arXiv:2303.10581v1 [cs.DC] 19 Mar 2023

亮点

用于加速二维凸包的 GPU 过滤器评估

Roberto Carrasco，Héctor Ferrada，Cristóbal A. Navarro，Nancy Hitschfeld

- 一项关于加速二维凸包的过滤方法的实验研究。
- 评估了四种筛选方案：一个 CUDA 内核、两个 Thrust 函数和一个 CUB 原语。
- 所有方法均与作为标准参照的 CGAL 库进行了比较。
- 测试了三种点分布：正态分布、圆周和偏移圆周。
- 源代码和基准测试已向社区开放。

用于加速二维凸包的 GPU 过滤器评估

Roberto Carrasco[a]，HéctorFerrada[b]，CristóbalA.Navarro[b] 和Nancy Hitschfeld[a]

[a]智利大学计算机科学系。[b]智利南方大学信息学研究所

文章信息

关键词：GPU计算、计算几何、凸包、筛选技术、并行归约

摘 要

凸包算法是计算几何中最重要的算法之一，在计算机图形学、机器人学和数据挖掘等领域有着广泛应用。尽管该领域的新算法不断发展，但为了快速解决更大规模的问题或实现实时处理，通常仍需进一步提升性能。本文对 GPU 筛选器进行了实验评估，以降低二维凸包的计算开销。该技术首先对输入点集进行预处理，在对数时间内筛除位于一个八顶点多边形内部的所有点，从而获得规模更小的候选点集。我们采用并行计算，并以曼哈顿距离为度量来确定多边形的顶点并筛选点。对于过滤阶段，我们研究了不同方法，包括自定义 CUDA 内核以及 Thrust 和 CUB 等库。测试了三种点分布：正态分布（有利情况）、圆周分布（最差情况），以及点从圆周随机偏移的分布（中间情况）。实验评估表明，GPU 筛选算法的速度最高可达串行 CPU 实现的 23× 倍，而整体凸包计算速度最高可达 CGAL 库所提供的最快实现的 30× 倍。

1. 引言

凸包是一种几何概念，指平面上能够包围给定点集的最小凸多边形 [1]。通俗地说，它是连接点集中最外围所有点后形成的形状。凸包的研究历史悠久，多年来人们开发了许多不同的算法和技术，以提高凸包计算的效率和准确性 [2]。

凸包在 科学技术的各个领域有着广泛的应用。例如： :

- 计算机图形学：凸包用于计算形状的凸包围多边形，以加速对其进行的几何运算，并构建德劳内三角剖分和沃罗诺伊图等其他结构[1, 3],。
- 碰撞：机器人学利用凸包计算机器人的可达空间，即机器人在不与障碍物发生碰撞的情况下能够移动到的点集 [4, 5]。
- 数据挖掘：利用凸包根据一组点之间的空间关系将它们聚类成若干组。这有助于从大型数据集中发现模式或趋势 [6]。

总的来说，凸包算法在许多不同领域有着广泛的应用，是解决各种问题的宝贵工具。多年来，凸包算法取得了显著进展，并发展出多种算法，例如礼品包装算法 [7, 8] in $O(nh)$ 时间（其中 n 是原始点集中点的数量， h 是凸包上的点数）、Graham扫描算法 [9] in $O(n \log n)$、QuickHull在最坏情况下的时间复杂度为 [10]in $O(n^2)$，平均情况下为 $O(n \log n)$ in、分治算法 [11] 以及增量法[12] in $O(n \log n)$等。此外，人们还开发了能够处理特殊情况的新算法，例如计算球面上点的凸包，或计算高维空间中的凸包。总的来说，凸包计算的最新进展仍在不断演进和改进，为各种问题提供愈加高效、准确的解决方案。Qhull [10], 和CGAL [13] 库提供了两种最高效的凸包实现，它们实现了上述一种或多种算法。值得注意的是，在本研究中，CGAL被视为与未来解决方案进行比较的标准参考。QHull

ORCID(s):

和 CGAL 库都针对顺序计算方案进行了优化。近年来，计算几何领域的许多应用已从基于中央处理器的顺序单核计算转向并行计算 [14]，以便处理更大规模的问题，同时避免执行时间过长。最常见的转变之一是从中央处理器计算转向基于图形处理器的计算。

本研究旨在探讨现代 GPU 所采用的各种编程方法，这些 GPU 配备了新型核心和用于提升效率的数据管理库。我们针对所提出的算法实现了四种不同变体，并进行了全面的性能评估，以确定这些尖端技术对性能的影响。Ferrada 等人 [15], 最近对凸包算法进行了顺序优化。在本研究中，我们利用现代 GPU 的并行处理能力，提出一种加速凸包算法预处理阶段的方法，剔除八边形内部的所有点。我们还展示了使用 GPU 编程模型实现的四种变体的实验结果。本研究的所有实现都特别关注以下关键思路：

- 快速预处理：本研究的主要成果是评估了四种筛选变体，用于计算一组点的凸包。
- 高效操作：基于 [15], 的工作，我们的另一项关键贡献是使用曼哈顿距离作为度量来确定初始多边形，并采用并行技术进行预处理。
- 并行算法：我们贡献的另一个关键方面是，在实现中采用最新的图形处理器编程模型 [16,17, 18]。
- 可扩展性：本研究提出了一种可扩展性更强的内核实现，能够处理比当前最先进的库所支持的数据集更大规模的数据集。

本文其余部分安排如下：第2节介绍相关工作。第3节阐述问题陈述和主要贡献，第4节介绍该算法的不同实现变体。第5节介绍所研究的点分布。第6节对照 CGAL 中可用的一种更快实现进行深入的实验性能比较。最后，第7节讨论实验结果并给出结论。

2. 相关工作

以往为加快凸包计算而开展的工作，采用并行算法实现传统算法中的部分操作，例如计算点之间的距离或确定极值点等；也采用并行预处理技术来筛选候选点。Srungarapu 等人提出了一种基于图形处理器的并行 QuickHull 算法，以加速二维凸包的计算。他们给出了一种基于 QuickHull 的算法，并行化了极值点的确定、标记多边形内部的点以及扫描操作，但 QuickHull 的主循环仍在中央处理器上执行。他们报告称，相较于传统的基于中央处理器的凸包求解方案，加速比最高可达 14× [19]。

Mei 提出了一种基于图形处理器的解决方案 [20]，利用预处理方法对所有点进行分类，并在图形处理器上剔除不属于凸包的点。这一预处理步骤使其相较于 Qhull 实现的加速比最高达到 6×。此外，Stein 等人介绍了一种使用 CUDA 编程模型计算一组点的三维凸包的并行算法 [21]。该方法基于 QuickHull 方法，相较于基于中央处理器的 Qhull 实现，取得了30×的加速比。

