

Southwest University of Science and Technology

K-D盆景-ISA扩展压缩K-D树用于自动驾驶任务

学生姓名: 李东

学生学号: 7220230878

专 业: 电子信息

任课教师: 王耀彬老师

学院(部): 计算机科学与技术学院

K-D盆景-ISA扩展压缩K-D树用于自动驾驶任务

摘要

自主驾驶（AD）系统广泛运用3D点云进行物体检测和车辆定位。因此，在这些系统中，高效处理3D点云至关重要。在本研究中，我们提出了K-D Bonsai，这是一种在半径搜索过程中减少内存使用的技术，而半径搜索是点云处理的一个关键构建模块。K-D Bonsai利用存储点云的数据结构（k-d树）中的值相似性来压缩内存中的数据。K-D Bonsai还利用减少的浮点表示对数据进行进一步压缩，利用点云值的物理有限范围。为了方便集成到现有系统中，我们通过Bonsai扩展实现了K-D Bonsai，这是一组用于压缩、解压缩和操作点的新CPU指令。为了保持基准安全水平，我们精心设计了Bonsai扩展，以检测由于压缩而导致的精度损失，如果必要，可以进行完全精度的重新计算。因此，K-D Bonsai减少了数据移动，提高了性能和能源效率，同时保证了基准准确性和可编程性。我们在Autoware.ai的欧几里德聚类任务上评估了K-D Bonsai，这是一个用于自主驾驶的先进软件堆栈。我们实现了平均端到端延迟提升9.26％，尾部延迟提升12.19％，能源消耗降低10.84％。与相关工作中提出的昂贵加速器不同，K-D Bonsai在最小面积增加（0.36％）的情况下改进了半径搜索。

关键词

自动驾驶硬件，点云，K-D树，半径搜索，压缩，ISA扩展

目 录

1 介绍 1

2 背景 2

**2.1** 自动驾驶中的点云数据 2

2.2 K-D树 2

2.3 半径搜索 2

3 在k-d树上对点云进行半径搜索的压缩 3

3.1 基于值相似性的压缩 3

3.2 通过更小的表示进行压缩 3

3.3 如何在较小的表示中保持准确性 4

4 拟议设计 6

4.1 硬件支持k-d树压缩 6

4.2 改变CPU 6

4.3 软件影响 8

5 结果 9

5.1 评估方法 9

5.2 性能分析 10

5.3 面积和功率分析 12

5.4 能源分析 12

6 相关工作 13

7 结论 14

1 **介**绍

随着传感器、算法和硬件的进步，自主驾驶变得越来越可行，因此人们开始关注如何使这些系统更加高效。点云是自动驾驶中广泛使用的一种数据表示形式，它包含了环境的三维信息，比传统的二维图像更加丰富。点云可以用于许多关键任务，如物体检测、距离测量和车辆定位。

其中，半径搜索是自动驾驶算法中的一个关键操作，其目的是返回离查询点一定距离内的所有点。半径搜索在许多场景中都有应用，例如将附近的点聚类在一起以推断周围的物体形状，或者优化车辆的定位估计。然而，半径搜索是耗时和能耗较高的任务之一。此外，半径搜索还在3D卷积神经网络中使用，用于获取点的邻居以进行卷积操作。

为了改进半径搜索的性能和能源效率，文中提出了一种名为K-D Bonsai的技术，用于对点云进行压缩，减少半径搜索执行过程中的数据移动。该技术利用传感器的操作范围限制和k-d树的特性进行压缩。通过合并重复的浮点数表示中的符号和指数字段，并减小尾数字段的大小，可以减少半径搜索所需的存储空间和内存访问次数，从而提高性能和能源效率。此外，文中还提到了如何在运行时检测精度损失并重新进行完全精度计算的方法。

K-D Bonsai的实现是通过对传统CPU进行少量的硬件和指令集架构扩展来实现的。这些修改对现有硬件进行了优化，并且对可编程性的影响很小。因此，采用K-D Bonsai对现有系统来说是容易的，而不需要昂贵且难以编程的离线加速器。

总结起来，这篇论文提出了以下几点贡献：

一，我们发现了存储在k-d树中的点云数据的浮点表示中的位字段冗余。

二，我们验证了对于自动驾驶中基于点云的算法而言，k-d树半径搜索是可以容忍格式表示的降低的关键操作。

三，我们推导出了一个数学方程，用于验证格式表示的降低是否会影响半径搜索操作的准确性，如果需要的话，将会以基准精度重新计算。

四，我们提出了K-D Bonsai，一种利用数据冗余和格式表示降低的压缩技术。K-D Bonsai在半径搜索期间减少了数据移动，提高了性能和能源效率。

五，我们将K-D Bonsai实现为新的CPU指令，即Bonsai扩展指令，证明我们的方案可以在下一代自动驾驶处理器上轻松采用。我们还使用了一种最先进的开源自动驾驶软件堆栈对所提出的方案进行了验证。

本论文的组织结构如下。第2节介绍了重要的背景概念，包括点云、k-d树和半径搜索。第3节解释了如何将压缩应用于k-d树数据。第4节讨论了设计细节，包括新指令和所需的硬件。结果在第5节进行了分析。最后，第6节回顾了相关工作，并在第7节中给出最终结论。

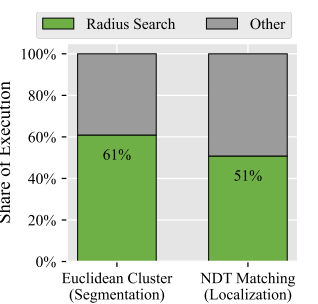
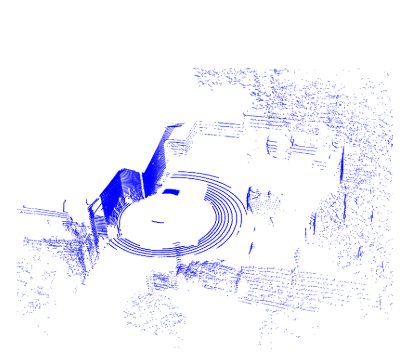


