实验二 基于 GPU 的图像处理多线程编程

1.1 各部分功能完成情况展示

1.1.1 任务一：三阶插值的缩放和旋转（GPU 加速）

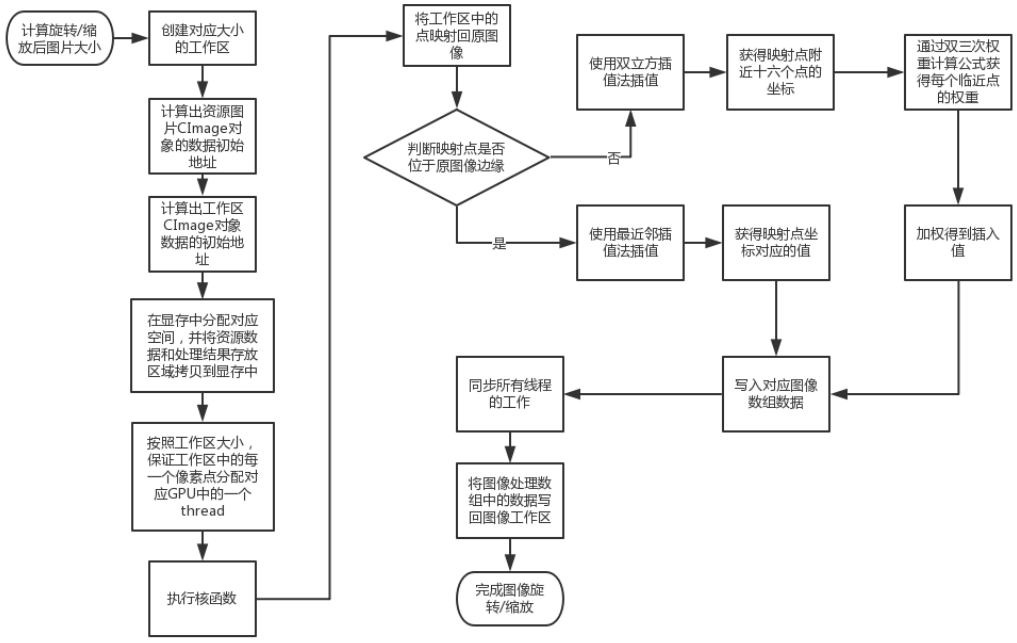

图 1 三阶插值的缩放和旋转的 GPU 加速版本程序流程图

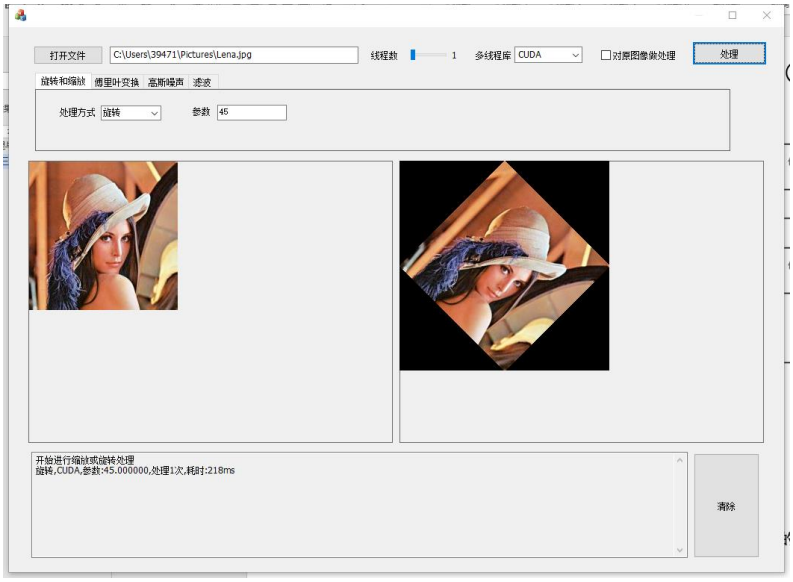

图 2 旋转结果

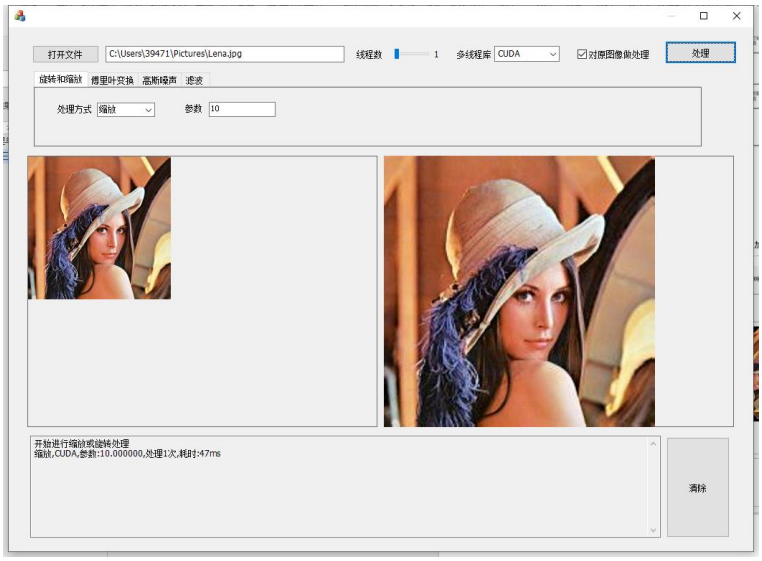

图 3 旋转结果

1.1.2 任务二：傅里叶变换 GPU 加速

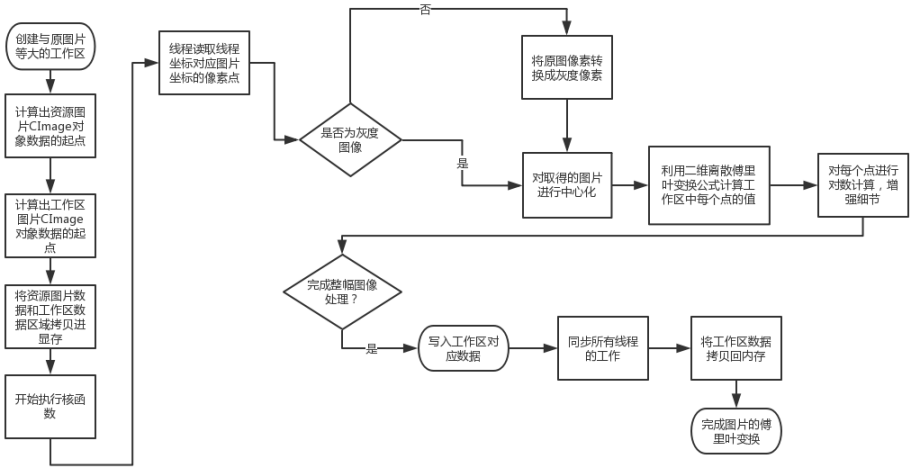

图 4 傅里叶变换 GPU 加速版程序流程图