Blelloch 等人对一种用于计算二维凸包的并行增量随机算法进行了理论分析 [22]：对于任意固定维度下包含 n 个点的集合，该算法以高概率具有 $O(\log n)$ 依赖深度。这带来了一种简单且工作量最优、跨度为多对数级的并行算法。

Barbay 和 Ochoa 提出了一种自适应算法，用于合并平面上的 k 个凸包。该算法首先将输入点序列分解为若干部分，并分别计算每个部分的凸包，这两个步骤都可以在线性时间内完成。随后，他们采用一种新颖且快速的合并技术，将所有部分凸包合并起来。这两项工作都展现了利用图形处理器加速凸包算法的潜力。

提升计算性能最常用且高效的方法，是剔除不可能属于凸包的内部点。然而，许多算法的性能高度依赖输入规模，而预处理算法可以通过 $O(n)$ 丢弃与凸包算法无关的点 [23, 24]，显著缩小输入规模。Mei

提出了一种用于计算二维和三维凸包的筛选技术：先将图形处理器上的所有点分别旋转三个不同的角度，以此识别凸包上的 16 个点，然后并行丢弃由这些已识别点构成的多边形内部的所有点。此外，Mei等人还提出了在图形处理器上使用迭代筛选技术，以进一步提升凸包的计算效率 [25]。

在计算凸包的预处理技术方面，较新的一项研究是 Al-shamrani 等人提出的方法。 [26], 他们提出了一种筛选技术，通过筛选四个顶点构成的多边形内的所有点，利用欧几里得距离找出极值点；与Graham扫描算法和 Jarvis步进算法相比，该方法分别实现了最高达 77× 和 12× 的加速。另一项重要且较新的研究是 Ferrada等人提出的方法。 [15], 他们开发了一种名为 heaphull 的顺序方法，使用曼哈顿距离作为主要度量，剔除二维凸包中的点。该方法剔除由八个顶点构成的多边形之外的所有点（ $O(n)$ ），从而缩减候选点集合。他们报告称，与 CGAL 库中提供的凸包方法相比，该方法的加速比为 1.7× 至 10× 。在这项最新研究中，他们继续开展 GPU 实现方面的工作；Alan等人 [27] 报告称，与顺序的基于CPU的算法（heaphull）相比，该方法快 4× ，与当前最先进的其他 GPU 方法相比则快 $3 \sim 4\times$ 。

近期许多基于中央处理器的研究都有望采用 GPU 编程模型实现，例如 Ferrada 等人的工作 [15], 。通过利用图形处理器的并行处理能力，该方法有望显著提升性能。这为改进现有筛选方法提供了机会：引入 GPU 编程模型和低成本几何运算。

3. 凸包过滤并行算法

本节介绍一种图形处理器筛选技术，该技术基于 Ferrada 等人提出的基于中央处理器的工作[15] ，并采用曼哈顿距离作为计算极值点的主要度量。所提出的算法由三个阶段组成。第一阶段如（3.1）所述，从输入数据中构造一个由极值点组成的八边形。第二阶段如（3.2）所述，剔除所有不可能成为凸包候选点的点。最后阶段如（3.3）所述，使用现有的先进算法计算凸包。最后，我们展示该算法的复杂度。

3.1. 寻找八边形

算法的第一阶段是根据 n 输入点构造一个八边形，然后用它筛选这些点，并将剩余的 $n' \leq n$ 点保留为凸包候选点。图1展示了应用这一阶段后，对凸包候选点进行点分类的结果，其中显示了八边形、极值点和角点。该多边形由四个极值点（最右、最上、最左和最下的点）以及四个点组成；这四个点根据曼哈顿距离，分别是最接近包围盒各角的点（见图1）。该包围盒由四个极值点定义。每个角点都是在逆时针方向上，从每对相邻坐标轴极值点的投影所形成的角（例如，右上角由最右侧的 x坐标和最上方的 y坐标构成）出发，与各点之间曼哈顿距离最小的点。通过并行归约计算最小距离，可在对数时间内完成此操作。由这些点构成的多边形内部的所有点都可以确定不属于凸包。该算法需要 $O(\log n)$ 时间来找出所有这些点。

此阶段旨在以对数时间找到角点。对于距离函数， [28] 采用曼哈顿距离，它能找到到各个角点的垂直距离与水平距离之和最小的点。此外，曼哈顿距离的计算简单且开销低，只需进行加法和减法运算；而欧几里得距离则需要乘法以及平方根等计算开销较大的运算 [29]。图2 展示了找到圆周上所有坐标轴极值点（包括坐标轴上的极值点和角点）所花费的时间。从图中可以看出，无论在图形处理器还是中央处理器上，计算曼哈顿距离的开销都低于计算欧几里得距离。

3.2. 构建过滤器

算法的下一阶段涉及一次 GPU内核调用，每个点由一个独立线程处理，并检查它是否位于上一阶段构建的多边形内。如果点位于多边形内部，算法就会将其筛除，如图 1 所示，其中黑色点被视为凸包的候选点，而灰色