图1：三维点云 图2：半径搜索执行时间

2 背景

**2.1** 自动驾驶中的点云数据

点云是3D空间中的一组点，在自动驾驶中常用于感知和定位任务。通过传感器（如激光雷达）获取点云数据，每个点有坐标（x，y，z）。在感知任务中，点云用于目标检测、跟踪和运动预测。在定位任务中，点云与先前的地图进行匹配，实现精确定位。

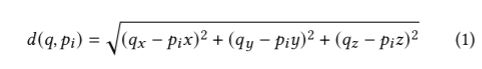
2.2 K-D树

点云数据通常是无序的，通过传感器收集的点被推送到数组中。在原始点云数据上进行搜索是耗时的，因此使用搜索友好的数据结构，如k-d树。k-d树是一种二叉树，用于高效搜索k维数据。我们使用PCL和FLANN库中实现的k-d树。树的创建过程包括选择分割坐标、找到中位数值、将点分配到左右子树等。为了优化性能，优化的k-d树仅在叶子节点存储点，每个叶子节点最多存储𝑚个点。树的拓扑结构受到每个叶子节点的点数和数据的影响。在树的创建过程中，计算每个子树的边界框，父节点使用该信息来计算与每个子树的距离。非叶子节点用于引导搜索过程，以找到符合搜索条件的叶子节点中的点集。

2.3 半径搜索

半径搜索的主要目标是返回距离查询点𝑞距离为𝑟的范围内的所有点。形式上，给定一个三维点云P = {𝑝1, 𝑝2, ..., 𝑝𝑁}，其中𝑝𝑖 ∈ R3，我们希望找到查询点𝑞 ∈ P，R3，在距离𝑟 ∈ R的范围内的邻近点集合N(q, 𝑟) = {p ∈ P | 𝑑𝑖𝑠𝑡(p, q) <= 𝑟}。这种操作通常用于从点云中聚类点，以获取环境中物体的形状。在这种情况下，半径搜索被连续用于将附近的点关联到聚类中：例如，如果点𝐴在点𝐵的半径内，而点𝐵在点𝐶的半径内，那么𝐴、𝐵和𝐶都属于同一个聚类。

要在k-d树上执行半径搜索，需要提供查询点𝑞和目标半径𝑟。将遍历树，将当前节点的分割坐标值与𝑞的相应坐标进行比较。此比较提供了一个最佳提示，表明哪个子树更接近，因此更有可能导向包含在𝑟范围内的点的叶子节点。这个下降过程将导致包含𝑞本身的叶子节点（以及该叶子节点中的其他点）。在展开树导航时，也会考虑到另一个子树（在下降时没有选择的子树）。如果从𝑞到子树的分割坐标上的距离小于𝑟，则访问该子树，并继续下降。每当搜索找到一个叶子节点时，计算𝑞与叶子节点上每个点𝑝𝑖之间的距离。通常使用欧氏距离𝑑进行计算。



为了避免进行平方根运算，一种常见的优化方法是计算欧氏距离的平方。



然后，我们可以将𝑑²与平方半径𝑟²进行比较，以对点进行分类。



每当𝑝𝑖在𝑞的半径范围内时，它会被添加到半径搜索结果列表中。

3 在k-d树上对点云进行半径搜索的压缩

3.1 基于值相似性的压缩

当构建k-d树时（如第2节所述），点云空间被细分，使得附近的点最终聚集在叶节点中。因此，点的坐标彼此相似。这种情况在图3中进行了示例，为简单起见，只展示了二维情况。

图3a展示了一个情况，其中空间接近的点被同一个k-d树叶节点持有。坐标系的原点位于车辆（激光雷达传感器所在位置），点到原点的距离以米为单位给出。图3b列出了这些点的坐标（在本例中是𝑥和𝑦），展示了它们的内部浮点表示（32位IEEE 754）。我们绘制了浮点表示的符号（𝑠）、指数（𝑒）和尾数（𝑚）字段。按照IEEE 754标准，存储的值由以下方程给出：



当点在空间中靠近时，它们的坐标很可能具有相同的符号（即它们都属于坐标系中的同一象限）和指数（即值在相同的2的幂次范围内）。例如，图3中的所有点的x坐标都在8.0和16.0之间，因此指数字段的值都是130。

为了验证这一观察的适用性，我们检查了涵盖超过3700万点的一组点云数据，这些数据用于Autoware.ai中的欧几里得聚类节点（有关数据集的详细信息可以在第5节中找到）。我们发现78%的叶节点在𝑥坐标上具有相同的指数和符号，而在𝑦坐标上为83%。

因此，点云内部浮点表示的值相似性非常常见，是k-d树数据的适合压缩的来源。如果一个坐标上所有点的符号和指数相同，我们只需存储它们一次，并在计算发生时仅在CPU内部重新构建值（详见第4节）。

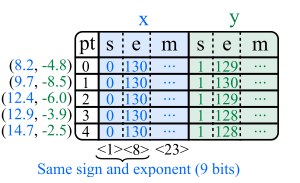
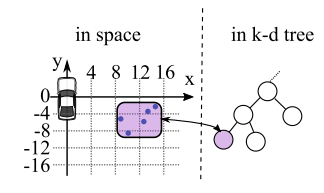


图3a：映射到同一k-d树叶子节点的邻近点 图3b：每个点的浮点字段

3.2 通过更小的表示进行压缩

压缩浮点表示的符号和指数字段（3.1节）可以在每个坐标使用32位的情况下达到最大的压缩比为9比32。这是在Autoware.ai和PCL中默认使用的32位基线。为了进一步提高压缩比，我们需要处理浮点表示的剩余23位，也就是尾数字段。问题在于尾数字段在叶节点中很少重复出现。因此，基于值的相似性进行的压缩对尾数位来说是不太有效的。然而，我们可以通过降低浮点表示的精度来减小其大小。表1显示了使用不到32位的不同浮点格式进行分类误差（公式3）的情况。我们使用与3.1节相同的点云数据集进行实验。我们尝试了两种常见的16位浮点表示：IEEE-754 16位（IEEE半精度格式）和bfloat 16位（用于机器学习应用，例如CUDA支持的格式），以及一个自定义的24位表示，用作比较的中间参考。