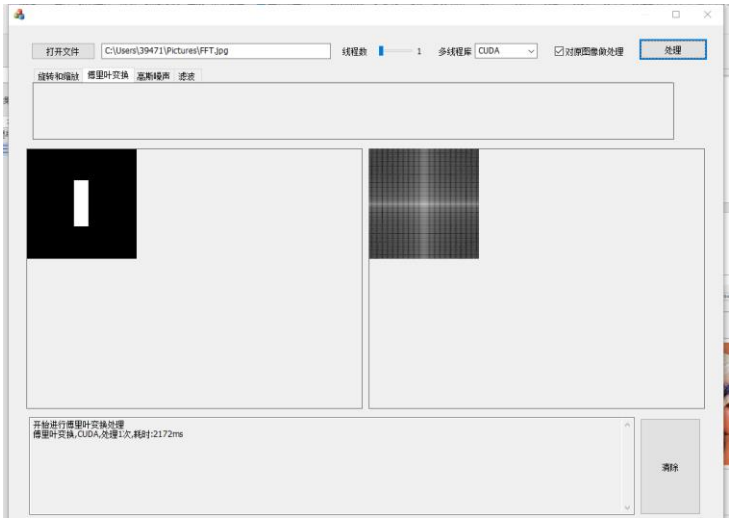

图 5 傅里叶结果

1.2 任务实现的关键方法

1.2.1 内存与显存之间数据交互

我认为这次任务的关键实现方法的突破点在于，显卡无法直接操作内存里面的数据，而是需要将数据拷贝至显存然后进行操作。

（1） 实例代码中是这样来进行数据之间的传递的。

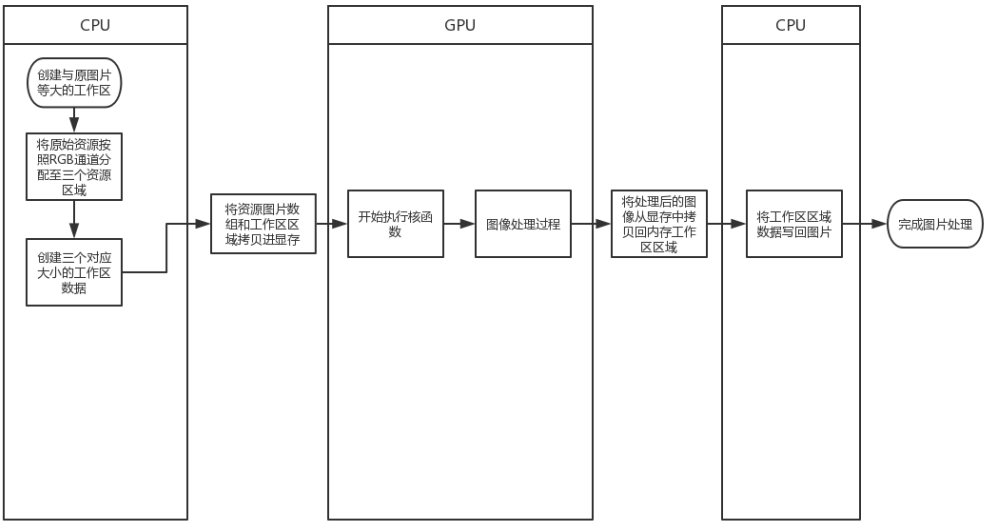

图 6 实例代码中内存和显存之间数据交互流程

经过大佬提醒和仔细思考后，发现这样的处理方式会让 CPU 有一个很大的负担。因为 CPU（单线程）需要遍历一次资源图片以及处理完成之后的图片，然后将每个像素的值从 CImage 