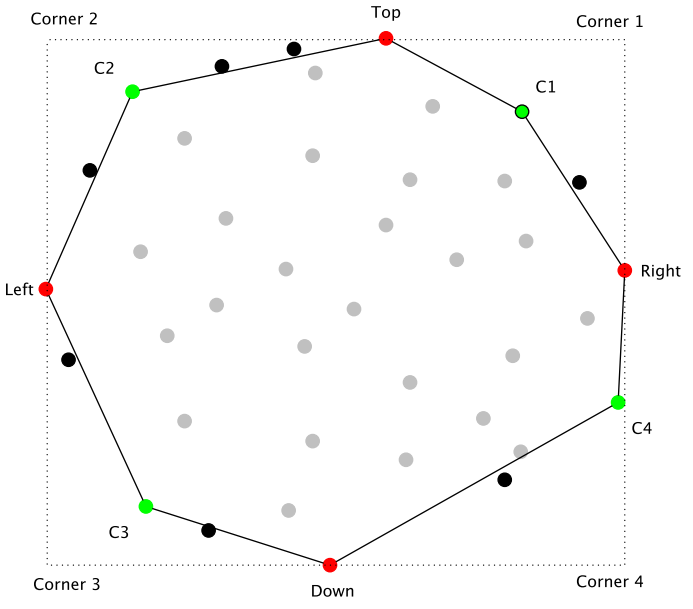


图1： 该图展示了将算法第一阶段应用于点云后得到的八边形。黑色点被视为凸包的候选点，灰色点则在过滤阶段被舍弃。各坐标轴上的极值点以红色表示，角点以绿色表示。

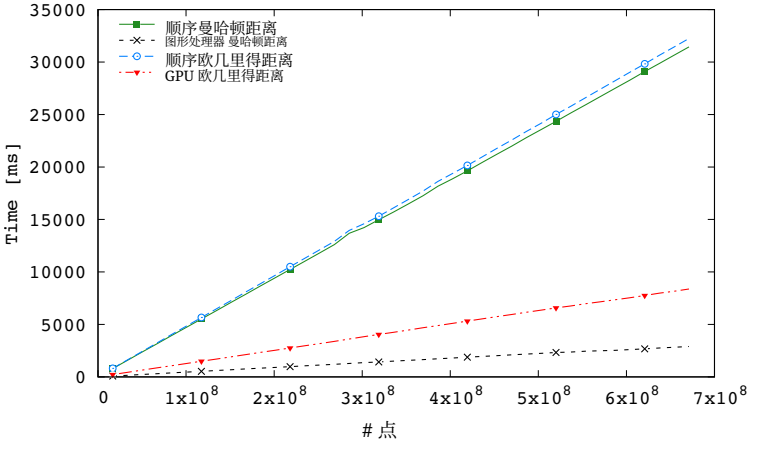


图2： 使用曼哈顿距离和欧几里得距离寻找极值点的性能比较。

点则被舍弃。经过这一阶段，我们得到一个包含 n 位的向量，其中每个值为 1 的位集都代表一个凸包候选点；如果其值为 0，则该点不属于凸包。

然而，由于事先无法确定原始数组中每个元素在压缩数组中的位置，因此无法通过一次并行内核调用同时压缩稀疏数组中的所有元素。不过，有一些技术可以在对数时间内压缩稀疏数组 [30]。将稀疏位数组压缩为较小的非稀疏数组，最常见的方法是使用并行扫描或原子函数；本工作

采用了专为我们开发的自定义压缩方法，该方法基于 Dakkak 等人开发的张量核心扫描 [16]，并针对整数数组进行了调整。

3.3. 计算凸包

作为上一阶段的结果，我们得到了一组 n' 来自初始输入点集的候选点（图1中的红色、绿色和黑色点；所有灰色点均在上一阶段被丢弃）。该算法可以与现有最先进的任意凸包实现结合使用。在本研究的所有实验中，我们使用了 CGAL 提供的一种高效的中央处理器实现，更多详情见第6节。

3.4. 算法复杂度

算法的复杂度由前两个阶段的复杂度及其执行的操作共同决定。从构造八边形开始，该阶段的开销取决于极值点的搜索：顺序执行时开销为线性，并行执行时为对数级。第二阶段对应筛选器的构造，其复杂度为对数级。这是因为，与开销为线性的顺序版本不同，该阶段无需遍历各点以判断其是否位于多边形内，而是直接将结果写入一个长度最小的数组中。并行执行时，该阶段包含两个子阶段。第一个子阶段判断哪些点是候选点，在 $O(1)$ 中只需调用一次图形处理器上的内核；第二个子阶段对应步骤中的数组压缩，其开销为对数级。因此，两个阶段的开销总和为 $O(\log n)$ 级，而对应的中央处理器方案仅提供线性时间的解法。

4. 筛选器实现

图形处理器上的编程有多种方式，涉及不同的编程方法，并采用新型核心和库来实现高效的数据管理。首先，可以使用 CUDA 提供的 C 语言低级内核进行编程。CUDA 还提供两个较高级别的应用程序接口（API）：第一个是 Thrust，它对 λ 函数提供强大支持；第二个是 CUB，它提供用于在图形处理器上处理数据的快速软件组件。本文采用不同的编程策略实现了四种筛选器变体，其中一种基于 GPU 内核 [31], 两种基于 Thrust，另一种基于 CUB。这些实现的代码可在以下网址获取：https://github.com/rcarrascoc/GPU-2D-Convex_凸包_筛选器。

我们为算法中的筛选定义了三个函数1。第一个函数对应于查找八边形阶段，记为 查找多边形，输入为一组点，输出为所有极值点和角点。第二个函数是 构建过滤器，它接收所有极值点，并标记哪些点是凸包的候选点。第三个函数根据筛选器得到的标志，在非稀疏矩阵中压缩输入数据，使该函数的输出仅包含凸包候选点。最后，使用当前最先进的任意算法，根据前一个函数的输出计算凸包。

算法 1筛选器

要求： 点集 S**确保：** 凸包候选点集 1:
$\{left, top, right, down, c1, c2, c3, c4\} \leftarrow$ 查找多边形(S) 2:
$bit_vector_flag \leftarrow$ 构建过滤器$(S, left, top, right, c1, c2, c3, c4)$ 3:
$filtered_set \leftarrow$压缩过滤后的点(bit_vector_flag) 4: $output \leftarrow$ 凸包算法$(filtered_set)$ 5: **返回** $output$

变体的命名规则如下：第一个词 textitThrust 或 *CUB* 表示寻找极值点的方法，分别是使用 GPU 内核中的并行归约，或通过 Thrust 或 CUB 函数实现。第二个词（扫描、复制或标记）指的是压缩过程中使该方法区别于其他方法的具体特性。本节分为三个小节，分别介绍各方法（名称中的第一个词）寻找极值点的方式；每个小节还包含相应的子小节，介绍根据第二个词所采用的压缩技术。这些技术不能互换，因此只有下文所述的组合方式可行。

4.1. 图形处理器实现 (内核)

此筛选器的基本思路是仅使用自定义图形处理器内核和图形处理器编程技术。该实现采用基于 CUDA shuffle 操作的并行归约技术，获取所有极值点（最左、最右、最高和最低点）[32, 33]，以找出各轴上的最小和最大点坐标。获得极值点后，再次使用并行归约计算角点，找出到包围盒各角点的最小曼哈顿距离。

4.1.1. 变体 *#1*： 所提过滤器

为了找出凸包的候选点，我们使用 GPU 内核检查这些点位于由极值点和角点构成的八边形内部还是外部。检查结果是一个大小为 n 的位向量（与输入点数量相同），其中标记为 1 的位表示该点是凸包的候选点，标记为 0 的位表示该点已从凸包计算中剔除。

现在，我们有了一个稀疏向量，用于标识某个点是否为凸包的候选点。然而，必须将凸包的输入缩减为点的精简列表。由于并行计算中的同步全局内存访问并非易事，因此有一些解决方案需要对数时间。图 3 展示了一种策略：复制所有标记为 1 的元素，并丢弃标记为 0 的元素。按照这一思路，每个标记为 1 的元素相对于第一个点的位置由累加和（扫描）[34] 给出。该方法需要三个步骤：筛选、扫描和散射，耗时为对数时间。

