译文 2

作者：Yafei Mao; Lluis Abello; Utpal Sarkar; Robert Ulichney; Jan Allebach

出处：IEEE International Conference on Image Processing. 2018:3973-3977.

基于通用图形处理器的误差扩散半色调算法并行实现

摘要：通用图形处理器（GPGPU）集成了大量执行单元，因此非常适用于高速图像处理。然而，误差扩散半色调算法难以利用大规模并行处理架构的优势——该算法不仅依赖邻域像素的计算结果，还需反馈输出误差。本研究通过分析依赖图识别出无依赖关系的可并行处理像素，并结合 GPGPU 的特性，提出一种同步开销更低的并行处理方法。

关键词：半色调、误差扩散、通用图形处理器、并行图像处理

1 引言

以 NVIDIA GTX 285 为代表的通用图形处理器（GPGPU）可通过数千个线程加速各类图像处理核函数的执行。但误差扩散算法 [1]（一种将多灰度级图像转换为二值图像的数字半色调技术）因包含反馈机制，难以实现并行化。图 1 展示了经典误差扩散过程的框图。已有研究针对误差扩散算法的并行实现展开探索：包括基于单指令多数据（SIMD）架构的设计（对比了三种并行处理方案 [2]），以及基于 MasPar 架构的实现 [3]。然而，GPGPU 需要数百以上的并行度，而 SIMD、超长指令字（VLIW）或超标量等传统并行架构的并行度通常不超过 16[4]。此外，误差扩散算法的像素处理顺序严格，其并行实现还需同步操作；GPGPU 中不同流式多处理器（SM）间的同步操作不仅困难且耗时，因此需尽可能减少同步次数。

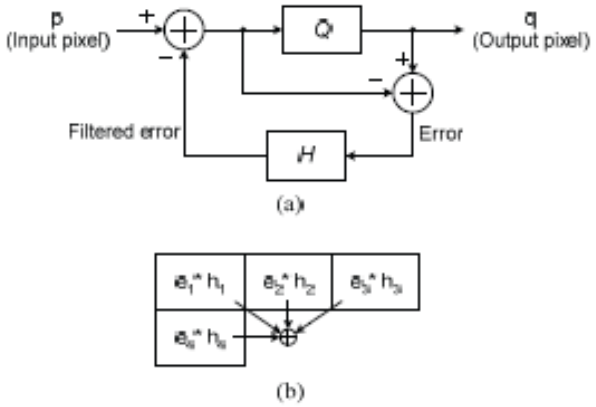

图 1:（a）误差扩散算法框图;（b）弗洛伊德-斯坦伯格滤波核 H

本研究采用 NVIDIA GTX 285 硬件及统一计算设备架构（CUDA）[5]：该硬件包含 30 个流式多处理器（SM），每个 SM 配备 8 个处理单元。为充分利用流水线处理单元，需为每个 SM 分配至少 32 个线程（称为一个线程束，warp）；同时需为每个 SM 分配多个线程束，通过任务切换掩盖全局内存访问延迟。在同一 SM 上执行的线程构成线程块（thread-block），一个线程块可包含多个线程束（每个线程束 32 个线程），因此线程块的线程数通常为 32 的整数倍。GPU 支持两级同步机制：同一线程块内的线程可借助专用硬件实现快速本地同步，而线程块间的同步需通过全局内存（DRAM）完成，耗时显著。每个 SM 包含四种片上内存：寄存器、共享内存、纹理缓存和常量缓存，其中共享内存可供同一线程块内的线程访问，且访问速度远高于全局内存。程序通过运行时调用 CUDA 接口管理设备内存（DRAM）空间，包括全局内存、常量内存和纹理内存。

本研究基于 GPGPU 的数千个线程实现了图 1 所示的误差扩散算法。尽管已有部分更适配并行处理硬件的误差扩散算法 [6]–[8]，但本研究仍选用图 1 中的经典算法——因其在数字印刷行业应用最为广泛。为消除线程间的依赖关系，采用斜扫描线（SSL）技术 [2]，该技术可利用相邻扫描线中斜向像素的独立性；同时，仅在线程块操作开始时执行耗时的线程块间同步，以降低同步开销，并解决了斜扫描线技术存在的负载不均衡问题。