总体而言，我们发现16位和24位浮点表示的分类误差都低于1%。这表明减少表示可以有效地进行压缩，并引入较少的错误。需要注意的是，对于IEEE-754 16位和自定义浮点（24位）表示，指数字段的大小也减小了，影响了可表示的数值范围。然而，来自激光雷达等传感器的点云数据具有有限的范围。例如，Velodyne HDL-64E（通常使用的激光雷达传感器）的最大覆盖范围为120米。实际上，表1中所示的错误都不是由于表示数值范围不足导致的。因此，在我们的情况下，减少指数位不是一个问题，而是一个可以利用的优势。

进一步地，我们评估了不同表示方法所涉及的权衡，以选择适合我们压缩方案的最佳表示。我们注意到IEEE-754 16位和bfloat具有相同的大小，但是在使用指数位（用于范围）和尾数位（用于精度）方面更加平衡，精度提高了一个数量级。而我们自定义的24位浮点数中额外的8位用于增加精度并没有得到回报，因为16位格式的精度已经足够好（<1%的误差）。最后，IEEE-754 16位已经在现代CPU上部分支持（例如，在ARM上进行存储），因此对于现有架构的影响较小，而不需要引入新的自定义格式。基于这些原因，我们选择使用IEEE-754 16位来表示k-d树叶节点的点，并在此基础上应用基于值相似性的压缩（3.1节）。

关于在k-d树半径搜索中使用较小的表示的主要结论如下：1.尾数位可以减少而几乎不会损失精度；2.自动驾驶算法使用的点都靠近车辆，因此指数位可以减少而仍然可以表示点云的值。

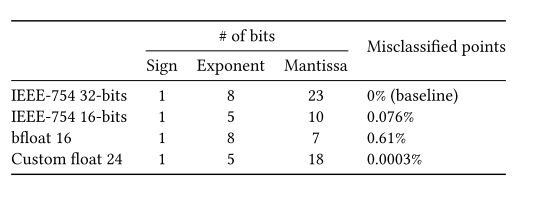


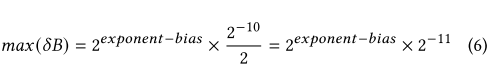
表1：使用较小浮点表示的欧氏聚类半径搜索的分类误差。

3.3 如何在较小的表示中保持准确性

到目前为止，我们已经讨论了两种不同的方法来减小k-d树搜索的点的大小，但副作用是引入了分类错误。然而，由于自动驾驶系统具有安全关键性，引入错误是不可取的，并且可能带来难以测试的后果。因此，我们提出了一种检测分类错误并以基准准确性重新计算的方法。为此，我们假设可以访问原始点和压缩点。这个想法是使用压缩点来减少内存使用，并在检测到可能的错误分类时特例地查找原始的32位值。  
设𝐵为我们希望在16位IEEE-754格式中表示的32位IEEE-754格式的数，代价是与精度损失相关的误差𝛿𝐵。令𝐵'表示𝐵在16位表示中的结果值。



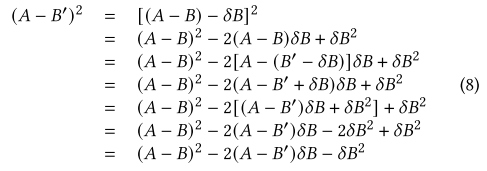
对于IEEE-754标准中的默认舍入模式，尾数的最低有效位（LSBs）会被舍弃，并且结果数会向最近的数进行四舍五入。对于指数在两种表示方式中都可以相等存储的值（我们的情况，参见3.2节），尾数的舍入是唯一的误差来源。在这种情况下，第11到第23位的尾数位将用于将数值四舍五入到最近的值，调整16位结果数的第10位。  
由于我们可以向上或向下舍入到最近的数，最大的尾数误差将是10位的值的一半，而最大的值误差也将取决于指数，因为2的指数次方乘以尾数形成浮点数（公式4）。在这些条件下，将一个数𝐵从32位转换为16位IEEE-754浮点数时，舍入的最大误差𝛿可由以下公式给出：



这里的要点是只使用指数就可以推断出最大的舍入误差。因此，有了𝐵'，就不需要查找𝐵，因为根据我们的假设，指数值在𝐵'和𝐵中都是可表示的。  
现在，让我们继续找到一个32位值𝐴和一个16位值𝐵'之间的平方差的误差。我们从减法开始，应用公式5。



其中−𝛿𝐵是相关误差。我们可以继续并评估平方操作（𝐴 − 𝐵'）²的误差，应用公式5、公式7和牛顿二项式定理。



其中−2(𝐴 − 𝐵')𝛿𝐵 − 𝛿𝐵²是差的平方操作（𝜖𝑠𝑑）的相关误差。注意，𝛿𝐵可以是正数或负数，这取决于数值是向上舍入还是向下舍入。然而，在运行时，我们将不知道是哪种情况，因为这将需要获取并检查原始值的最低有效位（LSBs），而我们正试图避免这样做。相反，我们可以保守一点，并计算𝜖𝑠𝑑的最坏情况幅度，使用𝑚𝑎𝑥(𝛿𝐵)（公式6）代替𝛿𝐵。



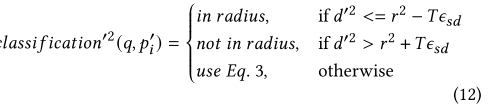
再次注意到，𝑚𝑎𝑥(𝛿𝐵)和𝑚𝑎𝑥(𝛿𝐵)²可以直接通过𝐵'的指数获得。最后，我们可以计算形式为(𝐴 − 𝐵')²的近似平方差，对每个坐标进行求和，以获得近似的欧氏距离平方𝑑'²。



同样地，我们可以将每个坐标的平方差的最大误差相加，得到总误差𝑇𝜖𝑠d。



最后，我们可以使用公式10和11进行分类（使用𝑝′𝑖代替𝑝𝑖）。



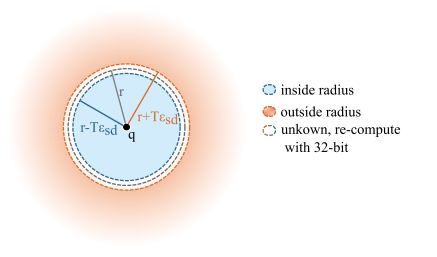


图4：公式12的直观表示

换句话说，我们可以使用最坏情况误差𝑇𝜖𝑠𝑑来确认使用𝑝′𝑖进行分类的正确性。我们通过定义一个围绕𝑟²的区域，值为𝑟²−𝑇𝜖𝑠𝑑和𝑟²+𝑇𝜖𝑠𝑑，如图4所示。每当𝑑′²落在区域之外时，分类与基准分类（由公式3计算）相同。例如，一个在半径内但在区域之外的点，即使将𝑇𝜖𝑠𝑑添加到𝑑′²上，也不能使其超出半径。另一方面，当𝑑′²落在区域之内时，误差可能足够大以改变分类，并且无法保证与基准分类相同。在这种情况下，我们建议获取原始点𝑝𝑖，并使用完整精度（使用公式3）重新进行分类。