- 第一遍会生成一个临时向量，其中通过筛选器的元素被视为凸包候选点，并设为 1；其他元素则设为 0，作为扫描输入。
- 第二步对上一步生成的位向量执行部分和（扫描）。每个通过筛选器且属于凸包候选点的元素，都会在扫描累积结果中占据一个位置。
- 第三步将每个作为凸包候选点的点放入输出数组。

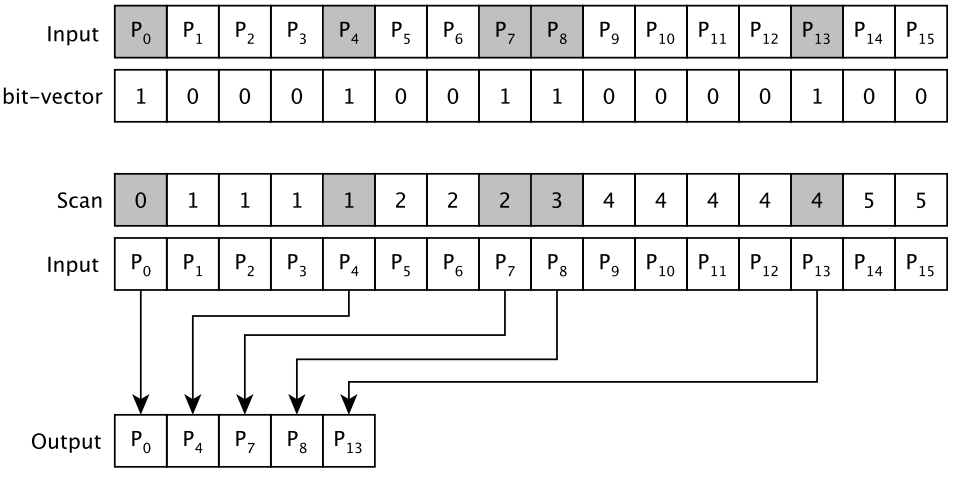


图 3： 图中展示了并行扫描 压缩的基本原理。我们要保留的输入点以灰色标记，并在每个灰色单元格中设为 1。随后计算这些 1 的扫描结果。

本工作采用了利用现代图形处理器张量核心的扫描技术，该技术可分为三个主要步骤：

- 分段扫描：使用张量核心对上一步生成的临时向量执行分段扫描 [16]。此步骤会生成每个点相对于各分段的新位置。
- 全局扫描：使用每个分段的最高有效值，通过 CUDA 核心计算所有分段的扫描结果。此扫描会计算每个分段相对于全局位置的位置。

- 压缩：利用上一步生成的新位置，将作为凸包候选的点压缩到输出数组中。最终位置由 $segment[pos]+global[pos/num_segment]$ 给出。

这种技术效率很高，能够处理大型数组，与 Thrust 和 CUB 等其他库相比，可实现更快的压缩。

4.2. Thrust 实现

Thrust API 为在支持 CUDA 的图形处理器上进行并行编程提供了易于使用的高级接口。它提供了丰富的功能，例如扫描、排序、最小值-最大值归约、数据变换和数组压缩；这些功能已广泛用于此前在图形处理器上实现的凸包算法中。 [23, 24, 25]不过，需要注意的是，Thrust 可处理的向量最大长度为 2^{30} 个点，这可能会限制其在需要更大点集的场景中的使用。相比之下，核函数实现仅受可用 GPU 内存的限制，因此在输入规模方面具有更好的可扩展性。

Thrust API 提供 min_element 和 max_element 函数，分别返回列表中最小和最大元素的索引。要找到极值点，必须针对每个轴坐标执行最大值-最小值归约。要找到角点，必须计算每个点与各角点之间的曼哈顿距离，然后进行归约。最后，使用位向量，通过变换函数检查每个点是否属于凸包。此外，还可以利用 Thrust 提供的归约，采用两种策略压缩输出。

4.2.1. 变体 *#2*：*Thrust* 扫描

此变体采用与扫描变体（小节 4.1.1）相同的策略，但使用 Thrust 的 `exclusive_`扫描函数，根据过滤阶段生成的位向量获取部分和（扫描）。然而，必须将前缀和所得的向量转换为存储在图形处理器内存中的兼容数组。为此，需要在调用图形处理器内核时散射每个点的正确地址。该变体利用了 Thrust 提供的高效归约。不过，将 Thrust 向量转换为图形处理器数组会产生额外的计算开销；如下一变体所述，可以通过使用较慢的压缩操作来避免这一开销。

4.2.2. 变体 *#3*：*Thrust* 复制

Thrust 提供的 copy_if 函数可根据给定条件，将初始数组中的元素复制到一个更小的新数组中。在此情形下，条件由筛选器确定：若某个点被筛选器标记为 1，则保留；若标记为 0，则丢弃。copy_if 函数在筛选器丢弃大量点时表现良好，但当筛选器保留大量点时，计算开销可能会很高。

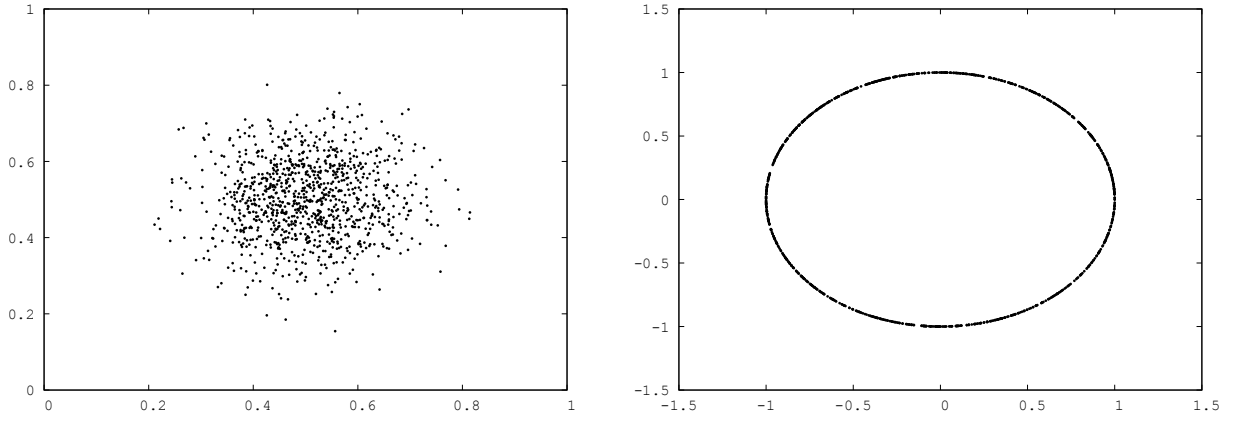
4.3. CUB 实现

CUB 为 CUDA 编程模型的各个层次提供先进且可复用的软件组件，其中包含三个层次的 CUB 原语：

- 在单个 warp 内运行的 warp 范围“集体”原语。
- 在一个线程块内的所有线程间运行的线程块范围“集体”原语。
- 在设备上的所有线程和线程块间运行的设备级原语。