本文结构如下：第 2 节阐述基于 GPU 的误差扩散算法并行化与实现技术；第 3 节展示实现结果；第 4 节给出结论。

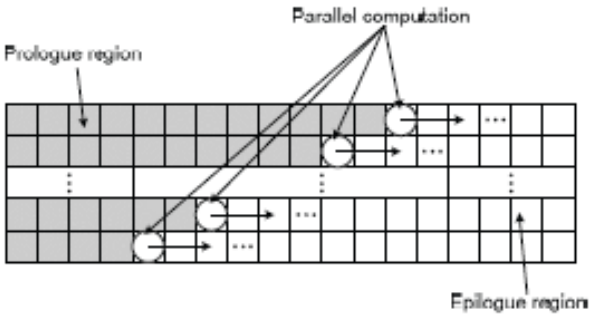

图 2: 分配给线程块的图像条带及条带内不同扫描线中的独立像素

2 基于 GPGPU 的并行实现

2.1 并行算法

GPU 可支持高度并发执行，因此不宜在单个像素处理层面挖掘并行性（误差扩散滤波核 H 仅包含少量可并行执行的算术运算），而需同时处理多个像素。同扫描线多像素处理的推测量化（SQ）方法 [2] 可适配 GPGPU 架构，但该方法虽能实现完美负载均衡，却存在补偿步骤耗时随并行度提升而增加、线程块间同步需求高等问题。

本研究采用改进版斜扫描线方法 [2]：在误差扩散算法中，看似需完成第 i 行所有像素处理后才能开始第 i+1 行，但深入分析可知，当处理第 i 行第 j 列像素时，可并行处理第 i+1 行第 j-2 列像素——这是因为第 i+1 行第 j-2 列像素仅依赖第 i 行 j-3、j-2、j-1 列像素的误差值。因此，(i,j)、(i+1,j-2)、(i+2,j-4)…位置的像素可并行处理（其中 (i,j) 表示第 i 行第 j 列像素）。当图像宽度为 W 时，该方法可支持多达 W/2 个并行线程；考虑到多数印刷图像的宽度达数千像素，该方法提供的并行度已足够。但该方法存在两大缺陷：一是斜向处理导致的负载不均衡，二是同步开销过大。如图 2 所示的起始区域（prologue），并非所有线程都能被有效利用；同理，算法还存在收尾区域（epilogue），需让提前启动的线程等待其他线程完成。对于宽度为 W 的图像，若使用 W/2 个线程，负载不均衡导致的效率下限为 50

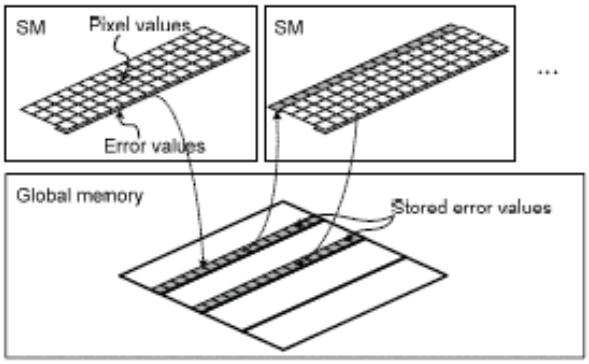

图 3: 线程块间的误差值交换

图 2 展示了分配给线程块的图像条带：灰色框代表已处理像素，带圈框代表当前正在处理的像素。线程块中的一个线程负责处理条带的一条扫描线，线程需与其他线程同步，按顺序处理扫描线中的像素。需注意，同一线程块内的线程同步是必需的——若斜向像素组中任一像素（首行像素除外）未与其他像素同步，将无法获取正确的误差值。与线程块间同步不同，线程块内同步的开销极低，仅需 4 个时钟周期 [5]。但该方法仍需线程块间同步：为使下一个条带尽快启动，需在当前条带计算出所需误差值后触发同步。线程块间同步通过全局内存访问实现消息传递，并通过极少执行同步（例如仅在条带首行的起始像素处同步）来最小化同步次数；同时，通过实验确定线程块间处理的时间裕量，确保下一条带的线程不会违反依赖关系。

