中国科学技术大学

本科毕业论文

针对大型数据集的排序算法的硬件加速框架搜索及其应用

作者姓名：	魏博逸
学　　号：	PB19020591
专　　业：	应用物理学
导师姓名：	金西 副教授
完成时间：	2023 年 5 月 9 日

摘　要

大数据集排序在如今的很多领域如光线追踪、碰撞检测等都有广泛的应用。然而，由于 CPU 平台的限制，大部分的排序算法的并行度并不能很好的挖掘，从而造成计算时间的指数上升。因此，对其进行加速具有很高的应用价值。近年来，对于该问题的研究一直在进行，但是绝大多数仅仅针对一个算法来进行讨论和优化，很少有能够涵盖大部分排序算法并对它们进行性能评估和筛选的工作。实际上，要找到最适合大数据集排序的算法，我们必须对多种排序算法及其组合进行评估和讨论，才能够得到最优的架构，这也是本工作的期望实现的目标。

根据软硬件协同设计的思想，如果要加速一个算法，我们可以从两个方面进行思考。一个是软件层面，我们可以设计“硬件友好”的算法，即在实现更高性能的同时，使用更少的计算资源；另一个是硬件层面，设计更加合理的硬件架构，从而能够更加适合算法的数据访问调用特性，提高并行度，从而提升性能。基于这种思想，本工作首先从软件层面调研各种算法（冒泡排序、基数排序、归并排序等）的优劣以及硬件友好度，并且基于现有的工作对各个算法进一步的优化以提升其硬件友好度；同时从硬件层面上为各种算法设计硬件加速框架，在有限资源下尽可能的提高并行度和吞吐率。最后以 FPGA 作为验证平台，通过对各个硬件加速框架进行测试和评估，得出了最优的排序算法及其硬件加速架构。最后，作为应用，我们将我们的加速器运用在了八叉树构建过程中。

本课题的另外一个亮点在于其是基于 Vitis HLS 工具进行设计的。相较于使用 Verilog HDL，HLS 直接通过 C 程序编译出 RTL 电路，通过使用 directives 可以很方便的以相同的策略实现各种硬件优化，同时也能更加直观的查看硬件资源的使用情况。在完成 C 语言文件的开发以后，HLS 会编译出相应的 ip 文件，我们在最后使用 Vitis 工具导入相应的 ip，实现了最后的仿真和上板操作。

我们将上述最优架构部署在了 Xilinx Alveo U280 加速卡上，相较于部署在 Intel Core i7-11800H CPU 平台上的软件算法，我们的硬件加速架构可以相对 C 语言平台有大约 4 倍的提升，相对于 Python 语言平台有 163 倍的提升。

我们的项目代码已经开源到 GitHub 当中: https://github.com/boyiwei/Sort-Acceleration

关键词：排序算法，硬件加速设计，现场可编程逻辑门阵列，高层次综合

ABSTRACT

Sorting for large datasets is widely used in many fields today. However, the parallelism of most of the sorting algorithms is not well exploited due to the limitation of CPU architecture, which causes an exponential increase in computation time. Though several research on this problem has been conducted, most of them only discussed one algorithm. Few works can cover most of the algorithms and evaluate their performance. In fact, to find the most suitable algorithm, we must evaluate multiple algorithms and their combinations in order to obtain the optimal architecture, which is also our goal.

According to the idea of hardware-software co-design, if we want to accelerate an algorithm, we can optimize from the software level and from hardware level. Based on this idea, we first investigated the advantages and disadvantages of various algorithms and their hardware friendliness from the software level, and further optimizes them to improve its hardware friendliness. Meanwhile, we designed hardware accelerators for them to improve the parallelism with limited resources. Finally, the best sorting algorithm and its hardware accelerator are selected by evaluating each hardware accelerator's performance on FPGA. As an application, we applied our accelerator to the Octree building process.