4 拟议设计

4.1 硬件支持k-d树压缩

在推导出一个压缩方案（第3节）之后，我们有兴趣将其应用于执行半径搜索的任务中。一种朴素的方法是使用仅软件的解压缩点。然而，通过软件逐次检查和重新排序位会使半径搜索变慢约7倍（数据集和实验平台见第5.1节），从而削弱了压缩的好处。另一种选择是添加硬件以有效支持K-D Bonsai。

有两个主要选项来在硬件中实现K-D Bonsai：1.使用离核加速器；2.通过ISA扩展在CPU中实现。

在这项工作中，我们选择后者，并在下文中进行了解释。首先，CPU将不得不传输数据以与加速器进行通信。然而，K-D Bonsai所做的叶节点处理是一个细粒度任务，只需要少量周期即可完成（实现细节见第4.2节）。因此，使用CPU内部的适当硬件来执行解压缩和点分类操作可以避免通信成本。或者，解压缩操作可以合并以分摊通信成本。然而，加速器可能更昂贵（见第6节）。同时，叶节点处理只是点云处理的一小部分，限制了最大性能提升（Ahmdal定律），并可能危及加速器的采用。尽管如此，工业界偏向于使用在实际解决方案中经过较少实验的方法来加速任务，很少使用加速器。

4.2 改变CPU

第3节讨论的思路的一个主要优势是它们在硬件中的易于实现和低成本性。实际上，所需的新功能集很小：1.我们需要压缩数据；2.解压缩数据；3.支持计算形式为（𝐴 - 𝐵'）^2（方程8）的平方差（以及相关误差）。

图5描述了我们添加到CPU的两个组件以及它们与现有硬件的交互方式。我们讨论的第一个添加的组件是压缩/解压缩单元，位于图的顶部。该单元分为两个部分：一个名为ZipPts缓冲区的缓冲区和一个压缩/解压缩逻辑。

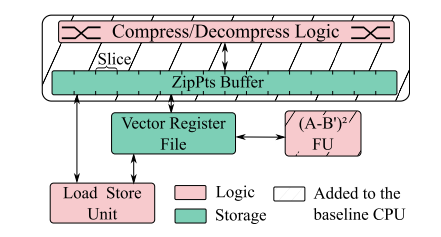


图5：添加到基准CPU的新组件以及它们如何与先前存在的组件交互

ZipPts缓冲区。ZipPts缓冲区用于存储压缩和未压缩的16位点，作为压缩和解压缩操作的源操作数和目标操作数。在我们的实现中，限制了ZipPts缓冲区的大小，最多可容纳16个点（默认情况下，PCL中每个叶子节点的点数为15）。缓冲区还为编码𝑥、𝑦和𝑧坐标是否压缩保留了3位空间。

缓冲区有两个128位端口用于与向量寄存器文件进行接口，并有一个128位端口用于与加载存储单元进行接口。因此，数据以128位的块进行交换，称为ZipPts缓冲区切片。当需要传输少于128位的数据时（例如，压缩数据的最后一个块），我们用零填充数据。端口的宽度与基准CPU中已有的端口相同（参见第5节），例如向量寄存器文件。因此，我们可以直接将数据从内存加载到ZipPts缓冲区，或将数据从ZipPts缓冲区存储回内存。总之，我们可以将要压缩的点加载到缓冲区中，将压缩后的数据存储回内存，并加载要解压缩的压缩数据。同时，我们还可以将ZipPts缓冲区中的值写入寄存器文件，使其可供功能单元（FUs）使用。ZipPts缓冲区与压缩/解压缩逻辑紧密耦合，压缩/解压缩逻辑负责重新排列数据位，后续将进行讨论。

压缩/解压缩逻辑。该单元重新排列ZipPts缓冲区中的数据，对来自k-d树叶子节点的点进行压缩和解压缩。在两种情况下，逻辑必须提供点的数量。在压缩过程中，该单元在ZipPts缓冲区中的每个坐标上读取并比较元组<𝑠𝑖𝑔𝑛, 𝑒𝑥𝑝𝑜𝑛𝑒𝑛𝑡>，见图6。如果所有点的元组<𝑠𝑖𝑔𝑛, 𝑒𝑥𝑝𝑜𝑛𝑒𝑛𝑡>在所有坐标上都相同，则压缩后的数据中只会出现一个副本的<𝑠𝑖𝑔𝑛, 𝑒𝑥𝑝𝑜𝑛𝑒𝑛𝑡>。每个坐标都有一个压缩位标志（𝑐𝑋，𝑐𝑌，𝑐𝑍），用于指示其<𝑠𝑖𝑔𝑛, 𝑒𝑥𝑝𝑜𝑛𝑒𝑛𝑡>是否被压缩。

在解压缩过程中，该单元读取压缩位标志，重新组织数据，并重新创建单个<𝑠𝑖𝑔𝑛, 𝑒𝑥𝑝𝑜𝑛𝑒𝑛𝑡>的多个实例。

为了举例说明，图6详细说明了压缩流程和压缩数据的组织方式。首先，尾数值直接绕过压缩，放入缓冲区。然后，压缩的<𝑠𝑖𝑔𝑛, 𝑒𝑥𝑝𝑜𝑛𝑒𝑛𝑡>元组放入ZipPts缓冲区，然后是剩余的未压缩的<𝑠𝑖𝑔𝑛, 𝑒𝑥𝑝𝑜𝑛𝑒𝑛𝑡>元组。三个压缩位标志位于缓冲区的开头。

