《计算机与地球科学》45（2012）212–220

目录列表可在SciVerse ScienceDirect上找到

计算机与地球科学期刊主页：www.elsevier.com/locate/

cageo

利用GPU加速空间栅格分析的批量处理

Mathias Steinbach [*]，Reinhard Hemmerling [1]

德国哥廷根大学计算机图形学和生态信息学系主任

ARTICLE INFO

文章历史：
2011年7月3日收到
收到修订版本
7 2011年11月
接受日期：2011年11月8日
2011年11月17日在线提供

关键词：
OpenCL GPU GIS
光栅分析缓存

ABSTRACT

地理信息系统（GIS）处理栅格数据的批处理操作耗时较长。现代高性能图形处理器（GPU）能够并行执行数百次算术运算，这有助于大幅缩短此类操作的计算时间。此外，大多数常用的栅格处理操作都存在I/O瓶颈问题——硬盘与内存之间的数据传输所耗费的时间，往往比实际运算过程更长。本文旨在提出一种高效的栅格数据两级缓存策略，并通过GPU加速选定的栅格操作，这些操作已作为插件集成到开源软件GRASS中。我们将展示一个基于真实应用场景的数据流示例，并测量和讨论可获得的实际预期加速效果。

1. 介绍

空间栅格分析操作几乎是每个地理信息系统（GIS）的标配功能。根据栅格数据的规模，这类操作大多运行速度极快，用户通常能在2秒内获得结果。但在复杂分析场景中，需要通过批处理方式连续执行大量栅格运算，这可能导致运行时间飙升至数小时甚至更久，严重影响效率。

造成这种长时间运行的原因有两个。首先，许多GIS软件只使用单线程执行计算，因此无法利用最新的多核CPU甚至GPU。其次，中间结果被写入硬盘，并在下次操作时重新加载。避免这些冗余的传输将加快批处理。

本文的贡献包括两种创新方法：一是实现中间数据的高效缓存，二是通过移植现有算法至GPU来提升运算速度。该实现方案以开源GIS软件GRASS的插件形式提供。文中通过示例批处理流程展示扩展功能的应用场景，并量化了该技术方案预期带来的加速效果。

2. 相关工作

图形处理器已成为加速科学计算的强大工具。例如，Viola等人（2003年）、Yang等人（2008年）等研究者...

Temizel等人（2011年）已率先运用GPU加速图像处理。空间栅格数据本质上也是一种图像形式，因此GPU同样可以成为地理信息处理领域的利器。Beutel团队（2010年）提出基于GPU的算法来加速插值运算，而Zhang研究组（2010年）则开发出专门用于处理海量栅格文件的GPU架构数据结构。沃尔什等人（2009）提出了加速地球物理模拟（如地球流体、地震学和磁力显微镜）的研究成果，而奥尔特加与鲁埃达（2010）则利用GPU加速了排水网络的确定过程。研究者们证明，基于GPU的模拟运算速度比CPU版本快八倍以上。吴等人（2007）提出了一套“利用GPU提升GIS空间栅格分析性能”的通用框架，他们证实GPU算法明显优于CPU算法，但尚未将其应用于实际GIS系统。该团队的核心创新在于无需将中间结果存储在硬盘即可实现数据复用。曼尼福德公司推出了首款支持GPU的GIS系统，成功加速了基于邻域分析的栅格运算任务。

3. 我们的方法

我们的目标是分析栅格数据批量处理的潜在加速潜力。为此，我们采用开源地理信息系统GRASS（Neteler和Mitasova，2008）作为开发平台，用于基于GPU的栅格运算，并将其作为基准来比较计算时间。我们开发了利用GPU计算能力的GRASS模块。通过访问OpenCL驱动程序（OpenCL，2010），GRASS模块能够与GPU进行通信。OpenCL承诺只要存在可用的OpenCL驱动程序，就能独立于计算设备和操作系统。通常情况下，OpenCL

[*] 通讯作者。电话：+49 163 8269280。
电子邮箱地址：mathias.steinbach@yahoo.com（M. Steinbach），rhemmer@gwdg.com（R. Hemmerling）。
[1] 电话：+49 551 393697。

0098-3004/$ -请参阅前言及2011 Elsevier Ltd.版权所有。doi：10.1016/j.cageo.2011.11.012

算法并不局限于GPU。此外，（多核）CPU也能执行它们。

首先，我们对GRASS中常用模块进行了并行化改造。每个模块都被重新设计为并行化版本，通过GPU进行计算。所有模块均以独立进程运行，数据交互完全依赖硬盘存储。为此我们特别开发了一个模块，它能整合所有并行算法，并根据用户提供的BASIC脚本执行批量处理。该模块优化了GPU内存、RAM与硬盘之间的数据传输效率。通过示例数据流（如图3所示，其BASIC脚本详见清单1），我们不仅测定了新方案的运行时间，还与GRASS标准栅格运算进行了性能对比。

清单1-以BASIC脚本形式列出的数据流。

```
声明变量
将dim i转换为浮点型
将dim区域设为栅格，x设为栅格，y设为栅格；将dim掩膜
设为栅格；将forest设为栅格；将forest500设为栅格。
将栅格转换为EDT格式，将平均值转换为栅格

木材的承载面积
面积=Load（ “areas_of_wood ”）

‘掩体林区
mask =如果（areas==empty，0,1）

存储邻域分析的结果（存储FocalSum mask，参数80、
100、50)，命名为“N50” “
存储（FocalSum mask，80、100、25)，“N25”）
存储（FocalSum mask，170、190、50)，“W50 ”）
存储（FocalSum mask，170、190、25)，“W25”）

带坐标的载荷图
x=Load（ “x-coordinate” ）
y=Load（ “y-coordinate” ）

i=0
重复
  将计算区域设置为映射区域
  SetRegion（区域，“rast ”）

  选择森林
  forest = If(areas == i, 1, empty)

  ‘调整森林大小
  EDT=EucDistance（森林）
  forest500 = If(EDT < = 500,1,empty)

  最小化计算区域
  SetRegion(forest500，“zoom ”）

  平均forest500坐标
  平均值= MaskMean（X，forest500）
  储存（平均值，“MeanX” + i）
  平均值= MaskMean（Y，Wald500）
  储存（平均值，“MeanY” + i）

  i = i+1
如果< 884then
  继续
endif
```