Another highlight of this project is that it is designed based on the Vitis HLS. Compared to Verilog HDL, HLS compiles RTL circuits directly from C programs. By using directives it can easily implement various hardware optimizations with the same strategy, and it is also easier see the utilization ratio of hardware resources. After finishing the C-program file, HLS compiles the corresponding ip file, and we use Vitis to import the corresponding ip in the end for the final simulation and on-board operation.

We implemented our architecture on Xilinx Alveo U280 Acceleration Card. Compared to the software algorithm deployed on an Intel Core i7-11800H CPU platform, our hardware-accelerated architecture can provide approximately 4× improvement relative to the C platform and 163× improvement relative to the Python platform.

Key Words: Sorting Algorithms, Hardware Acceleration Design, FPGA, HLS

目 录

符 号 说 明

英文缩写	英文全称	中文释义
FPGA	Field Programmable Gate Arrays	现场可编程逻辑门阵列
HLS	High Level Synthesis	高层次综合
DRAM	Dynamic Random Access Memory	动态随机存储器
HBM	High Bandwidth Memory	高带宽存储器
LUT	Look Up Table	查找表
DSP	Digital Signal Processor	数字信号处理器
II	Initiation Interval	启动时间间隔

第一章　研究背景

本章我们将系统的分析本项目的研究背景以及相关知识，并且分析现有工作已经取得的成果与不足。本章将从排序算法简介、软硬件协同设计综述、FPGA与高层次综合综述、相关研究等四个方面进行介绍。

第一节　排序算法简介

排序算法一直是一个热门的研究问题，即使其问题表述非常简单，但是人们一直期盼能够找到更加高效的排序算法。早在 1951 年，Betty Holberton 就开始着力解决排序算法问题；在 1956 年，Howard 提出了冒泡排序算法[1]。在此之后，新算法层出不穷，有些算法如 Timsort[2], Library sort[3]等均已经得到了非常广泛的应用。

目前，对于小型数组的快速排序（尽可能少的比较和交换次数）仍然是一个值得研究的问题，同时，在并行系统上如何能够尽可能的提高排序算法的效率，也是一个值得讨论的话题。本课题即希望通过一些算法和架构的优化设计，得到一些解决此问题的启发。

第二节　软硬件协同设计综述

软硬件协同设计是指通过硬件和软件的同时设计，利用它们的协同作用来实现系统级目标[4]。软硬件协同设计是一个比较宽泛的概念，但是其核心思想主要为软件层面和硬件层面上的协同优化，从而达到更好的性能。

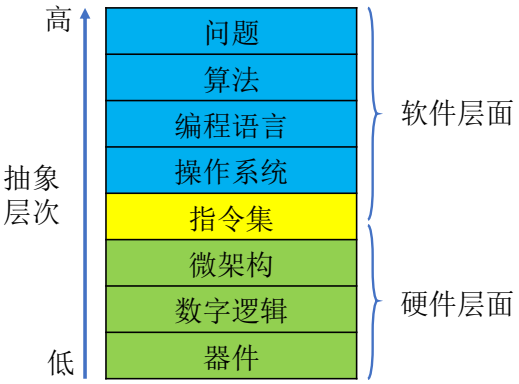

图 1.1　计算机系统的抽象层次示意

对于计算机系统，我们可以使用不同的抽象层级来对其进行描述，如图1.1所

示。对于研究人员而言，如果想要提升计算机的性能，我们可以从不同的抽象层次来进行优化：从硬件层面上来说，可以通过使用新兴器件取代传统 CMOS 器件，设计新的微架构等方式来提升芯片的性能；从软件层面上来说，可以通过采用新的高效的指令集，采用更加优秀的操作系统、编程语言以及算法来提升计算机解决问题的能力。而软硬件协同设计，则是从“跨抽象层次”的角度来看待问题：在优化软件算法的同时考虑到相关的硬件资源问题，即设计“硬件友好型算法”；在设计硬件架构的同时考虑到对应算法的数据流特征，即设计“特定领域的加速器”。二者相互兼顾的好处就是软件层面与硬件层面能够高度融合，从而达到相较于传统算法以及通用处理器（CPU 等）更加优异的性能。