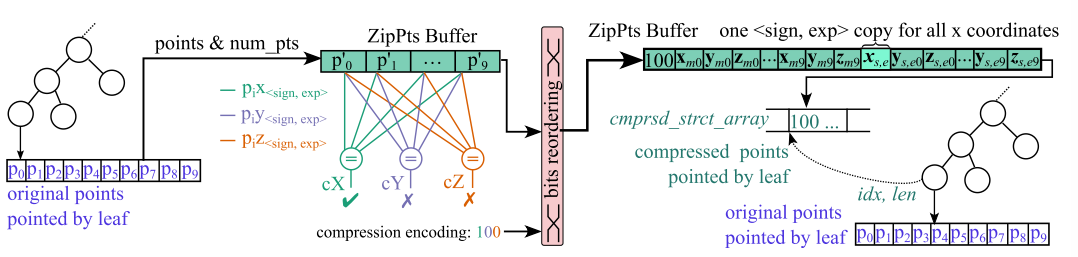


图6：压缩叶子节点中的点

近似平方差功能单元。当从内存中获取压缩点并将其解压缩为16位值后，可以将它们从ZipPts缓冲区移动到向量寄存器文件中。此时，可以进行用于计算平方差和误差的功能单元（FU）。该单元实现了公式8，并可以连续多次使用（对于每个坐标）来计算公式10和11以进行分类。图7详细描述了FU的内部结构。它有两个输入操作数，A是32位值（例如查询点的一个坐标），B'是16位值（例如叶子中某个点的相同坐标），然后将其扩展为32位（不改变B'的值），以便在32位硬件中进行计算，防止16位错误被放大。平方差的计算通过常规的减法和平方操作进行。

最坏情况误差（𝑚𝑎𝑥(𝜖𝑠𝑑)，公式9）的计算比平方差本身的计算更复杂。幸运的是，我们可以利用一些观察结果来简化其计算。首先，由于𝑚𝑎𝑥(𝛿𝐵)仅取决于𝐵'的指数，并且存在32个可能的指数（2^5 = 32），我们可以预先计算2·|𝑚𝑎𝑥(𝛿𝐵)|和|𝑚𝑎𝑥(𝛿𝐵)|^2的值，并将其存储在一个小型（32行）查找表中。在操作开始时，可以使用𝐵'的指数来查找此小表（在图7中称为部分误差存储器）。此外，为了计算最坏情况相关误差𝑚𝑎𝑥(𝜖𝑠𝑑)，可以借用为平方差计算的|𝐴 − 𝐵' |。

由于解压缩同时输出多个点，可以同时进行计算。为了利用这一点，可以实例化多个近似平方差功能单元（图8），以向量方式进行计算。在每个功能单元中，同时计算差的平方和相关误差，每个功能单元分别对输入向量𝑣𝐴和𝑣𝐵'的一部分进行计算。对于半径搜索分类，将查询𝑞的一个坐标加载到𝑣𝐴的所有索引中，将多个点的相同坐标加载到𝑣𝐵'中。

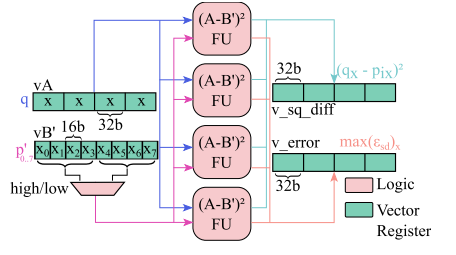
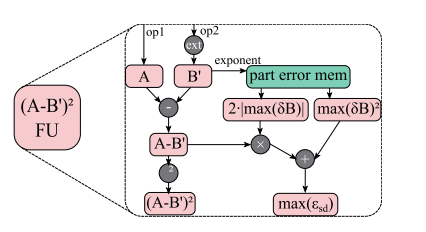


图7：平方差与误差的详细过程 图8：向量差的平方功能单元

4.3 软件影响

在软件中使用新硬件的方法如下：我们将第4.2节提到的新硬件功能作为Bonsai-extensions暴露为新的CPU指令（表2）。这些指令分为压缩、解压缩和计算三个类别。一些指令触发多个微操作，我们将在使用说明中一起解释。在构建k-d树期间创建叶子节点时，我们可以使用压缩指令对叶子节点中的点进行压缩。首先，使用LDSPZPB指令逐个将点加载到ZipPts缓冲区，并在加载时将原始32位值转换为16位。然后，使用CPRZPB指令在ZipPts缓冲区中进一步压缩数据。此时，ZipPts缓冲区中包含压缩的结构，并记录了其字节大小（长度）。最后，使用STZPB指令将压缩的数据存储到内存中，并指示需要存储的ZipPts缓冲区切片数量。在修改的PCL代码中，我们创建了一个名为cmprsd\_strct\_array的字节数组，用于按顺序存储压缩的结构。同时，我们还记录了在k-d树中放置在cmprsd\_strct\_array中的压缩结构的起始地址和长度，以便在半径搜索（树遍历）期间获取压缩数据。我们使用C联合体来重复使用在叶子节点上未使用的树字段来存储此信息，从而不增加k-d树的大小。在PCL代码中，我们还跟踪数组中下一个空闲索引的位置，以供下一个压缩结构使用。

首先将压缩的结构加载到ZipPts缓冲区，需要从cmprsd\_strct\_array获取压缩数据的地址和大小。解码器根据切片数量生成相应数量的加载微操作，将数据从内存加载到ZipPts缓冲区。然后进行解压缩微操作，读取压缩编码并按照16位点重新排序。最后，使用写回微操作将点的值移动到向量寄存器文件中。将解压缩后的点写回到六个向量寄存器中，每个坐标需要两个向量寄存器。

对于每个坐标，使用SQDWEL和SQDWEH指令计算与查询点的向量之间差的平方。查询点的坐标值可以使用现有的向量指令加载到向量寄存器中。由于基线CPU SIMD单元（ARM NEON）有四个32位通道，但每个坐标有八个值（在FUs中计算的32位中的16位），将值分为低位和高位两组，并在四个通道中逐个计算。每个点索引的结果在两个向量寄存器中可用，一个保存计算得到的差的平方值，另一个保存最大误差。

然后，使用现有指令对每个坐标上的距离进行累加，并与𝑟2进行比较，进行分类。如果结果不确定，可以使用基线代码，即读取32位点并计算32位距离。为了保证性能，这种情况应该很少发生，否则压缩/解压缩将消耗时间而没有实际好处。