前两个层次对应于内核内执行的操作，而最后一个层次对应于主机上的操作。

CUB 提供名为 ArgMax 和 ArgMin 的设备级原语函数，分别用于查找最大值和最小值，并以键值对的形式表示，其中键对应点的索引，值对应坐标值。CUB 还提供扫描函数，但在实验中，其性能并未优于其他已实现的变体。



(a) 正态分布。 (b) 圆周（最坏情况）。

图 4： 分布的图形表示，左侧的图形对应均值为 1、标准差为 0。1 的正态分布；旁边是一个以原点为中心、半径为 1 的圆周

4.3.1. 变体 *#4*： *CUB* 标记算法：

DeviceSelect 提供设备级并行操作，用于压缩存储在设备可访问内存中的数据序列，并从中选出所需项。这些操作会根据筛选条件，将指定输入序列中的项有选择地复制到紧凑输出序列中。就编程语言而言，它使用 d_flags 序列来确定将 d_in 中的哪些项复制到 d_out，并将选中项的总数写入 d_num_selected_out。这使得在设备上高效并行地压缩数据成为可能。

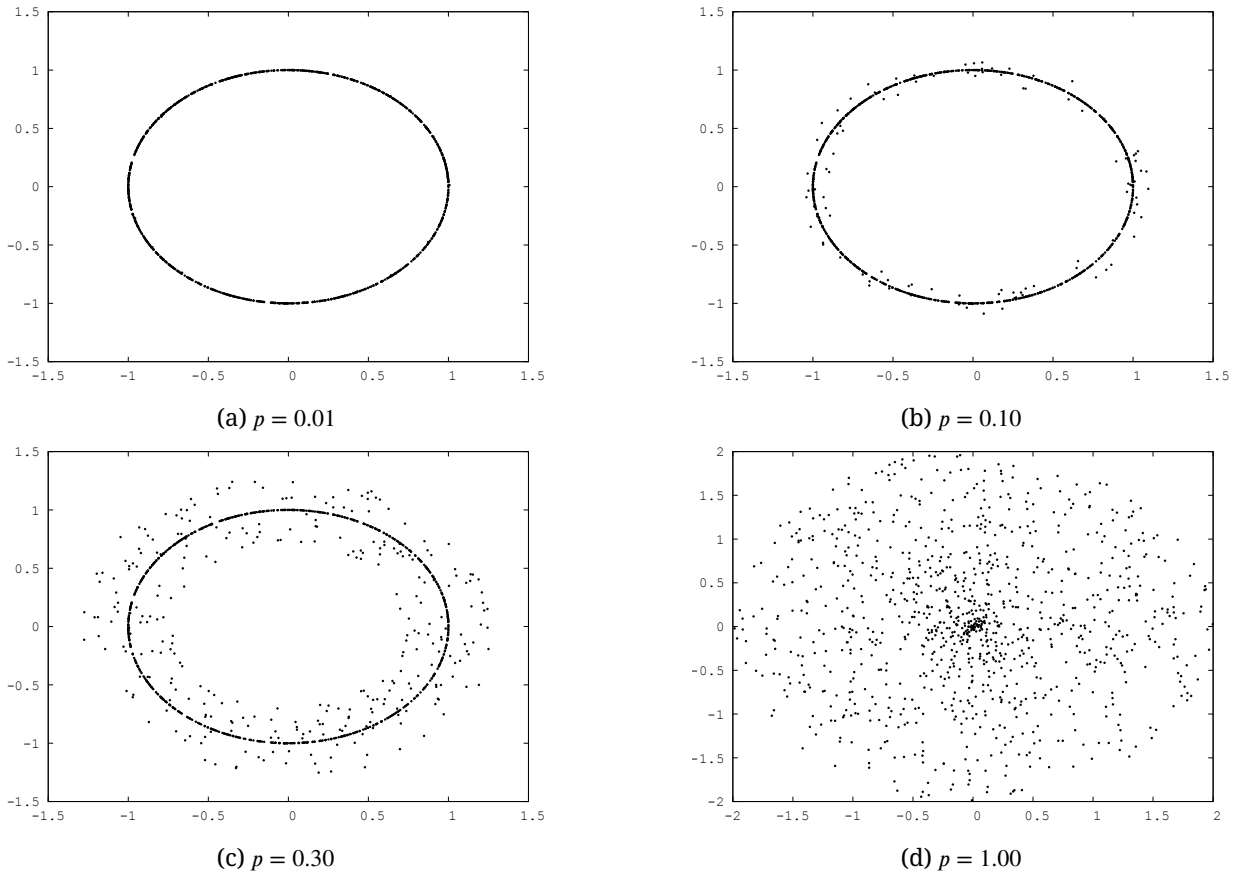
5. 研究案例

筛选出的点的数量在很大程度上取决于平面上的点分布：如果点集中在点云中心，筛选出的点就会比所有点都位于点云边缘时更多。本研究探讨了三种有趣的情形，以研究所提方法的表现：(1) 正态分布，(2) 完美圆周，以及 (3) 偏移圆周。

1.- **正态分布 - 常见情形：** 正态分布常见于许多问题中，例如物理现象、人类行为、科学及其他领域，因为这些事件通常自然呈现出与高斯分布相似的行为，如图 4a所示。此测试在平面上生成 n 个服从正态分布的随机点，其中 $\mu = 0.5$ 且 $\sigma^2 = 0.1$。一些输出敏感型凸包算法，例如 *QuickHull*，会利用这类情形的特点。

2.- **完美圆周 - 最坏情形：** 图 4b 展示了所有点都位于同一圆周上的情形，因此所有点都处于凸包上。由于没有点被筛选掉，这对所提算法而言是最坏情形。与正态分布情形不同，一些输出敏感型算法（例如 礼品包装算法）在此情形下效率低下。在这种情况下，建议使用一种能够保证在 $O(n \log n)$ 中具有良好性能的算法。