负载不均衡、同步开销与全局内存访问次数之间存在权衡：增加条带高度（即每个线程块分配更多线程）会加剧负载不均衡，但可减少全局内存访问次数和线程块间同步开销。

2.2 实现细节

在 GPU 实现中，线程从全局内存加载自身负责的输入像素，利用存储在共享内存或全局内存中的其他误差值计算当前像素的误差值，随后计算输出像素值并写入全局内存。该过程中需尽可能减少全局内存访问次数——读取全局内存数据至少需要 400 600 个时钟周期 [5]。线程块间共享误差值时需将其存储在全局内存，但仅需存储线程块最后一行的误差值（供下一个线程块使用），而非所有误差值（图 3）。

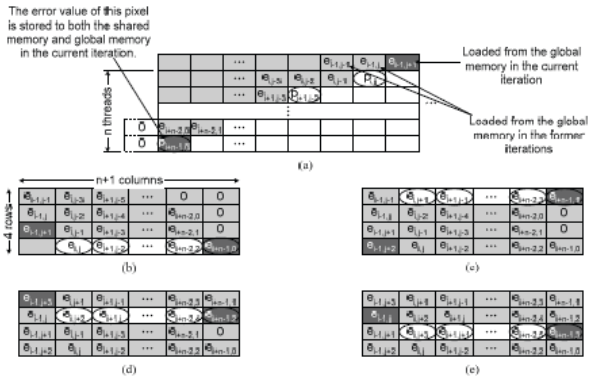

图 4: 线程块中的像素与计算得到的误差值，以及共享内存数组的内容

图 4 展示了误差扩散算法 GPU 实现中共享内存的使用方式:（a）为线程块正在处理的图像条带,浅灰色框为已处理且包含误差值的像素,带圈框 $p_{i,j}, p_{i+1,j-2}, \ldots, p_{i+n-1,0}$ 为当前步骤正在量化的像素（即构成一条并行处理的斜向线）。处理 $p_{i,j}$ 需使用误差值 $e_{i-1,j-1}, e_{i-1,j}, e_{i-1,j+1}$ 和 $e_{i,j-1}$——其中 $e_{i-1,j+1}$ 需从全局内存读取，其余误差值存储在共享内存中。（b）（e）为共享内存中存储的误差矩阵（大小为 4 行 ×(n+1) 列，n 为线程块的线程数）：-（b）中，本轮迭代生成的新误差值 $e_{i,j}, e_{i+1,j-2}, \ldots, e_{i+n-1,0}$ 存储在第四行；计算第二列第四行的 $e_{i,j}$ 时，需使用第一列的 $e_{i-1,j-2}, e_{i-1,j}, e_{i-1,j+1}$ 和正上方的 $e_{i,j-1}$。这种 L 型访问模式适合同一斜向线的所有像素，且因采用步长为 4 字节的线性寻址，无存储体冲突（bank conflict）。- 完成当前像素量化并计算误差值后，处理下一条斜向线 $p_{i,j+1}, p_{i+1,j-1}, \ldots, p_{i+n-1,1}$，新误差值存储在共享内存误差矩阵的第一行（c）——即采用循环缓冲区机制，最后一行的下一行是第一行。由于（b）中第一行的数值不再复用，可存储新误差值 $e_{i,j+1}, e_{i+1,j-1}, \ldots, e_{i+n-2,3}$ 和 $e_{i+n-1,1}$。- 后续迭代重复上述过程，持续更新共享内存内容（d）和（e）；当（e）中带圈框对应的误差值计算完成后，数组中误差值的相对位置与（b）一致，完成一个循环。