对象按照通道写出来以及写回去 CImage 对象，而访问每一个像素的时候都需要计算偏移（进行两次乘法和一次加法运算），对于小图片来说，可能压力不是很大，但是一旦是对稍微大一点的图像进行处理，或者是进行缩放变换这种产生的最终图片很可能是一个很大的图片的变换，让 CPU（单线程）这样去遍历，计算，将会产生大量的冗余时间。

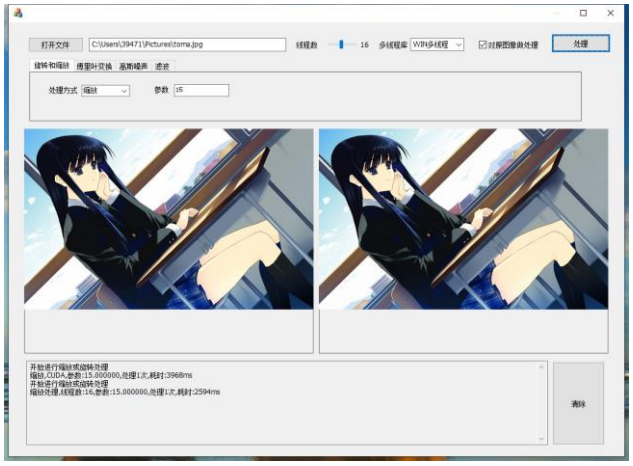

图 7 原逻辑下 GPU 缩放处理与 CPU 进行对比

图片大小	缩放倍数	处理后图片大小	CPU 时间	GPU 时间
1229*768	15	18435*11520	2594ms	3968ms
943872		212371200		约为 CPU 的 1.5 倍