3.1. 数据流示例

该数据流基于一张包含844个林地的地图（见图1），其尺寸为2275×2263像素，对应22.7公里×22.6公里的区域。整个流程包含两个主要步骤：首先对整张地图进行邻域分析，统计特定邻域内的单元格数量。其中包含木材。然后对每片林地进行平均值分析，从而计算出每片林地的某种中心。

我们不会讨论这些结果，而是想重点讨论这个简单任务的运行时间。我们无法分析每一个可能的栅格操作。因此，我们选择了一个包含常用标准操作的数据流。我们在下面描述这些操作。

第一步（邻域分析）需要统计每个含木头的单元格周围特定邻域内的单元格数量。ArcGIS提供了多种预定义的邻域形状，我们选用‘楔形’形状，因其运行时间异常长。图2展示了示例：棕色单元格为含木头区域，白色单元格为空白区域。我们需要统计这些含木头单元格的数量，并将结果存储在标记单元格中。该分析需针对每个单元格单独执行，通过创建掩膜实现计数——将含木头单元格设为1，其他单元格设为0。在GRASS中使用地图运算（r.mapcalc函数）即可获得此结果，而ArcInfo则通过AML命令IF提供相同功能。图3(2)展示了最终结果。

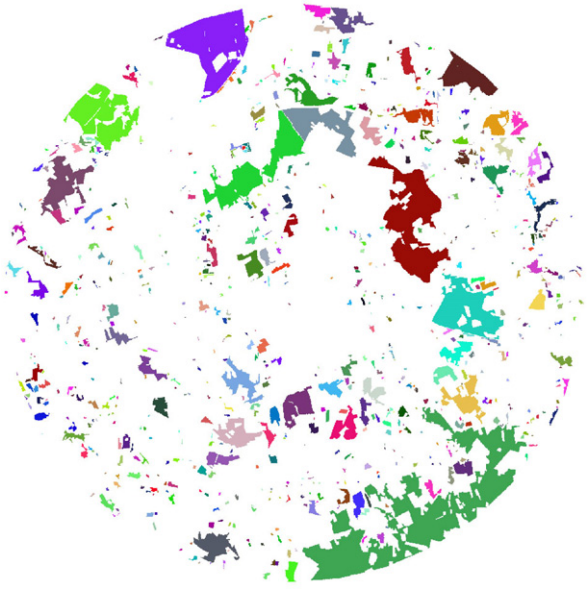

图1.德国林地分布图，比例尺为515公里^2，共500万格，每种林地用不同颜色表示。

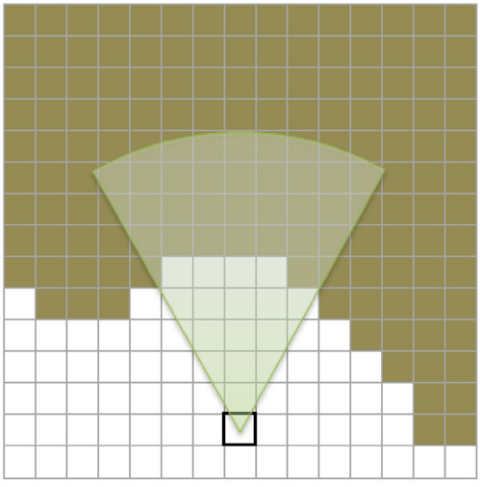

图2.一个细胞的楔形邻域中的细胞。

现在我们可以统计该细胞周围楔形区域内所有单元格的数值。我们使用AML的FocalSum命令，通过不同大小的邻域进行计算。虽然GRASS软件没有专门的此类命令，但可以通过r.mfilter工具模拟小尺寸邻域的计算。四个结果如图3（2.1-2.4）所示。第一个楔形区域由170度和190度两个角度覆盖，半径为50个单元格(500米），方向朝西。

第二步（平均值分析）针对每片林地单独进行，因此需要通过地图代数从地图中选取一片林地。图3(1)展示了特定林地的选择过程。

接着，我们为每个地块计算其到当前林地的最短距离。生成的地图称为欧氏距离变换（EDT），如图3（1.1）所示。ArcInfo的EucDistance函数可计算精确的EDT，而GRASS的r. grow distances函数仅能提供近似值。

我们使用EDT，让林地以500米的半径(50个单元）进行扩展。因此，我们选择所有数值小于500的单元格，并使用地图代数将它们存储到新的结果图中（见图3（1.2））。

为了减少计算量，我们将计算区域(ESRI称之为感兴趣区域AOI）缩小到生长林地。图3（1.3）显示，需要计算的单元数量显著减少。

AML的ZonalMean命令用于计算包含多个区域的地图中各区域单元格的平均值。我们以生长林地作为唯一区域，并使用两个坐标图（一个为x坐标图，另一个为y坐标图）作为数值图。通过这三个地图进行两次ZonalMean运算，结果如图3（1.4和1.5）所示。GRASS模块的r平均值功能也得出了相同结果。这两个结果共同描绘了生长林地的中心位置，而所有生长林地的中心点则来自第二步运算的结果。

