2021,58(22):285-293.

[9] Ni J, Luo G, Yu T, et al. No-reference image sharpness Algorithm based on gradient shape[C]//2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI). IEEE, 2016: 786-790.

[10] Hassen R, Wang Z, Salama M. No-reference image sharpness assessment based on local phase coherence measurement[C]//2010 IEEE International Conference on Acoustics, Speech and Signal Processing. IEEE, 2010: 2434-2437.

[11] Gu K, Zhai G, Lin W, et al. No-reference image sharpness assessment in autoregressive parameter space[J]. IEEE Transactions on Image Processing, 2015, 24(10): 3218-3231.

[12] 杨勇.数字图像处理与分析[M].第 3 版.北京:北京航空航天大学出版社,2015:238-239.

[13] Gloria Bueno Garcia, Oscar Deniz Suarez, Jose Luis Espinosa Aranda. OpenCV 图像处理[M].北京:机械工业出版社,2016:163-170.

[14] 樊哲勇.CUDA 编程基础与实践[M].北京:清华大学出版社,2020:58-64.

[15] R. Sheikh, Z. Wang, L. Cormack and A. C. Bovik. LIVE Image QualityAssessment Database Release 2[Data set]. http:// live. ece. utexas. edu/ research/quality.