表格 1 原逻辑下 GPU 缩放处理与 CPU 进行对比

（2） 优化后数据交互

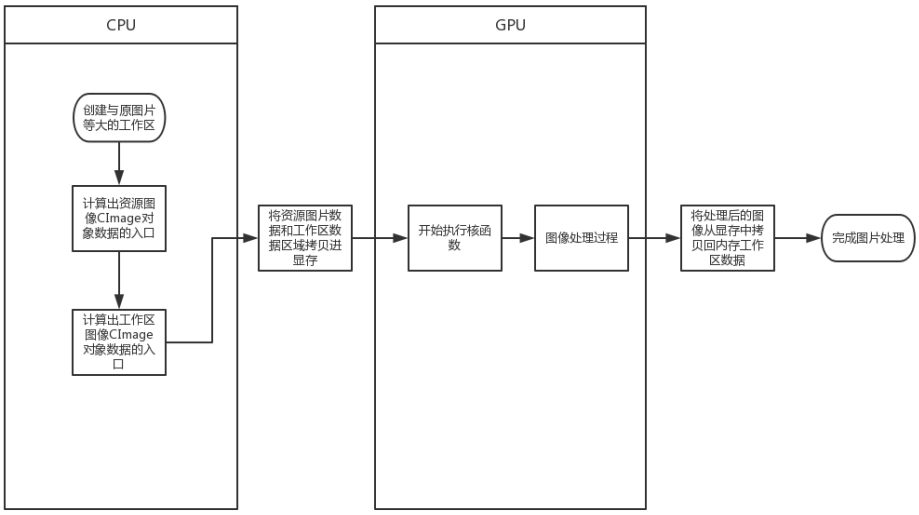

图 8 优化后的内存和显存之间数据交互流程

可以看到，优化后的做法省去了中间数据缓存这一个部分，直接将资源数据拷进显存，然后，在处理结束之后，也是直接将数据拷贝回内存就可以了，不需要把数据从缓存区写入图像。因此，可以把很大一部分 CPU 的压力缓解了。

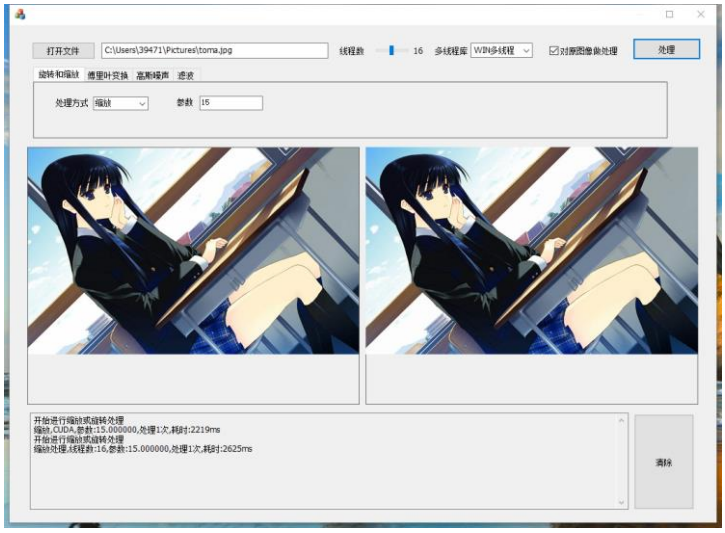

图 9 原逻辑下 GPU 缩放处理与 CPU 进行对比

图片大小	缩放倍数	处理后图片大小	CPU 时间	GPU 时间
1229*768	15	18435*11520	2625ms	2219ms
943872		212371200		约为 CPU 的 0.85 倍