3.2. 中间文件的缓存

GIS应用程序产生的中间数据以图像形式存储在硬盘上。由于读写、打包/解包、转换（例如坐标系）等原因，这会占用相当多的计算时间。实际上，在开始通过为GPU重写它们来加速单个操作之前，应该首先优化处理这些中间结果的方式。

当然，不可能将整个批处理脚本的所有光栅都保留在主内存中。因此，我们开发了一个内存管理的运行时环境。该环境基于最近最少使用（LRU）策略，决定哪些变量（因此

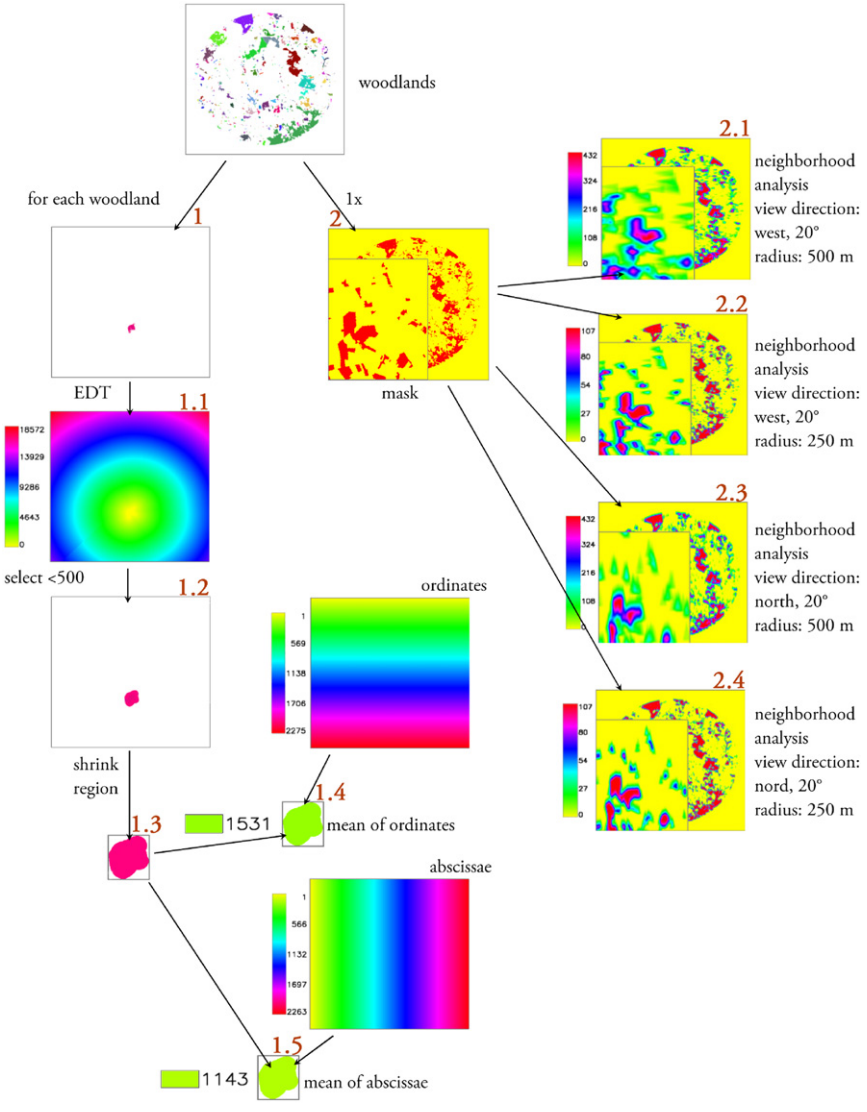

图3.示例数据流。

（栅格）被保存在主内存中，然后交换到硬盘。长时间未使用的变量会被交换出去。LRU算法确保最后一次操作的结果始终保留在主内存中，并准备好作为下次操作的输入。这一点对于循环尤其重要。

在使用GPU进行计算时，内存层次结构中还存在另一个关键阶段。我们可以采用相同的LRU算法，尽可能将数据保留在GPU内存中，仅在需要时才切换到主内存。表1和表2分别列出了不同光栅尺寸下的数据传输时间。主内存与GPU内存之间的传输速度，比硬盘与主内存之间的传输速度快约60倍。这一对比结果表明，必须避免频繁访问硬盘。

根据Aho等人（2006年）的研究，我们开发了一个脚本编译器，用于生成内存管理运行时环境的代码。该编译器能够识别BASIC语法。清单1展示了我们用BASIC语法编写的示例数据流。

清单2-用于添加两个栅格的内核代码。

```
//求和=左加数+右加数
__内核空隙的增加
(__仅写入图像2D总和，
__只读图像2D左附加项，
__只读图像2D右侧附加项，
    int2左加数偏移量，
    int2右加数偏移量)
{
//通过线程ID选择栅格单元
int2 current_cell =（int2）（get_global_id（0U），get_
                                global_id（1U））；

//我们使用块大小为16×16
//一个多处理器计算16×16个单元
//如果输出栅格的高度和宽度不
// an integer multiple of 16,
//必须检查栅格边界
if(currentcell.x＜getimagewidth(sum)&&
  currentcell.y＜getimageheight(sum))
{
  //从输入光栅中读取单元值
  浮动左单元格值=
    read_image_f（左加数，采样器，
    currentCell左添加偏移量(x)；
  浮动右侧单元格值=
    read_image_f（右加数、采样器、当前单元格+右加
      数偏移量）.x；
  //计算结果单元格值
  float4结果= 0. 0f；
  结果.x =左单元格值+右单元格值；
  //将结果写入输出栅格
  write_image_f（总和，当前单元格，结果}）
}
```