3.- **偏移圆周 - 中间情况：** 该测试允许点分布模型中属于凸包的点数较多或较少。在这里，我们要找出算法开始表现良好的临界点。测试通过在以原点为中心、半径为 $r = 0.25$ 的圆周上（或非常接近圆周处）生成 n 个点来构建。测试提供一个 p 参数供选择，该参数会为每个点生成一个介于 $[r - rp, r + rp]$ 范围内的偏移概率。此位移可能使点向内移至中心，或向外移至圆周，从而形成一圈围绕圆周的点带。图 5 展示了不同 p取值下的位移情况。



(a) $p = 0.01$ (b) $p = 0.10$

(c) $p = 0.30$ (d) $p = 1.00$

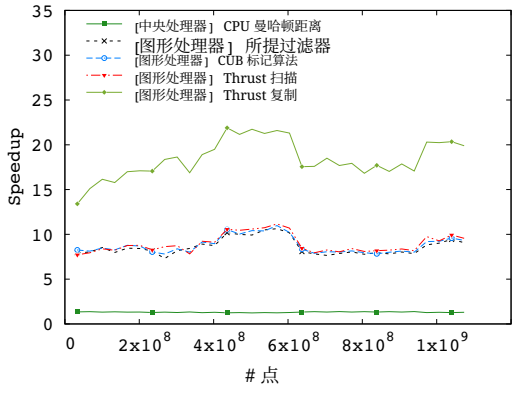
F图 5: 在不同位移值下随机选取位移的中间情况测试

6. 实验与结果

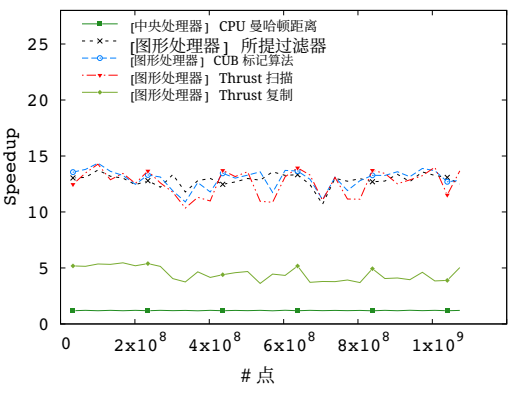
我们使用 C++ 实现了筛选算法，中央处理器部分采用 -O3 优化，图形处理器部分则使用 CUDA 和 NVCC 11.4.2。我们的实现采用单精度浮点运算（FP32）。我们将其性能与 CGAL 5.5 库进行比较，该库是使用最广泛、引用次数最多的库之一。所有实验均在 Patagón超级计算机 [35], 上进行。该超级计算机配备一个 Nvidia DGX A100 GPU 节点、2 个 AMD EPYC 7742 中央处理器（2.6GHz、64 核、256MB L3 缓存）、1 TB DDR4-3200Hz RAM，以及 8 个 40GB 的 Nvidia A100 GPU。实验仅使用一块 A100 GPU。针对每个变体，我们收集了以下指标：过滤时间、点过滤百分比，以及本文所开发各变体的运行时间。

我们通过将各个筛选器变体与其中央处理器对应版本进行比较，评估了它们的性能（使用欧几里得距离和曼哈顿距离查找极值点），结果见图 6。如图 6a所示，对于正态分布，速度最快的实现是 Thrust 复制，相较于 23×，其加速比为 中央处理器-欧几里得距离筛选器。不过，对于圆周分布，它的速度最慢。其他实现的性能相近，如图 6b所示；它们在正态分布和圆周分布下的加速比分别为 11× 和 14×，并且比 Thrust 复制和中央处理器变体都快。需要注意的是，所有图形处理器变体的性能都受筛选元素数量的影响。

图 7 展示了本文所述所有变体的加速效果，计算凸包时使用 CGAL 提供的最快实现（`CGAL:convex-hull-2`），加速比是相对于未使用筛选器的`CGAL:convex-hull-2`计算得出的。请注意，`CGAL:convex-hull-2`对应一种混合算法，它会根据分布计算在 $O(hn)$ 或$O(n \log n)$ 算法技术之间进行选择。图 7a 所示的正态分布加速时间表明，`thrust-copy` 变体显著提升了 23×的速度。同时，其他变体的加速比达到 10×。然而，对于圆周分布，由于问题本身的性质，任何预处理算法都无法实现加速，如图7b所示。与中央处理器变体不同，

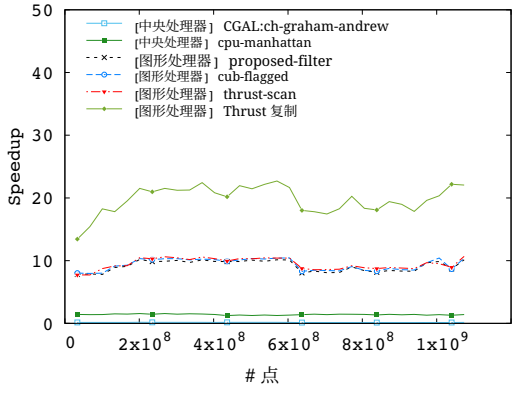


(a) 正态分布的筛选器

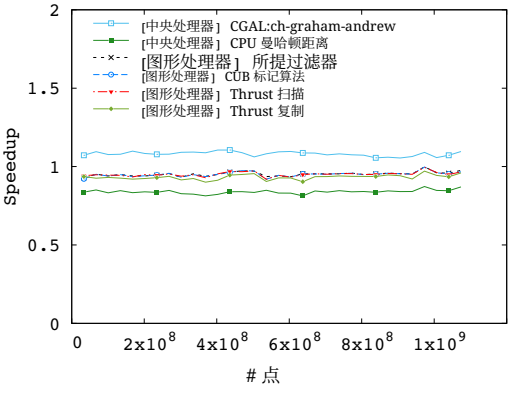


(b) 圆周的筛选器

图 6： 仅在预处理筛选阶段，相较于 CPU 欧几里得距离过滤器，在正态分布和圆周分布下的加速比，点数介于 2^{25} 和 2^{30} 之间。



(a) 正态分布。



(b) 圆周。

图7： 对于正态分布和圆周分布，分别在 2^{25} 至 2^{30} 个点之间，带筛选器的 + CGAL:convex-hull-2 相较于不带筛选器的 CGAL-convex-hull-2 的加速比。

图形处理器变体仅有轻微的开销。

图8 说明，要加速锁的计算，点数必须达到一定的最小值。图 8a 显示筛选一组较少点所需的时间。图 8b 则显示筛选大量点所需的时间。可以看出，这两个数据集的表现不同；表 1表明，对于较小的数据集，需要筛选 $p < 0.03$ 才能加快凸包计算；对于较大的数据集，则需要 $p < 0.01$，从而移除约 6% 的点。此外，还需注意，最高加速比是在 $p = 0.1$时达到的，这一点在图 8的两幅图中均可观察到，曲线在达到最大加速比时出现谷值。最后，图9展示了两种分布下的加速比，其中最快的变体（`thrust-copy`）对于大型数据集，相较于 30× 可达到 `CGAL:convex-hull-2`的加速比；数据集大小为 $p = 0.1$。