第三节　FPGA 与高层次综合 (HLS) 综述

一、FPGA 简介

现场可编程逻辑门阵列（Field Programmable Gate Arrays），即 FPGA，是一种半定制的集成电路芯片。它可以通过编程来实现几乎任意一种数字逻辑或系统[5]。与传统的 ASIC（Application Specific Integrated Circuit）相比，FPGA 可以在不改变硬件电路的情况下进行功能修改，因此更加灵活。

FPGA 由可编程逻辑单元（PL）和可编程输入/输出单元（IO）组成。PL 是 FPGA 的核心部件，它由查找表（LUT）、触发器、多路选择器等多种逻辑门组成，可以通过编程实现不同的逻辑功能。IO 单元则提供了 FPGA 与外部系统的接口，包括数字信号、模拟信号、时钟信号等。FPGA 整体架构如图1.2所示。

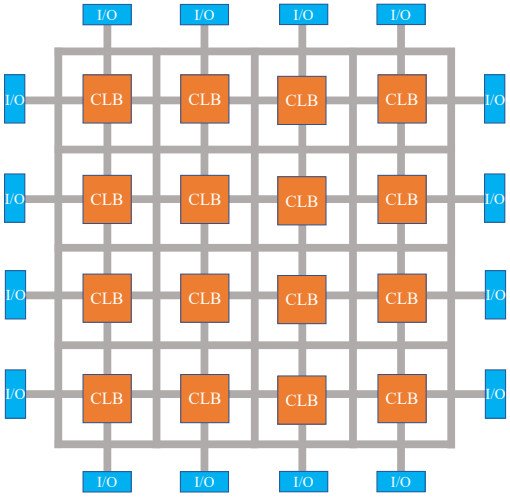

图 1.2　FPGA 架构示意图。CLB 为可编程逻辑功能块（Configurable Logic Blocks）。

FPGA 具有灵活性高、开发周期短、成本低等优点，因此被广泛应用于数字信号处理、控制系统、汽车工业等领域。时至今日，FPGA 已经成为数字系统设计中不可或缺的一部分。随着 FPGA 技术的不断发展，其性能和功能都在不断提升，如今的 FPGA 不仅支持更高的逻辑密度和更快的工作频率，还可以集成更多的外设和处理器核心。同时，FPGA 也面临着新的挑战，如如何提高功耗效率、如何实现更高的可靠性等问题。

二、高层次综合简介

高层次综合（HLS）是一种用于数字系统设计的工具。HLS 允许设计人员使用高级编程语言（如 C / C ++）来描述数字系统，并自动将其转换为硬件电路。与传统的硬件设计方法相比，HLS 具有更快的设计周期和更高的生产效率。

HLS 的主要优点之一是可以将抽象的高级语言代码转换为硬件电路，从而减少了设计人员需要了解的硬件细节。这使得设计人员可以更加专注于系统级的设计，而不必担心底层的硬件细节。此外，由于 HLS 工具可以自动优化设计，因此可以更加轻松的获得拥有更高性能和更低功耗的解决方案。

HLS 的发展历程可以追溯到 20 世纪 80 年代。当时，人们开始使用高级语言来描述数字系统，但由于计算机性能的限制，这种方法并不实用。随着计算机性能的提高，HLS 工具开始被广泛应用。现在，许多公司和组织都提供了 HLS 工具，如 Xilinx、Altera、Cadence 等。[6] [7]

随着数字系统的不断发展和应用需求的不断提高，HLS 工具也在不断发展和完善。目前，HLS 工具已经具有了更高的性能和更好的可靠性，使得设计人员可以更加高效地进行数字系统设计。此外，HLS 工具还可以与其他工具和平台集成，如 EDA、IP、FPGA 等，以进一步提高设计效率和性能。

HLS 的未来发展将面临一些挑战。首先，相较于直接编写 Verilog，HLS 并不能给出最优的硬件架构，即，其给出的方案仍然具有优化空间。另外，在综合编译大型项目的时候，HLS 软件的性能仍然需要进一步优化，以 Vitis HLS 为例，在综合过程中仍然只能利用 CPU 单核进行处理，这大大限制了 HLS 工具的性能。[8]