清单3-计算非空单元格周围边界框的内核代码。

```
//在用零初始化曲线后，这些曲线的计算方式如下：
__内核空配置文件
(__只写入图像2D文件中的配置项x，
__write_only image2d_t profile_y，
__只读图像2D输入光栅
)
}
//通过线程ID选择栅格单元
int2 单 元 =（int2）（get_global_id（0U），
                    get_global_id（1
                    U））；
//我们采用16×16的块尺寸//一个多处理器可处理16×
16个单元格//如果输出光栅的高度和宽度不是16的整数
倍，
//必须检查栅格边界，如果（cell.x<getimagewid-
th（inputraster））&&
  cell.y＜getimageheight(inputraster))
{
  //从输入栅格读取当前值float4 current_value = re-
  ad_imagef（input_raster，采样器，cell）；
  //如果当前值不是空单元格，//则配置文件将被填
充
  如果（！！isan（current_value.s0））
  {
    int4 one = 1；
  pos_x = (int2)(cell.x, 0)
  pos_y = (int2)(cell.y, 0)
    write_image_i（profile_x，pos_x，one)；
    write_image_i（轮廓Y，泊松Y，一）；
  }
}
}
//计算边界框__kernel void bounding_
box（__read_only image2d_t profile_
x，__read_only image2d_t profile_y，
__全球      int  顶部、
__全球      int  左、
__全球      int  右，
__全球）    int  底部的
{
int thread_id = get_global_id(0U);
int i；
int4值；
//如果（thread_id == 0），第一个线程搜索左边界
{
  i=0;
  value=read_imagei（profile_x，
  sampler，（int2）（i，0））；
  当(value. s0 == 0且i<getimage-
  width（profilex））时
    i++;
    value=read_imagei（profile_x，
  sampler，（int2）（i，0））；
  }
  左=i；
//如果（thread_id == 1），第二个线程搜索右边界
{
  // …
}
//第三、第四线程搜索顶部
//以及profile_y中的底部边界//...
}
```

3.3. GPU处理

现代高性能消费级显卡（如英伟达GeForce 580或ATI Radeon HD 6870）最多可提供500个计算单元，这意味着它们能同时处理多达500个光栅单元。相比之下，现代CPU在处理多媒体指令时，单次运算能力仅约30个单元。这些GPU的每个计算单元都并行执行同一算法模块——称为内核。程序员只需定义需要计算的算法实例数量，这些实例被称为线程，每个线程都有独立标识符。算法通过这个标识符来识别输入输出参数的地址空间。我们采用二维线程标识符来定位光栅操作的输入输出单元。

3.3.1. 光栅代数

栅格代数是执行加、减、乘、除和比较等标准数学运算的局部操作，输出栅格单元的值仅取决于同一位置的输入栅格单元。一个简单的例子是向栅格的每个单元添加一个常数值。另一个可能性是对比两个光栅。

表1
硬盘与主存之间的数据传输时间。

地图尺寸	从硬盘读取(s)	写入硬盘(s)
1138 × 1132	0.15	0.33
2275 × 2263	0.32	0.80
4550 × 4526	0.95	2.50

表2
GPU内存和主内存之间的数据传输时间。

地图尺寸	从GPU阅读内存（毫秒）	写入到GPU内存（毫秒）
1138 × 1132	4.4	3
2275 × 2263	8.2	9
4550 × 4526	34.0	33

设Mx，y和Nx，y分别表示两个光栅M和N中第x列第y行的单元格。结果R = M < N的计算方法如下：$R_{x,y} = \begin{cases} 1 & \text{if } M_{x,y} < N_{x,y}, \\ 0 & \text{otherwise.} \end{cases}$

GPU算法如下所示

$$Result_{ID_x,ID_y} = M_{ID_x,ID_y} < N_{ID_x,ID_y},$$

其中，ID是一个二维线程ID。行和列的循环由线程数隐式给出，并由GPU调度程序执行。

输入光栅与输出光栅之间，或多个输入光栅之间可能存在偏移。通常情况下，我们会为每个输入光栅定义一个偏移量（Δx，Δy）。此时，GPU算法将变为

结果ID_x，ID_y = MID_x þΔx_M，ID_y þΔy_M <NID_x þΔx_N，ID_y þΔy_N。列表2展示了用于添加两个光栅的GPU代码。

3.3.2. 通过卷积得到的FocalSum

FocalSum分析每个细胞的邻域。我们需要为每个单元格计算相邻单元格的值之和。因此，首先要确定楔形区域内的邻近单元格。我们使用矩阵来定义这个楔形区域，该矩阵可通过图4所示的参数进行计算。GPU通过检查以下条件，可以并行计算卷积矩阵的每个元素：

$$x^2 + y^2 \le r^2,$$

$$\varphi_1 \le arctan\frac{y}{x} \le \varphi_2,$$

其中x和y由二维线程ID给出。卷积矩阵的每个元素都可以并行计算。

输入光栅M与卷积矩阵W的卷积运算通过以下方式计算：

$$Result_{s,t} = \sum_{y=1}^{W_{rows}} \sum_{x=1}^{W_{columns}} W_{x,y} * M_{s-x,t-y}.$$

GPU能够并行计算这个方程。每个单元格由单个线程独立处理。由于相邻区域存在重叠，不同线程会访问相同的光栅单元格。GPU能缓存被多次访问的单元格，在我们的实现中使用纹理缓冲区作为缓存。GPU自主管理这个缓存机制：若缓存中没有预期值，就会从全局内存加载该数值。如果