1.2.2 TDR(Timeout Detection and Recovery)调整

这次实验中，遇到的另一个困扰了很久的 bug 是，在缩放处理较大图片的时候或者进行傅里叶变换的时候，最后的结果会出错。在确保算法是正确的情况，上网查找了许多资料，发现有说法说，是 GPU 的 kernel 函数在①执行较为复杂的算法②执行较大运算量的时候，会导致运算结果出错。

于是，在使用 nsight 调试工具进行单步调试以及使用输出调试之后，发现，即使进行较为复杂的计算（傅里叶），在崩溃之前输出的结果仍然是正确的。因此，不存在说，执行较为复杂算法会导致计算出错的情况。

那么会不会是运算量较大的时候，会出现问题呢，于是，我在原来 Grid 和 Block 都是二维的基础上，实现了一个二维 Grid，三维 Block 的 thread 分配，使用第三个维度来分配更多的 thread，从而减少单个 kernel 函数的运算量。结果还是失败了，也就是说，并不是因为单个 kernel 的运算量过大而导致的结果错误。

最后，在调试过程中，发现了另外一个可能原因，windows 系统中，有一个 TDR（Timeout Detection and Recovery）参数，用于检测显卡的计算时间，当显卡的计算时间超过所设定的时间，就会让显卡停止工作来达到恢复的效果。然后，发现 windows10 默认的 TDR 是 2s，将其调整至 100s 以后，发现最后的结果正确了。

1.3 性能对比分析

1.3.1 运行环境

因为本次对比涉及不同硬件之间性能的对比，因此，先在此列举一下本机的设备环境（高端，低端等判定均来自脚本之家网站的天梯图）

CPU：AMD Ryzen 2700　8 核　16 线程（高端 CPU）

GPU：GTX 960　2G 显存　NVIDIA 计算能力　5.2　（中低端 GPU）

1.3.2 性能对比

(1) 测试数据

注：

(a) 以下数据均为执行 5 次/10 次后，去除波动较大的数据，取平均值所得。

(b) 计时器最小测量单位为 16ms

处理方式 / 图像处理	图像大小	CPU					GPU
		1 线程	8 线程	16 线程	24 线程	32 线程	
放大 10 倍	160*160	250ms	39ms	39ms	39ms	40ms	31ms
	1229*768	9375ms	1454ms	1172ms	1190ms	1315ms	913ms
旋转 45°	160*160	0ms	0ms	0ms	0ms	0ms	0ms
	1229*768	149ms	31ms	24ms	24ms	32ms	31ms
傅里叶变换	160*160	16172ms	2078ms	1620ms	1640ms	1728ms	1188ms
	391*220	182000ms	23100ms	18200ms	18500ms	19587ms	12546ms
处理方式 / 图像处理	图像大小	CPU					GPU
		48 线程	64 线程	128 线程	256 线程		
放大 10 倍	160*160	40ms	40ms	39ms	54ms		31ms
	1229*768	1200ms	1218ms	1226ms	1172ms		913ms
旋转 45°	160*160	0	10ms	24ms			0ms
	1229*768	24ms	24ms	32ms			31ms
傅里叶变换	160*160	1725ms	1680ms	1664ms	1656ms		1188ms
	391*220	19031ms	18440ms	18296ms	18282ms		12546ms