对于AD任务，通常在开始时为每个帧构建一次树，然后在帧处理过程中进行多次搜索。压缩叶子节点的点会增加树构建的开销，但在搜索过程中加载的数据量较少，可以摊销初始开销。

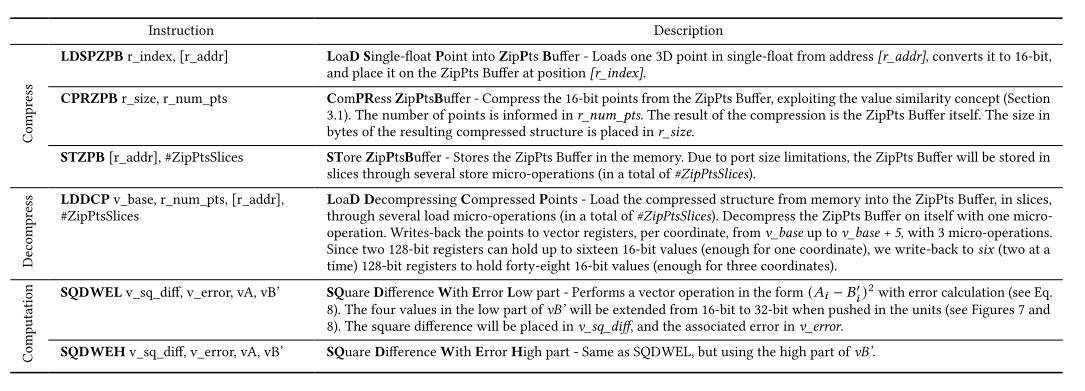


表2：Bonsai-extension指令

5 结果

5.1 评估方法

从软件角度来看，我们依赖于Autoware.ai 来验证我们的想法。Autoware是一种先进的开源软件堆栈，用于自动驾驶，由学术界和工业公司的贡献构建。它具有多种算法，用于执行自动驾驶，包括传感器处理、感知和执行控制等。在这项工作中，我们选择了Autoware中的一个代表性算法，即欧几里德聚类算法，来验证我们的k-d树半径搜索优化的效果，尽管其他算法也适用于我们的优化（例如Autoware的定位算法）。欧几里德聚类算法是Autoware.ai感知流水线的重要组成部分。该算法对源点云进行聚类，有助于推断物体的形状、几何和距离。值得注意的是，先前的研究报告称欧几里德聚类算法是Autoware.ai流水线中延迟较高的任务之一。在欧几里德聚类算法中，半径搜索操作被广泛用于查找应属于同一聚类的附近点。我们使用来自八分钟汽车驾驶序列的点云帧子集来模拟欧几里德聚类算法。由于我们的周期精确模拟器（下文详细介绍）的执行速度比真实硬件慢几个数量级，我们使用系统性子采样（等时间间隔的固定大小样本）来选择点云帧的子集。这个想法受到先前工作的启发，如果正确选择参数（间隔数量和长度），可以获得良好的结果。我们尝试了多种参数，最终选择了每个样本300毫秒的20个样本，总计6秒的真实数据和60帧的处理。表3详细说明了子采样误差，证明它是代码行为的快速准确代理。

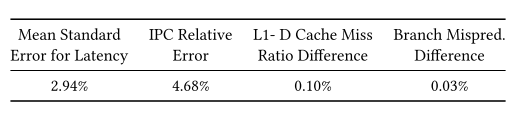


表3：子采样误差

我们在gem5模拟器中实现了Bonsai扩展（表2），针对具有ARM AArch64 ISA的OoO（Out-of-Order）CPU。我们基于gem5中预定义的大型CPU模型（见表4），调整参数，如频率，以匹配技术扩展，以复制ARM Cortex A72的行为。虽然我们的解决方案与ISA无关，但我们选择了ARM作为自动驾驶的代表性ISA（例如NVIDIA DRIVE中使用的ISA）。我们修改了PCL版本1.10及其辅助库FLANN版本1.9.2，使用我们的指令在半径搜索过程中，如第4.3节所述。我们没有修改编译器，而是直接使用ARM汇编语言中的.byte-code指令在库内部编写指令。我们在PCL中公开了一个布尔变量，用户可以激活新指令以进行半径搜索。当变量为true时，代码使用Bonsai扩展，否则使用基线代码。两种情况下的搜索结果是相同的。

我们在gem5中以Full System模式（Ubuntu 18.04）执行Autoware的欧几里德聚类算法。我们使用gem5的快速转发功能和KVM硬件虚拟化来达到子采样帧的区域。对于能耗结果，我们在32纳米工艺下使用McPAT对CPU进行建模，并使用gem5报告的统计数据提供给McPAT功耗模型。我们使用Synopsys Design Compiler在14纳米工艺下综合新的功能单元（压缩/解压缩和差值平方与相关误差），以估计其面积和功耗。为了在单一工艺下统一结果，我们使用Stillmaker等人描述的方法，将McPAT报告的基线CPU数据从32纳米工艺缩放到14纳米工艺。

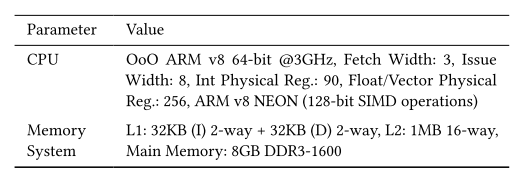


表4：使用的基准CPU型号

5.2 性能分析

图9a显示了欧几里德聚类提取核心的关键性能指标，包括基线代码和使用Bonsai扩展的情况。这是算法的主要核心，占据了90%的执行时间（使用Valgrind 测量），其中进行了k-d树的构建和搜索。由于每个指标具有不同的尺度，我们对每个指标进行了基线代码的归一化处理。我们可以看到，Bonsai扩展减少了内存指令的数量，负载减少了23%，存储减少了18%。

图9b提供了对这一改进的直观理解，描绘了在一帧搜索过程中从内存中提取点所需的字节数的显著减少。当我们使用Bonsai扩展加载压缩点时，相比基线代码，我们只需要加载原来所需字节数的一部分（37%）。尽管这个值是针对数据集的第一帧的，但在所有帧中的行为都是类似的。