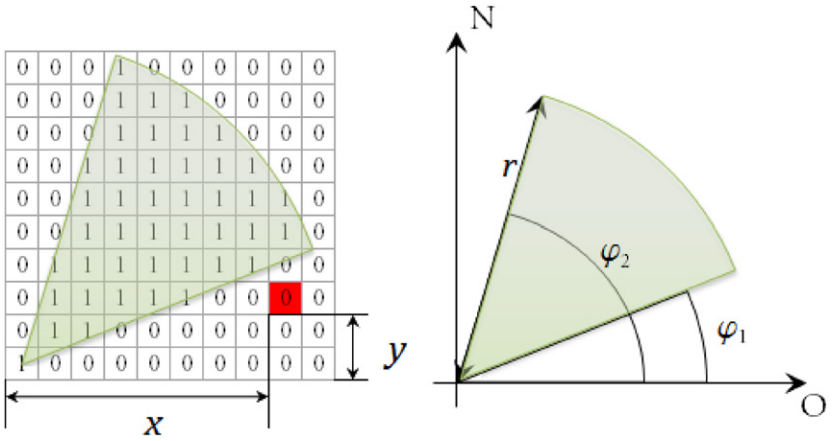

图4.用矩阵和参数描述的楔形。

该缓存包含一个预期值，该值从纹理缓存中读取的速度与访问寄存器或本地内存相当。当使用OpenCL纹理函数read_imagei和write_imagef读取输入数据时，GPU会填满这个缓存。纹理函数的用法如清单2和3所示。

3.3.3. 缩小计算区域

为了降低计算工作量，缩小计算区域是有用的。因此，我们需要找到栅格中第一行和最后一列中各自有非空单元格的行。挑战在于仅用数百个线程计算四个值，每个线程执行相同的算法。

我们使用行轮廓和列轮廓。轮廓是向量。行轮廓的元素数为R=行计数，列轮廓的元素数为C=列计数。R+C线程用零初始化向量。

我们使用R*C线程来检查单元格是否为空。如果单元格（x，y）不为空，我们会在行轮廓的x位置和列轮廓的y位置各放置一个1。图5展示了一个示例。清单3展示了GPU代码。

现在我们可以使用四个线程并行地搜索结果。第一个线程从行配置文件的起始位置开始，搜索第一个1，而第二个线程从行配置文件的结束位置开始，向后搜索，同样也搜索第一个1。

结果是第一行和最后一行包含非空单元格。另外两个线程对列配置文件执行相同操作。

3.3.4. 欧氏距离变换

欧氏距离变换（EDT）用于计算每个单元格到相邻非空单元格的最短欧氏距离。我们采用曹等人（2010年）提出的并行带状算法，通过GPU进行EDT计算。

大多数EDT算法都基于行扫描和列扫描（更多详细信息请参见Cuisenaire，1999）。一般来说，两个行扫描

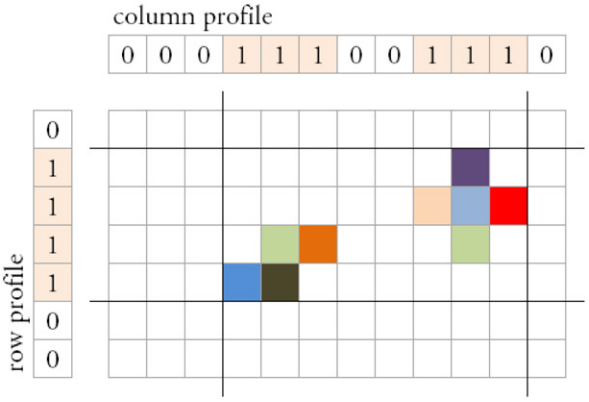

图5.示例栅格的行和列轮廓。

用于获取行中下一个非空单元格。图6展示了这种行扫描的结果。如果输入行至少包含一个非空单元格，输出行中的每个单元格都会包含该行中最接近的非空单元格的位置。

现在，一个列包含可能成为该列最近邻单元格的引用。这些单元格仅在行内是最近的。Cao等人。（2010）的研究采用Voronoi图技术，通过筛选方法剔除某一列中所有非最近邻单元格的引用。具体而言，系统会逐个比对三个连续单元格的引用关系。如图7所示，设a、b、c为列i中被引用的非空单元格。将a和b的垂线平分线与列i的交点记为p（i，u），b和c的垂线平分线交点记为q（i，v）。若u > v，则说明b不是列i中任何单元格的最近邻，可将其从该列移除引用。该比对过程通过自上而下的逐列扫描完成，且各列计算可并行进行。

在第三步中，通过最后一列扫描来计算距离。最终结果的列从上至下填充时，会通过比较该列中连续引用单元格之间的距离来完成。设单元格a为第i列中的第一个引用单元格，c为紧随其后的单元格，k为EDT结果中实际计算出的单元格。当a与k之间的距离小于c与k之间的距离时，a即为k的最近邻单元格，并将a与k之间的距离值存储起来。当沿列向下迭代时，c与k之间的距离会小于a与k之间的距离。在这种情况下，c将切换到a先前的角色，而c之后的下一个引用将切换到c之前的角色。列扫描后，结果包含欧氏距离转换。