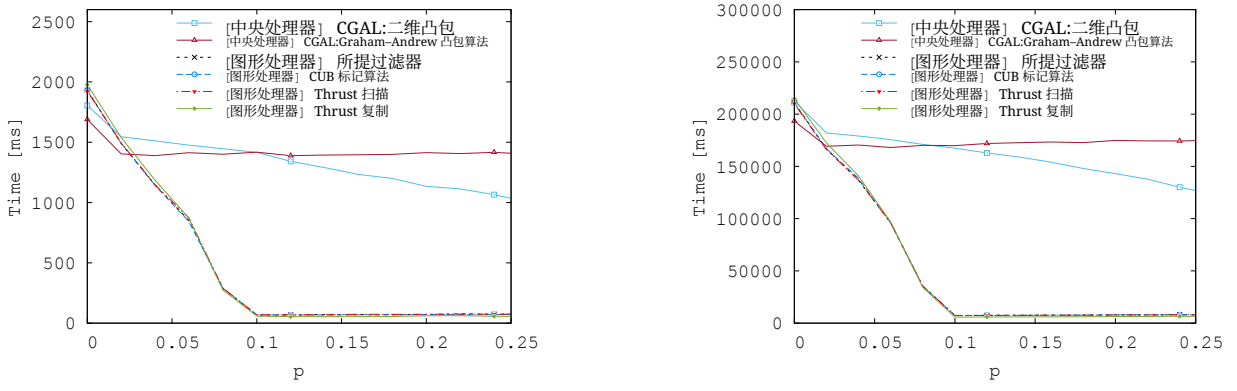
7. 讨论与结论

本文对最新的图形处理器过滤技术进行了全面的性能评估，以解决二维凸包问题。尽管已有一些图形处理器实现采用自定义 CUDA 代码，另一些则使用 Thrust库，但本文的贡献在于比较了三种主要的图形处理器筛选方式：自定义 CUDA 内核、Thrust库和 CUB库，并重点分析了各自的优缺点。

所提出的筛选方法构建一个八顶点多边形，剔除其中所有内部点，并利用并行最小值-最大值归约和曼哈顿距离计算角点。曼哈顿距离简化了

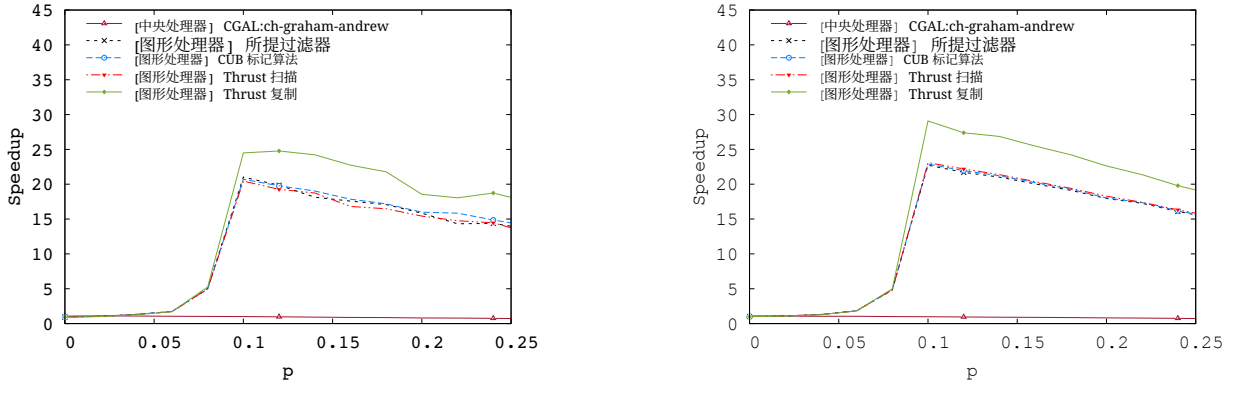
	$n = 10^7$ 点		$n = 10^9$ 点	
p	过滤百分比	加速比	过滤百分比	加速比
0.00	0.01	0.88	0.01	0.92
0.02	13.08	0.94	13.04	1.02
0.04	28.58	1.21	28.69	1.24
0.06	48.48	1.62	48.51	1.75
0.08	81.88	4.83	81.71	4.82
0.10	97.16	20.46	97.17	23.42

表1偏移圆周情况下，算法在过滤阶段平均剔除的点的百分比，以及最快变体相对于 CGAL:ch-graham-andrew 的加速比。



(a) 点数为 10^7　　(b) 点数为 10^9

图8： 偏移圆周（中间情况）下，凸包算法的运行时间随 p 在 $[0, 0.25]$ 之间变化，点数固定。



(a) 点数为 10^7　　(b) 点数为 10^9

图9： 偏移圆周（中间情况）下，凸包算法相对于 CGAL:convex-hull-2 的加速比随 p 在 $[0, 0.25]$ 之间变化，点数固定。

计算过程，无需进行平方根运算。实验结果表明，对于在二维空间中按照正态分布随机生成的点，该方法可将凸包计算速度提高 30×，相较于 CGAL 提供的 CGAL:convex-hull-2 函数而言。

当所有点都位于凸包上时（例如点呈圆形分布），所提出的方法并无任何优势，因为筛选算法不会移除点；这是所有最先进算法的共同特点。

然而，与在中央处理器上运行筛选算法相比，在图形处理器上运行该算法的成本极低（低于 6%）。这意味着，采用基于图形处理器的筛选过程，只会略微增加计算凸包所需的时间。在某些情况下，视具体应用而定，为了在其他场景中获得潜在的加速比，这种成本或许可以接受。

这项工作的一个优点是算法具有良好的可扩展性。Thrust 和 CUB 提供了快速且易于使用的应用程序接口，可用于开发并行算法。然而，它们用于计算最小点和最大点的函数不适合处理大型数据集（超过 2^{30} 个点）。CGAL 也有 2^{30} 个点的上限。另一方面，所提出的 GPU 内核实现易于扩展，只要图形内存允许，就能处理更多的点。此外，GPU 内核实现的性能与基于库的变体相当。

实验结果表明，这种方法在大多数情况下都很高效；即使在最坏情况下，与传统凸包计算相比，筛选开销造成的性能损失也很小。此外，最坏情况在大多数情况下都极不可能发生。作为未来工作，值得研究在图形处理器上实现完整的并行凸包算法，采用所提出的预处理方法，并避免在设备内存与主机内存之间进行不必要的数据复制。此外，加速三维凸包计算，以及利用张量核心和光线追踪核心完成这项任务，也已成为值得关注的课题。