内存使用的减少带来了几个好处。首先，它减少了已提交指令的数量，最终表明我们的Bonsai扩展减少了计算成本，并提高了半径搜索处理的效率。其次，它减少了对L1 D-cache的访问量，减少了应用程序对内存的依赖，也增加了CPU的效率。第三，由于前两个原因，它减少了提取核心的执行时间，降低了12%。延迟是自动驾驶算法中的一个重要问题，然而，当我们观察到这些好处仅来自于向ISA中添加了五条新指令时，这一点尤为显著。

图9a还显示K-D Bonsai增加了L1 D-cache的缺失次数。尽管Bonsai扩展从内存中加载压缩点，这些点在内存中是连续的，但当分类结果不确定时（图4中的白色外壳），它也会访问原始点列表。这些对另一个数据结构的不频繁访问是导致较高级别的内存层次中缺失的主要原因。然而，从绝对数量来看，这并不是一个问题。由于L1缓存的访问次数比L2高47倍，比主内存高300倍，我们仍然可以看到执行时间的好处。图10将内存访问次数放在不同的内存层次的视角下。这一现象凸显了选择适当的浮点数表示的重要性，正如我们在第3.2节和表1中讨论的那样，以最小化32位重新计算的开销。在我们的实验中，只有0.37%的分类需要依赖基线计算。如果我们在选择表示时不小心，错误可能不会那么少见，从而危及K-D Bonsai的好处。

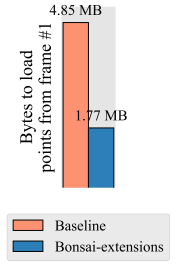
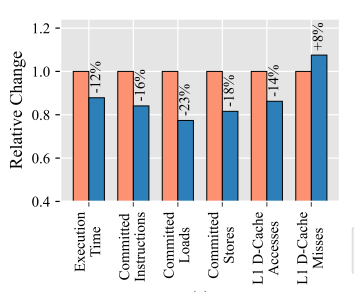


图9a 图9b

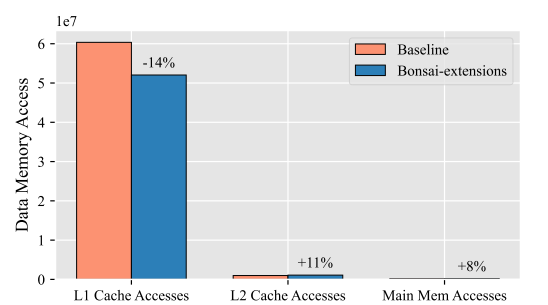


图10：访问内存层次结构的不同级别

接下来，我们评估了欧几里德聚类处理帧的端到端延迟。这一点很重要，因为迄今为止评估的提取核心只是算法工作的一部分。其他任务，如点云预处理和将点标记为其相应的聚类，也必须执行。图11显示了所有子样本帧的欧几里德聚类端到端处理时间的分布情况。与任何标准箱线图一样，箱子包含50%的值。我们用一个白色圆圈和辅助虚线表示每个分布的均值（而不是箱线图的中位数）。使用Bonsai扩展可以加速平均端到端延迟9.26%。在自动驾驶的背景下，减少端到端延迟意味着减少车辆的反应时间，因此更快地执行动作，并提高整体安全性。此时，我们要记住K-D Bonsai的好处是在保持相同基准准确性的情况下获得的（第3.3节）。此外，由于欧几里德聚类通常是感知瓶颈，K-D Bonsai的改进直接转化为整体自动驾驶的改进。

自动驾驶算法的另一个重要方面是它们的端到端尾延迟。与平均延迟不同，尾延迟评估算法在计算需要最长时间的情况下的性能（例如，在欧几里德聚类中，当点云中有更多的点需要处理时）。K-D Bonsai在99th百分位尾延迟方面再次显示出优势，将其加速了12.19%。因此，K-D Bonsai在最需要的时候改善了性能。

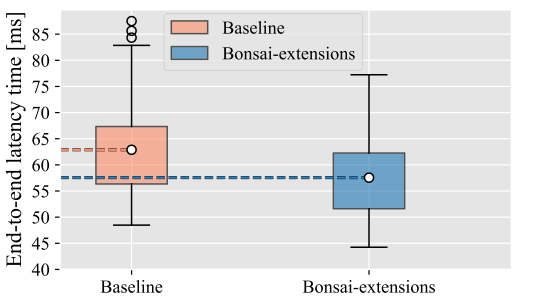


图11：欧几里德聚类算法的端到端时延分布

5.3 面积和功率分析

现在让我们来研究实施我们技术的硬件成本。表5展示了支持K-D Bonsai所引入的面积和功耗开销，根据5.1节中所解释的方法。总体而言，支持新指令的硬件非常简单，面积增加了0.051mm2，相对于基线增加了0.36%。同样，支持K-D Bonsai会增加24mW的动态功耗（相对于基线增加了1.29%）。这些结果强调了我们的解决方案是多么非侵入性。在自动驾驶的背景下，引入最小的功耗和面积开销对于满足散热约束和设计小型自动驾驶车辆（例如用于交付）尤为重要。

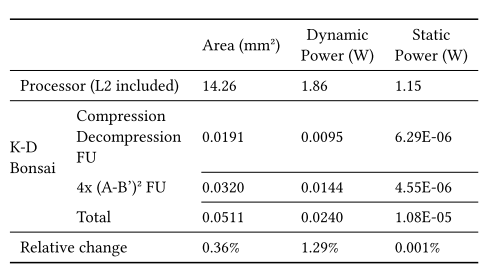


表5：基准CPU和K-D Bonsai的面积和功耗

5.4 能源分析

最后，我们将介绍K-D Bonsai在欧几里德聚类的提取核心中的能量消耗结果。图12以箱线图的形式展示了结果（与端到端延迟的图11相同）。能量消耗的减少是由执行时间、指令数量和内存访问次数的减少驱动的，这抵消了动态功耗的小幅增加（表5）。平均而言，使用Bonsai扩展可以将能量消耗降低10.84%。K-D Bonsai成功提高了能量效率，这在自动驾驶中是一个关注点，以确保计算平台不会减少行驶里程（例如在电池驱动的车辆上）。