为了利用GPU固有的并行特性，曹等人将每一行/列划分为若干B个频带，每个频带都

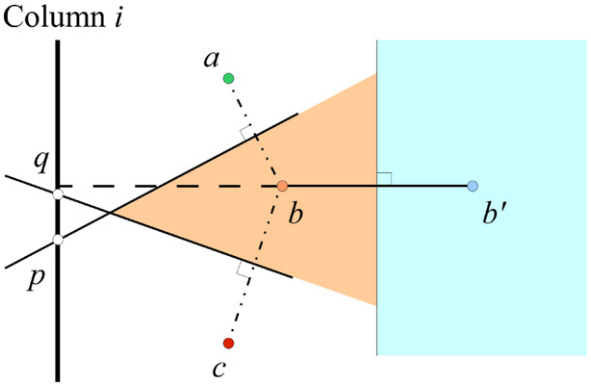

图7.用于计算EDT的Voronoi图属性示意图（Cao等人，2010）。

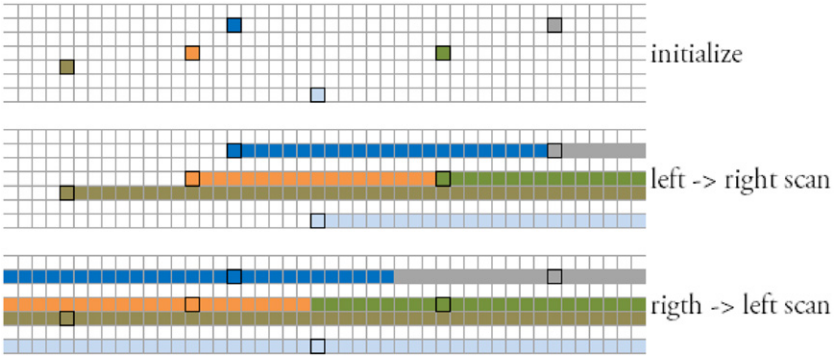

图6.行扫描过程：颜色代表行中最近的非空单元格。

通过扫描进行分析。使用B*R或B*C线程对整个光栅进行完整扫描，其中R表示行数，C表示列数。这些扫描结果需要在波段角上传播。完整的并行化方法详见曹等人（2010年）的研究。

3.3.5. 区域内的业务

区域由栅格中具有相同数值的单元格共同构成。区域运算需要使用带有区域划分的栅格数据，并对每个区域进行分析。本文重点介绍ArcInfo软件中的“区域平均值”功能。该功能接收两个输入栅格：一个是包含区域划分的栅格，另一个是数值数据栅格。系统会计算区域内数值的平均值，并将结果存储在该区域所属的所有输出单元格中。需要注意的是，由于GPU执行内核时缺乏动态内存分配机制，因此仅支持固定数量的区域划分。

3.3.6. 面罩平均值

在我们的示例数据流中，我们只需要一个分析单个区域的操作。因此，我们创建了一个名为MaskMean的操作。包含掩码的栅格定义了一个区域，每个非零值标记属于该区域的单元格。Mask Mean可以获得一个带有掩码的栅格和一个带有值的栅格，并计算出与ZonalMean对一个区域所计算出的结果相同的结果。

我们需要统计掩膜内的单元格数量（count）并求和其数值（sum）。这里再次面临一个问题：如何在多线程环境中仅计算少量数值（仅数量和总和）。我们采用迭代法解决这个问题。每次迭代中，每个线程会分析四个单元格。每次迭代的结果包含两个栅格数据：一个存储数值总和的栅格，另一个记录单元格数量的栅格。这些结果的大小仅为输入数据的四分之一。图8展示了该算法的迭代过程。第一次迭代的输出直接作为下一次迭代的输入。我们重复执行此算法，直到生成的栅格数据仅包含单个单元格。

平均值是总和与数量的比值。它必须存储在掩码范围内的每个结果单元格中。因此，我们创建行数乘以列数的线程。每个线程填充一个单元格。如果单元格位于掩码范围内，则填充平均值；若不在掩码范围内，则标记为空单元格。

3.4. GRASS插件

GRASS的每个栅格操作都是通过模块实现的。当应用一个模块时，就会启动一个新的进程执行模块的代码。除非通过硬盘上的文件，否则无法在内存中直接从一个模块调用向另一个模块调用传输数据。

为了解决这个问题，我们为GRASS开发了一个GPU模块，该模块执行一个调用加速GPU操作的BASIC脚本（示例见清单1）。使用前文所述的LRU算法（第3.2节），可以按需交换光栅。

我们还提供了一个函数，允许用户在BASIC脚本中运行标准的GRASS模块。这使得GPU加速模块更加多功能。当然，这些模块的输入和输出光栅必须存储在硬盘上。

4. 结果

我们对单个光栅操作的性能以及整个示例数据流所需的时间进行了测量，测试对象包括GPU和CPU。用于测量的测试机器配备了AMD Phenom II X4 940 CPU和GeForce GTX 260 GPU，内存为896 MB。

4.1. 计算

在本小节中，我们只关注每个特定栅格操作的计算时间。我们并不关心硬盘与内存之间的数据传输时间，因为缓存策略确保栅格通常已存储在GPU内存中。所有测量均限于计算过程。

4.1.1. 光栅代数

表3对比了添加两个栅格的计算时间。这是一个简单的操作，即使处理大型栅格，CPU也能在不到半秒内完成。表3显示，我们的GPU执行此操作的速度比CPU快40倍。所有加速比都表明了这一点。

表3
栅格代数运算时间（R = A+B）。

地图尺寸	CPU（毫秒）	GPU（毫秒）	加速
1138 × 1132	11	0.3	39
2275 × 2263	47	1	46
4550 × 4526	175	4	44