致谢

本研究得到智利南方大学的Patagón超级计算机（FONDEQUIP EQM180042）以及FONDECYT资助项目 #1211484, #1221357的支持。

参考文献

[1] Mark de Berg、Otfried Cheong、Marc van Kreveld 和 Mark Overmars。《计算几何：算法与应用》。Springer-Verlag TELOS，美国加利福尼亚州圣克拉拉，第3版，2008年。[2] Joseph o'Rourke 等。《C语言计算几何》。剑桥大学出版社，1998年。[3] Sergio Salinas-Fernández、Nancy Hitschfeld-Kahler、Alejandro Ortiz-Bernardin 和 Hang Si。Polylla：基于终端边区域的多边形网格生成算法。《Eng. with Comput.》，38(5):4545–4567，2022年10月。[4] S. Meeran 和 A. Share。利用凸包和局部搜索启发式算法进行最优路径规划。《机电一体化》，7(8):737–756，1997年。[5] A. C. Nearchou 和 N. A. Aspragathos。《基于凸包概念的碰撞检测方案，用于生成运动学可行的机器人轨迹》，第477–484页。Springer 荷兰，多德雷赫特，1994年。[6] A.P. Nemirko 和 J.H. Dulá。基于凸包分析的机器学习算法。《计算机科学会议录》，186:381–386，2021年。第14届国际"智能系统"研讨会。[7] Donald R. Chand 和 Sham S. Kapur。凸多胞体算法。《J. ACM》，17(1):78–86，1970年1月。[8] R.A. Jarvis。关于识别平面有限点集的凸包。《信息处理快报》，2(1):18–21，1973年。[9] R.L. Graham。确定有限平面点集凸包的高效算法。《信息处理快报》，1 133，1972 4 132( ): – .[10] C. Bradford Barber、David P. Dobkin 和 Hannu Huhdanpaa。凸包的 Quickhull 算法。《ACM Trans. Math. Softw.》，22(4):469–483，1996年12月。[11] F. P. Preparata 和 S. J. Hong。二维和三维有限点集的凸包。《Commun. ACM》，20(2):87–93，1977年2月。[12] Michael Kallay。rd 中增量凸包算法的复杂度。《信息处理快报》，19(4):197，1984年。[13] Susan Hert 和 Stefan Schirra。三维凸包，2018年。[14] Guy E. Blelloch、Yan Gu、Julian Shun 和 Yihan Sun。随机增量算法中的并行性。《J. ACM》，67(5)，2020年9月。[15] Héctor Ferrada、Cristóbal A. Navarro 和 Nancy Hitschfeld。用于快速构造凸包的筛选技术（r2）。《Journal of Compu- tational and Applied Mathematics》，364:112298，2020年。[16] Abdul Dakkak、Cheng Li、Jinjun Xiong、Isaac Gelado 和 Wen-mei Hwu。利用张量核心单元加速归约和扫描。载于美国计算机协会国际超级计算会议论文集，ICS '19，第46–57页，美国纽约，2019年。美国计算机协会。[17] Cristóbal A. Navarro、Roberto Carrasco、Ricardo J. Barrientos、Javier A. Riquelme 和 Raimundo Vega。利用 GPU 张量核心实现快速算术归约。《IEEE 并行与分布式系统汇刊》，32(1):72–84，2021年。[18] 英伟达。《A100 张量核心 GPU 架构白皮书》，2020年。[19] S. Srungarapu、D. P. Reddy、K. Kothapalli 和 P. J. Narayanan。在图形处理器上快速计算二维凸包。载于*IEEE* 高级信息网络与应用国际会议专题研讨会，第7–12页，2011年。[20] Jiayu Qin、Gang Mei、Salvatore Cuomo、Guo Sixu 和 Yixuan Li。Cudachpre2d：一种用于加速图形处理器上二维凸包计算的直接预处理方法。《并发与计算：实践与经验》，32，2019年4月。[21] Ayal Stein、Eran Geva 和 Jihad El-Sana。Cudahull：图形处理器上的快速并行三维凸包。《计算机与图形学》，36(4):265–271，2012年。几何处理应用。

[22] Guy E. Blelloch、Yan Gu、Julian Shun 和 Yihan Sun。随机增量凸包算法具有高度并行性。载于第32届ACM算法与体系结构并行性研讨会论文集，SPAA '20，第103–115页，美国纽约，2020年。美国计算机协会。[23] Jiayu Qin、Gang Mei、Salvatore Cuomo、Guo Sixu 和 Yixuan Li。Cudachpre2d：一种用于加速图形处理器上二维凸包计算的直接预处理方法。《并发与计算：实践与经验》，32，2019年4月。[24] Gang Mei 和 Nengxiong Xu。Cudapre3d：一种用于加速图形处理器上三维凸包计算的替代预处理算法。《电气与计算机工程进展》，15:35–44，2015年5月。[25] Gang Mei、John Tipper 和 Nengxiong Xu。一种求平面点集凸包的算法。载于第二届国际计算机科学与网络技术会议论文集，ICCSNT 2012，2012年12月。[26] Reham Alshamrani、Fatimah Alshehri 和 Heba Kurdi。一种快速凸包计算预处理技术。《计算机科学会议录》，170:317–324，2020年1月。[27] Alan Keith、Héctor Ferrada 和 Cristóbal A. Navarro。利用并行图形处理器算法加速凸包计算，2022年。[28] Eugene F Krause。《出租车几何：一次非欧几何探险》。Courier Corporation，1986年。[29] Michel Marie Deza 和 Elena Deza。《距离百科全书》，第1–583页。Springer Berlin Heidelberg，德国柏林、海德堡，2009年。[30] Guy E. Blelloch。前缀和及其应用。技术报告 CMU-CS-90-190，卡内基梅隆大学计算机科学学院，1990年11月。[31] Cristóbal A. Navarro、Nancy Hitschfeld-Kahler 和 Luis Mateu。关于并行计算及其在采用图形处理器架构的数据并行问题中的应用综述。《计算物理通讯》，15(2):285–329，2014年。[32] Mark Harris。优化 CUDA。载于国际高性能计算、网络、存储与分析会议论文集，SC '07，2007年。[33] Mark Harris。将计算概念映射到图形处理器。载于 ACM SIGGRAPH 2005课程，SIGGRAPH '05，美国纽约，2005年。ACM。[34] G.E. Blelloch。作为基本并行操作的扫描。《IEEE计算机汇刊》，38(11):1526–1538，1989年。[35] Patagón超级计算机。https://patagon.uach.cl，2021年。