表格 2 各个操作的 GPU 和 CPU 性能对比

(2) CPU 线程数性能对比

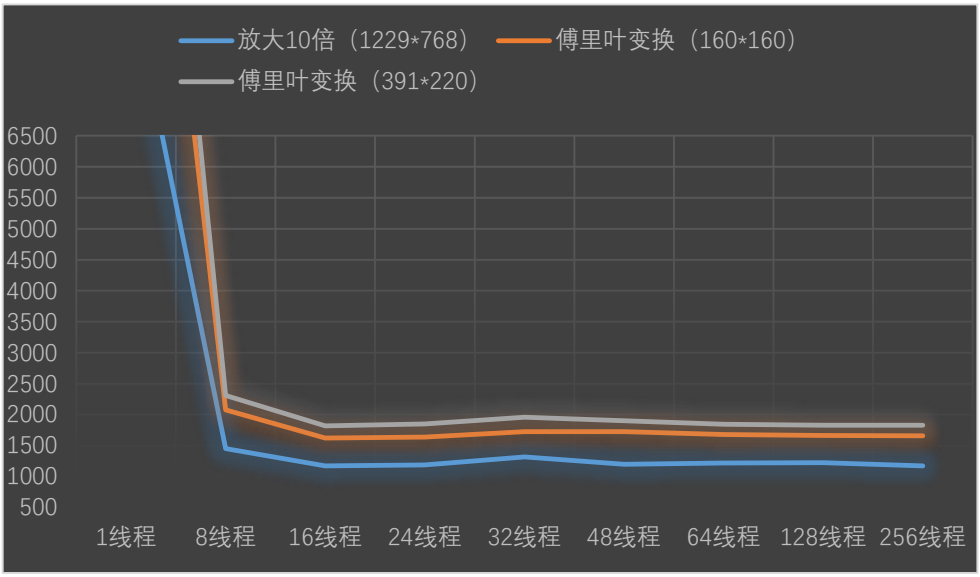

图 11 CPU 线程数与性能的折线图

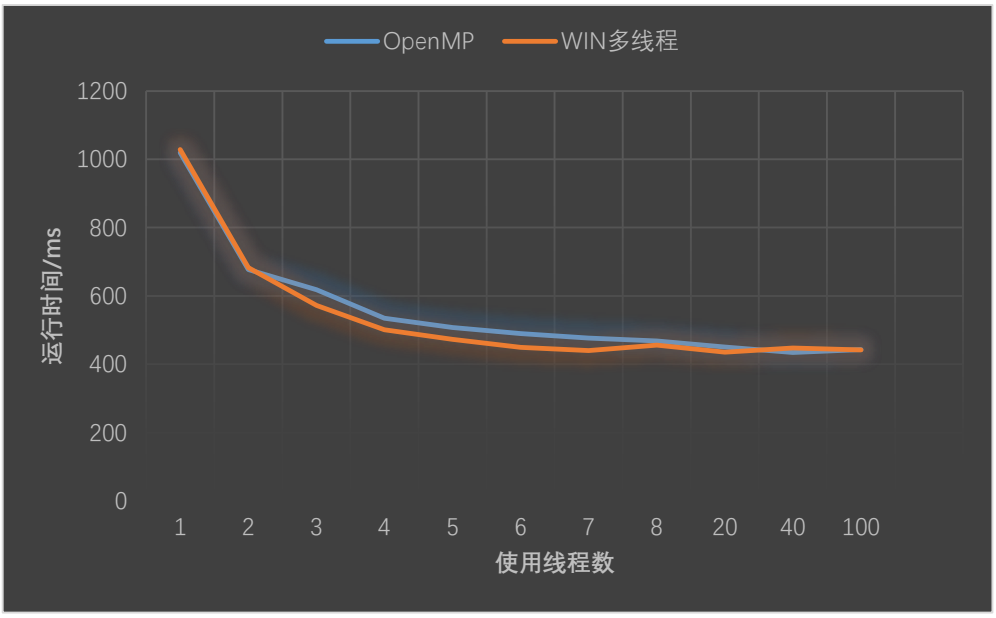

图 10 上次实验中的线程数与性能的折线图

由于旋转的计算量太小，计时器的精度不够，因此，很难针对结果比较 CPU 和 GPU 性能，不过，值得注意的是，在旋转 45° 时，如果使用 128 线程来运行，在其他运行时间均为 0ms，即均小于计时器最小计时单位 16ms 时，会有 16ms-32ms 之间的运行时间，可以猜想这是多线程分配的额外开销导致的结果，不过由于计时器精度有限，难以测量出线程分配开销的具体值。