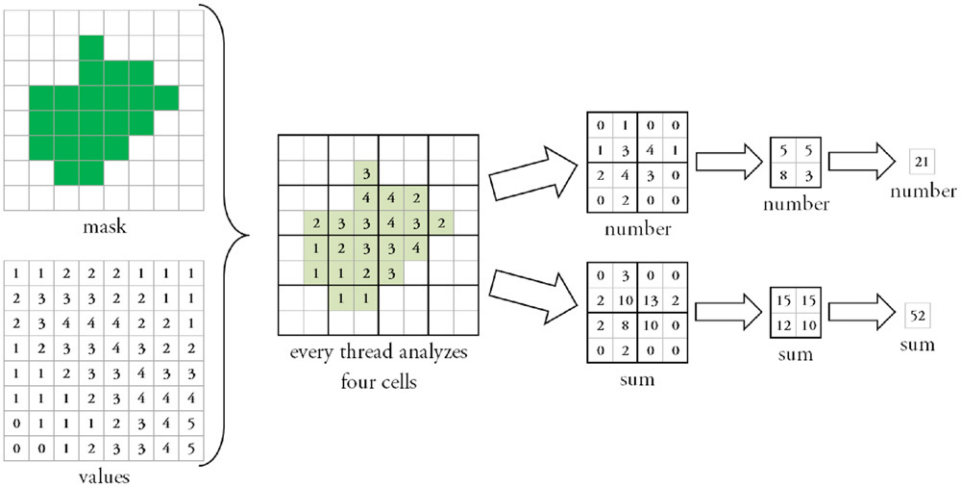

图8.隐藏值单元的汇总与计数。每个箭头代表一次迭代。

基本的运算和比较类似于
提出了一项。

4.1.2. 通过卷积进行的邻域分析

我们选择“楔形”邻域是因为ArcGIS在本次分析中出现了意外的长时间运行。ESRI地理信息系统在处理$\varphi_1 = 80$、$\varphi_2 = 100$且半径为50个单元格的区域时（参见第3.3.2节），需要约20分钟处理一个2275×2263像素的栅格数据。造成这种长时间运行的主要原因可能是ESRI系统对每个栅格单元重复计算卷积矩阵，而这种操作其实并非必要。

GRASS模块的r.mfilter函数无法处理如此庞大的卷积矩阵。为此我们开发了专用的GRASS模块来执行该运算，并将其与GPU实现进行对比测试。表4展示了当楔形参数设置为$\varphi_1 = 80$、$\varphi_2 = 125$且半径为50个单元格（卷积矩阵尺寸：38×50）时的计算时间。结果显示，我们的GPU实现速度比CPU版本快约50倍。

4.1.3. 缩小计算区域

为了缩小计算区域，最小的边界
需要找到包含所有非空单元格的方框。CPU和GPU都能快速完成这项任务。表5展示了结果。GPU的速度大约是CPU的10倍。

4.1.4. 欧氏距离变换

曹等人提出的PBA算法能计算精确的欧氏距离变换，而GRASS模块的r. grow distancia函数仅提供近似值。这两种算法存在本质差异，因此表6中的对比不仅涉及CPU与GPU的性能比，更包含两种算法本身的性能对比。虽然PBA算法同样适用于CPU架构，但GPU在距离变换运算上的速度仍比CPU快约40倍。

4.1.5. 特定区域的平均值

由于迭代过程的存在，我们的GPU算法复杂度高于CPU算法。这种复杂度差异直接影响了加速效果。表7展示了CPU和GPU的计算时间对比。

表4
使用38×50矩阵进行卷积的计算时间。

地图尺寸	CPU（秒）	GPU（秒）	加速
1138 × 1132	4.0	0.08	47
2275 × 2263	16.6	0.32	52
4550 × 4526	63.4	1.32	48

表5
寻找最小边界框的搜索时间。

地图尺寸	CPU（毫秒）	GPU（毫秒）	加速
1138 × 1132	3.1	0.6	5
2275 × 2263	11.7	1.2	10
4550 × 4526	44.7	3.2	14

表6
欧几里得距离转换的计算时间。

地图尺寸	CPU（毫秒）	GPU（毫秒）	加速
1138 × 1132	200	6	33
2275 × 2263	720	17	42
4550 × 4526	3080	58	52

实现。GPU版本的速度只有单线程CPU版本的15倍左右。

4.2. 示例数据流的批处理

在本小节中，我们将重点分析示例数据流的完整运行时表现。首先探讨避免硬盘数据传输的影响：虽然所有中间结果都保留在内存中，但未使用GPU进行计算。如表8所示，仅通过减少内存与硬盘之间的冗余数据传输，该示例数据流在CPU环境下运行速度提升了八倍。

在第二步中，我们利用GPU加速光栅运算。通常情况下，无法将所有中间结果都保留在GPU内存中。因此，我们测试了不同场景，具体数据如表9所示。执行示例数据流需要6个光栅运算，其中三个必须保留在GPU内存中（两个输入和一个输出）。若六个变量全部保留在GPU内存中，运行时间将比GRASS快150倍。若所有中间结果都存储在硬盘上，加速效果则降至1.2倍。光栅数据交换到主内存的时间成本并不高。运行时间保持在2分钟以下，相当于加速约40倍。

一般来说，不采用分块处理就无法计算出任何大型栅格数据。在GPU上执行栅格运算所需的最小内存容量，就是存储所有输入和输出栅格数据的总和。例如，在MaskMean运算中需要存储三个栅格数据：两个用于输入，一个用于输出。我们成功实现了分辨率为6100×6100像素单元的示例数据流计算。现代高性能GPU可提供高达6 GB的内存（例如NVIDIA GeForce GTX 590 3 GB、ATI HD 6990 4 GB、NVIDIA TESLA C2070 6 GB)。因此，计算栅格数据时最大内存容量可达1 GB。对于大多数应用场景而言，这个容量应该足够使用，但显然并非适用于所有场景。张等人（2010年）

表7
MaskMean的计算时间。

地图尺寸	CPU（毫秒）	GPU（毫秒）	加速
1138 × 1132	15	1.5	10
2275 × 2263	58	3.4	17
4550 × 4526	229	13.5	17

表8
使用GRASS对示例数据流进行运行时优化。这种“优化”运行时可避免在硬盘之间进行冗余的数据传输。

地图尺寸	草	优化（最小）	加速
1138 × 1132	21分钟	2.6	8.1
2275 × 2263	79分钟	10.3	7.7
4550 × 4526	5小时23分钟	41.1	7.9