总的来说，HLS 作为一种数字系统设计工具，具有广泛的应用前景和重要的地位。未来，随着 HLS 工具的不断发展和应用领域的不断拓展，它将继续为数字系统设计和其他领域的发展做出贡献。

第四节　排序算法加速的相关研究

许多研究人员已经对排序算法的硬件加速进行了相关的研究。但是他们的侧重点各不相同。总的来说，他们的研究工作可以通过以下几点来进行分类：

1. 选择研究的排序算法，如归并排序，堆排序，基数排序等；
2. 使用数据集的大小；
3. 验证平台，如 CPU、GPU、FPGA 等。

具体来说，Zurek 等[9]比较了不同的并行排序算法在 CPU 平台和 GPU 平台上的性能表现；Jan 等[10]对三种并行排序算法：奇偶排序、等级排序和双调排序算法在 CPU 和不同 GPU 架构上的性能进行了比较分析，同时还实现了一种名为 min-max 蝴蝶网络的新型并行算法，用于在大型数据集中查找最小值和最大值；Konstantinos 等[11]在 FPGA 平台上测试了归并排序的性能；Chen 等[12]对提出了一种新的基于 FPGA 平台的双调排序算法，使其能够以更高能效、更低延迟运行；Matai 等[13]对插入排序进行了优化，并且使用 HLS 工具对其设计了硬件加速器；Purnomo 等[14]对冒泡排序进行了并行优化，并在 FPGA 上面进行了部署。Romanous 等[15]虽然声称实现了在 FPGA 平台上实现了对基数排序的加速，但是其方案的具体细节并没有公布；Qiao 等[16]在 HBM-FPGA 上改进了归并排序，基于存内计算对其实现了加速；Jayaraman 等[17]也基于 HBM-FPGA 加速了列排序算法；根据上面提供的角度，我们可以对有关研究进行分类，如表1.1所示。

表 1.1　相关工作总结

研究	排序算法	平台	是否通过 HLS
Zurek 等[9]	归并排序、双调排序	CPU、GPU	否
Jan 等[10]	奇偶排序、等级排序、双调排序	CPU、GPU	否
Kostantinos 等[11]	归并排序	FPGA	是
Chen 等[12]	双调排序	FPGA	否
Matai 等[13]	插入排序	FPGA	是
Purnomo 等[14]	冒泡排序	FPGA	否
Romanous 等[15]	基数排序	FPGA	未知
Qiao 等[16]	归并排序	HBM-FPGA	否
Jataraman 等[17]	列排序	HBM-FPGA	是

上述工作虽然对排序算法的优化以及硬件实现进行了探讨，但是他们仍然存在着一定的缺点：

1. 部分工作仅仅是在算法层面进行了优化，在硬件架构上并没有讨论，如上述基于 CPU、GPU 平台的工作[9-10]；
2. 在 FPGA 平台上的工作仅仅讨论了部分排序算法的优化，且没有对研究成

果进行应用，如[11-14]；

3. 大部分工作并没有发挥 HLS 的优势，而是使用传统的方式（如 Verilog）等进行设计，如[12,14]；

基于以上的原因，我们的工作有以下优点：

1. 充分体现软硬件协同设计的思想，即从软件算法角度和硬件加速角度进行了协同优化；
2. 探究了多种排序算法，同时以尽可能充分利用硬件资源为指导原则，充分发掘各个算法的效率和并行性；
3. 通过 HLS 来设计硬件加速架构，更加直观高效。
4. 将设计的硬件加速架构应用到了解决实际问题当中，在这里我们以八叉树构建过程举例，直观地展示了加速器的加速效率。