在放大 10 倍和傅里叶变换这两种计算量较大的图形处理中，上一次实验得到的结论仍然成立，在核数范围内，几乎是一个线性的时间，从 1 线程到 8 线程，时间也几乎是 1/8。然后，在超线程（CPU 核心数<线程数<逻辑处理器数）的时候，效率大概提升 10%到 20%，这些都和上次实验得到的结果相符。

但是,在上一次实验报告中，时间随着线程数的减小逼近最小值的状况在这次没有出现，这次是在与逻辑处理器数持平时，达到最高的性能点。然后在两倍逻辑处理器数量的线程数

的时候达到性能倒退的极点，然后随着线程数的增加之后，性能逐渐上升，但是也仅是逐渐逼近之前的性能最高点。

两次实验都是在 windows 平台下进行的数据测试，而且使用的测试方法也相同，但是不同的是本次实验使用的是 AMD 生产的 CPU，上次是使用的 Intel 的 CPU，我猜想会不会是 windows 对两种平台的 CPU 的调度有所区别导致。

(3) GPU 和 CPU 性能对比

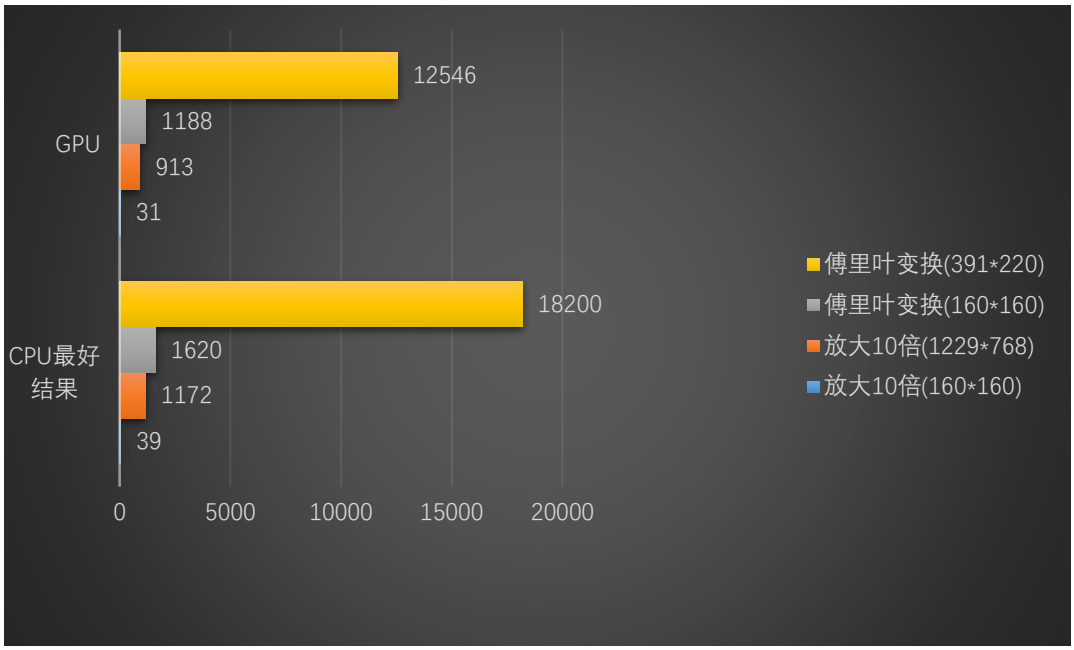

图 12 CPU **和** GPU **性能对比**

可以看到，尽管使用的是中低端的 GPU，但是在进行图片处理的时候，时间却比高端 CPU 更短，可以看到 GPU 在进行图像处理这种适合并行处理的任务时候的加速效果。