表9
GPU在示例数据流上的运行时，其中中间结果具有不同的存储位置（光栅尺寸）。

正在启用的光栅数			示例数据流的运行时
GPU	RAM	硬盘	
6	0	0	32.5 s
5	1	0	57 s
4	2	0	1分36秒
3	2	1	2分20秒
3	1	2	4分9秒
硬盘上的所有中间结果			51分19秒

本文提出了一种基于GPU处理大规模栅格数据的方法。通过改进该方法，栅格的最大尺寸得以提升。但另一方面，部分计算算法需要针对这种数据管理方式进行调整。

5. 结论

基于一个示例数据流，我们分析了加速空间栅格数据批处理的可能性。我们利用GPU的计算能力来加速局部操作、邻域分析和距离测量操作。我们还尝试加速区域操作，但未能找到一种通用的方法。

我们使用GRASS作为并行化操作的开发平台，并将其作为示例数据流运行时的参考。GRASS需要79分钟来完成整个数据流。

我们改进了地理信息系统（GIS）处理批处理中间结果的方式。这类结果会尽可能长时间地驻留在内存中（ArcGIS提供的内存工作区机制与此类似）。在Linux系统中，另一种选择是让GRASS将中间文件存储在内存磁盘上。若没有这些缓存工具，使用GPU进行并行计算就缺乏实际意义。若仅加速计算过程，我们的示例数据流总耗时将比GRASS未采用并行化处理时快1.2倍(51分钟）。内存管理方式的改变为用户带来显著优势。若将示例数据流的所有中间结果保留在内存中，其运行速度可提升八倍——而GRASS在未使用并行化时仅需10分钟。因此用户需要区分需要存储到硬盘的结果与仅作为中间结果的数据流：中间结果在数据流完成后将不再可用。

如果我们也使用GPU来加速计算，可以达到150倍的加速。整个数据流的运行时间从79分钟下降到33秒。

致谢

我们衷心感谢审稿人对我们工作的关注及提出的宝贵意见。特别感谢来自Rainer Schulz博士的...

布伦瑞克大学格奥尔格-奥古斯特学院的生态信息学、生物统计学与森林生长系ö感谢Tinggen公司为我们提供了示例数据流和栅格地图。同时，我们也要感谢Max Daenner在初期启发性讨论中给予的宝贵意见。

参考文献

Aho，A.V.，Lam，M.S.，Sethi，R.，Ullman，J.D.，2006年。编译器：原理、技术与工具，第二版。Addison Wesley August.

贝特尔，A.，莫尔霍夫，T.，阿加瓦尔，P.K.，2010.基于自然邻域插值的网格数字高程模型构建方法（GPU实现）。收录于：第18届SIGSPATIAL国际地理信息系统进展会议论文集（GIS‘10），美国纽约ACM出版社，第172–181页。

曹天台、唐科、穆罕默德·A、谭天生，2010.基于GPU的并行带状算法实现精确距离变换。收录于：《2010年ACM SIGGRAPH交互式三维图形与游戏研讨会论文集》（I3D‘10）。美国纽约ACM出版社，第83–90页。http://doi.acm.org/10.1145/1730804. 1730818.

Cuisenaire，O.，1999年。距离变换：快速算法及其在医学图像处理中的应用。

Neteler，M.和Mitasova，H.，2008年，《开源地理信息系统：GRASS GIS方法》，第3版，Springer出版社。

OpenCL，2010。OpenCL规范：版本1.1。Khronos Group，文档修订版：36。

奥尔特加，L.，鲁埃达，A.，2010年。基于CUDA的并行排水网络计算。《计算机与地球科学》第36卷第2期，171-178页http://www.sciencedirect.com/ science/article/pii/S0098300409002970。

泰米泽尔，A。哈利奇（Halici）、洛戈格鲁（Logoglu）、特米泽尔（Temizel）、奥姆鲁宗（Omruuzun）与卡拉曼（Karaman），2011年，《基于CUDA和OpenCL的图像与视频处理经验》，载于《GPU计算宝石》（摩根·考夫曼出版社，波士顿，第547-567页）（章节编号34)/http://www.sciencedirect.com/science/article/pii/B9780123849885000346S）。

维奥拉，I.、卡尼察尔，A.、格罗勒，M.E.，2003年10月。基于硬件的非线性滤波与分割技术——运用高级着色语言。收录于：图尔克，G.、范·维克，J.、穆尔黑德，K.（编），《IEEE可视化会议论文集2003》。IEEE出版社，第309-316页。

沃尔什，S.D.、萨尔，M.O.、贝利，P.、莉尔雅，D.J.，2009年。加速图形硬件上的地球科学与工程系统仿真。《计算机与地球科学》第35卷第12期，2353–2364页。/ http://www.sciencedirect.com/science/article/ pii/S009830040900190 3S。

吴毅、葛宇、严伟、李翔，2007.基于GPU的GIS空间栅格分析性能优化。收录于龚鹏、刘勇主编的《SPIE会议论文集》（第X卷）。6754.SPIE，pp.67540P1-67540P11。

杨志、朱宇、蒲野，2008.基于CUDA的并行图像处理。载于《2008年国际计算机科学与软件工程会议论文集》（CSSE‘08）第3卷。IEEE计算机学会，华盛顿特区，美国，第198-201页。

张军、尤思、格伦瓦尔德·L，2010.大规模栅格地理空间数据的索引构建——基于大规模并行GPGPU计算。收录于：第18届SIGSPATIAL国际地理信息系统进展会议论文集（GIS‘10）。美国纽约ACM出版社，第450–453页。