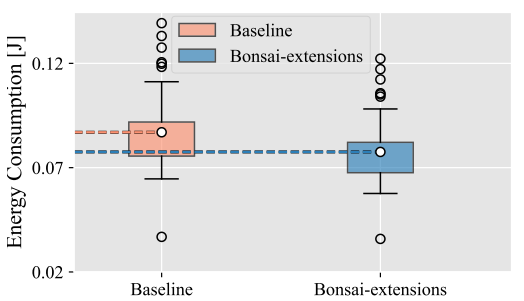


图12：欧氏聚类算法中提取核的能耗分布

6 相关工作

近年来，激光雷达技术和自动驾驶的进展推动了改进点云处理的先前工作。Heinzle等人提出了一种特定的硬件，用于改进k-d树点云中的半径搜索和最近邻搜索。他们的主要思想是搜索比必要的点稍多一些（例如，请求更大的半径），并将扩展的结果集用于后续的空间相邻查询。他们在FPGA上实现了这个想法，并且与CPU相比，在忽略CPU-FPGA传输成本的情况下，查询吞吐量提高了68%。该工作声称所使用的平台缺乏高效的CPU-FPGA接口，如果考虑通信成本，那么基准CPU性能的一半将会减少。这凸显了加速器集成面临的重要挑战。由于K-D Bonsai是作为CPU的一部分实现的，因此在数据进出核心时不需要额外的开销。此外，尽管他们的工作侧重于加速遍历过程，但K-D Bonsai改进了叶节点处理，减少了在遍历之后将数据带入核心（通过CPU中的（解）压缩）所需的负载总数，因此与他们的技术是相互独立的。

最近的一项工作介绍了Tigris，这是一种用于加速点云配准中的半径搜索和最近邻搜索的加速器（这是k-d树搜索的一个主要应用，详见第1节）。Tigris以前端/后端的方式划分遍历和叶节点处理。多个查询并行遍历，将叶节点数据卸载到后端，多个FU将执行距离检查。为了利用更高的性能，Tigris还具有一种在先前获得的结果集上进行搜索的方案，导致它们的搜索是近似的。该加速器相对于CPU改进了配准的端到端延迟达86.6%，但需要总面积为15.57mm2（超过我们的基准CPU，参见表5）。该工作没有报告将查询卸载到加速器的成本。QuickNN也加速了基于k-d树的点云的最近邻搜索。在他们的目标应用中，点云帧被用作新帧的参考。他们利用这种行为在加速器架构中进行了重叠执行和数据共享的树构建和树遍历。此外，他们提出了一个聚集读取和聚集写入缓存，将对离片外存储器的访问合并在一起。与Tigris类似，他们的工作并行处理多个查询，并进行近似搜索。还提出了在高维空间中进行最近邻搜索的加速器，但问题的属性和要求与AD使用的3D数据不同。基于点云的CNN的加速器Mesorasi（例如PointNet++）提出了延迟聚合，允许邻居搜索和特征计算在时间上重叠，隐藏延迟。在他们的提案中，点云搜索时间保持大致相同，大部分的好处来自更快的特征计算。一项较新的工作通过将k-d树搜索的回溯步骤限制在子树中来改进Mesorasi的方法，但会牺牲一些准确性。此外，在存在冲突的情况下，他们的解决方案可以重复使用相似的点或完全忽略遍历路径。这种近似方案的准确性大部分在训练过程中得到校正，因此限制了其适用于机器学习场景。与Mesorasi类似，PointAcc也加速基于点云的CNN，提出了一种基于排名的通用加速器单元。他们相对于Mesorasi增加了映射操作的数量，例如支持半径搜索和最远点采样以获取点云中的输入。

一些工作利用GPU改进了点云处理。Buffer k-d树提出了使用缓冲区延迟处理相同叶节点的查询，直到收集到足够的工作。RTNN将最近邻搜索问题转化为射线追踪问题，并利用GPU中的现代射线追踪硬件改进了搜索效果。然而，该工作显示出在点云规模减小时，数据传输开销变得越来越重要，其效果仅在比通常在一个激光雷达帧中处理的点云数量大得多（数千个点）的点云上有效。Nguyen等人关注软件角度，使用不同的数据结构实现了欧几里德聚类任务，并观察了它们在GPU硬件中的效率。然而，Autoware.ai算法的评估表明，使用GPU进行欧几里德聚类与使用乱序执行的CPU表现相似，这是由于GPU通信开销引起的。确实，Autoware.ai默认使用CPU而不是GPU运行欧几里德聚类。同时，仅为点云处理添加一个GPU所带来的面积和功耗开销比K-D Bonsai大几个数量级，这是AV中需要最小化的关键方面。

总结起来，大多数先前的工作提出使用加速器，这意味着通信开销大、面积成本高，并且缺乏可编程性。此外，许多工作引入了近似方法来实现有效的解决方案。另一方面，K-D Bonsai提出了一小组在CPU内实现的新指令，同时保证基准准确性。这种基本差异使K-D Bonsai成为当今系统中可编程且易于采用的解决方案。就性能而言，我们的解决方案相对较为适中，但在面积和功耗方面比K-D Bonsai低几个数量级。然而，我们的（解）压缩方案旨在为叶节点处理提供更好的数据提取，这是先前工作未开发的一种正交概念。这里介绍的技术也适用于CPU以外的平台，因为它只取决于算法（半径搜索）和输入（点云）。

7 结论

在这项工作中，我们提出了K-D Bonsai，这是一种改进现代自动驾驶中点云处理算法中关键操作——k-d树半径搜索中叶节点处理的新方法。K-D Bonsai通过一种（解）压缩方案来减少内存操作，利用k-d树叶节点中点的值相似性和精度降低容忍度，而不会损害基准准确性。与先前依赖于离线加速器的工作不同，K-D Bonsai以ISA扩展（Bonsai扩展）的形式在乱序CPU中实现，并在全系统模式下使用用于自动驾驶的最先进软件在周期准确的模拟器中进行验证。我们的解决方案K-D Bonsai在降低端到端延迟和能量消耗方面非常高效，同时在面积和功耗方面产生最小的开销。此外，它只需要对通用CPU进行增量硬件修改，并且对程序员来说易于使用（在PCL中设置一个标志），因此是下一代自动驾驶系统的简单解决方案。