所有线程块完成计算后，即可得到最终的半色调图像。但如前所述，线程块依赖前一个线程块的计算结果，因此需同步。图 5 展示了整幅图像分配给 N/n 个线程块的情况（N 为图像高度，n 为线程块的线程数）：处理过程中，下一个线程块需等待前一个线程块处理到图像的特定像素位置 m；当前一个线程块到达该位置时，向全局内存写入特定值，下一个线程块从核函数启动时持续读取该值，确认值被修改后开始自身处理。该同步过程持续至最后一个线程块启动。当图像宽度较小或裕量 m 足够大时，每个线程块仅需在起始点执行一次同步操作，因此开销可忽略。

3 实现结果

为评估实现性能，对比了两种方案的执行时间：运行在 CPU 上的传统 C 程序，以及运行在 GPU 上的 CUDA 程序。实验平台为 Intel Core 2 Quad CPU（主频 2.83 GHz）和 NVIDIA GTX 285 GPU（240 个可用核心，主频 1.48 GHz），操作系统为 Linux Fedora 10，分别使用 GCC 编译 C 代码、NVCC 编译 CUDA 代码。

实验中有两个关键参数影响执行时间：1. 线程块大小：如前所述，影响负载均衡、全局内存访问次数和线程块间同步开销；2. 裕量 m：用于避免频繁同步，其取值依赖线程块大小和图像宽度（通过实验确定）。例如，图像尺寸为 1K×1K 像素、线程数为 32 时，m 取 40。m 越小，加速比越高，但过小可能导致当前线程块的处理速度超过前一个线程块；为稳妥起见，可检查从先前条带读取的误差值是否在预设范围内。

图 5: 线程块间的同步机制

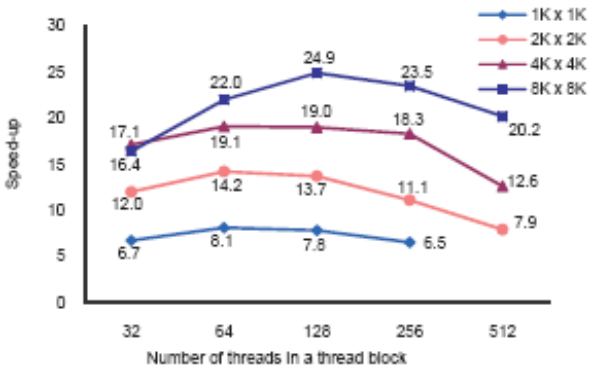

图 6: 不同线程块大小和图像尺寸下的性能表现

综合上述因素，测试了不同尺寸图像的加速比（图 6）：总体而言，线程块大小为 64 128 时可获得最大加速比——此时负载均衡、全局内存访问次数与线程块间开销的权衡达到最优。需注意流式多处理器（SM）的占用率：线程块过大时性能下降的主要原因是片上内存资源受限（线程数增加导致每个线程可用内存减少）。此外，图

像尺寸越大，加速比越高——这是因为大尺寸图像可激活更多线程，提升 GPU 利用率。

若计入主机与设备间的数据传输时间，加速比会降低（表 1）——数据传输时间占比显著。例如，8K×8K 像素图像的核函数执行加速比达 24.9，但计入数据传输时间后降至 9.8。不过，当多个应用以核函数组形式在 GPU 上执行时，该开销的影响可忽略。

表 1：不同图像尺寸（像素）下的核函数执行时间（秒）与数据传输时间（秒）对比

图像尺寸	核函数执行时间（a）	数据传输时间（b）	CPU 时间（c）	加速比 c/a、c/(a+b)
1024 × 1024	0.004865	0.002212	0.039457	8.1　5.6
2048 × 2048	0.010066	0.007886	0.143056	14.2　8.0
4096 × 4096	0.026404	0.030981	0.505470	19.1　8.8
8192 × 8192	0.073270	0.112392	1.824052	24.9　9.8

4　结论

本研究基于 GPGPU 的数千个线程实现了误差扩散半色调算法：采用斜扫描线方法识别独立像素，通过减少线程块间同步次数、高效利用共享内存最小化开销。结果表明，与运行在 PC 上的传统 C 程序相比，该技术可实现约 24.9 倍的加速比。