第二章　排序算法优化

第一节　排序算法的性能分析

一、传统十种排序算法的性能分析

下面对十种传统的排序算法进行介绍和分析，主要的分析角度为时间复杂度、空间复杂度以及硬件友好程度。

1. 冒泡排序

冒泡排序的基本思想在于对数组重复遍历，逐个比较数组相邻元素的值并进行交换；当遍历完一遍以后，没有进行交换操作，即说明排序完成。对于冒泡排序来说，其时间复杂度为 $O(n^2)$，空间复杂度为 $O(1)$，就其本身而言，并没有并行性；但是基于其的改进算法，如奇偶排序，可以应用于并行计算。

2. 选择排序

选择排序基本思想为：首先从未排序的数组中遍历，选出最小（大）的数放在排序数组的起始位置；再从剩余未排序的数组中选择最小（大）的数，放在已排序数组的末尾，以此往复，直到将未排序数组完全遍历。在实际操作过程当中，通常将数组的前 n 位作为有序数组，当进行第 $n+1$ 次遍历时，从第 $n+1$ 位元素开始，选出最小（大）的数，再将其与第 $n+1$ 个数进行交换，从而形成长度为 $n+1$ 的有效数组。选择排序的时间复杂度为 $O(n^2)$，空间复杂度为 $O(1)$。

3. 插入排序

插入排序的思想是通过对数组中已经完成排序的有序数列进行扫描，将未排序的数据插入到已排序序列当中。和冒泡排序和选择排序类似，插入排序的时间复杂度也是 $O(n^2)$，空间复杂度为 $O(1)$。

4. 堆排序

堆排序的主要思想是通过不断的构建大顶堆，并提取堆顶的根节点，实现对于数组的排序。堆排序的步骤为：首先将待排序数组构造成一个大顶堆，再将堆顶元素和堆底元素进行对调，此时末尾元素即为最大值。再将剩余的 $n-1$ 的元素重新构造成大顶堆，按照之前的步骤反复执行，即可得到一个有序数列。在堆排序中，初始构造堆的过程的时间复杂度为 $O(n)$，接下来交换堆顶堆底元素并重新构造堆的过程，每次操作的时间复杂度为 $O(\log n)$，重复 n 次，故全过程的时间复杂度为 $O(n+n\log n)=O(n\log n)$。

5. 归并排序

归并排序针对于两个有序数列进行排序。在两个有序数列中分别设置指针，初始时二者分别指向两个有序数组的第一个元素，比较两个元素，选择较小者放入合并空间中，并移动该元素所在序列的指针；重复上述操作，直到所有元素均被放入合并空间当中。对于归并排序来说，如果有 n 个数需要排序，则需要进行 $\log n$ 次归并操作，每次归并操作的时间复杂度为 $O(n)$，所以归并排序的时间复杂度为 $O(n\log n)$。归并排序的特点使其具有并行化的可能，对于每一层的归并排序，都可以将其并行化。关于归并排序的并行化我们会在后面详细阐述。

6. 快速排序

快速排序的思想在于每次排序时，从数列中挑出一个元素作为基准值，将小于基准值的数放在它的前面；将大于基准值的数放在它的后面，然后递归地将小于基准值的元素的子序列和大于基准值元素的子序列进行排序，从而最后形成有序数列。对于快速排序来说，其时间复杂度可以通过递归的方式求得。设快速排序的时间复杂度为 $T(n)$，则一次单独的排序步骤的需要进行 $O(n)$ 次操作，而剩下的时间由两个较小的子序列的排序时间复杂度决定，即 $2T(n/2)$，故可得关系式 $T(n) = O(n) + 2T(n/2)$，从而解得 $T(n) = O(n\log n)$。快速排序的空间复杂度为 $O(\log n)$。

7. 希尔排序

希尔排序通过以一定步长交换不相邻的元素进行排序，每次排序减小步长，直到最后的步长为 1。希尔排序可以看做改进版的插入排序，其最优时间复杂度通常为 $O(n\log^2 n)$，平均时间复杂度为 $O(n^{1.3\sim2})$，空间复杂度为 $O(1)$。

8. 计数排序

计数排序首先找出数组中最大和最小的元素，然后统计数组中每个值为 i 的元素出现的次数，存入新开辟的数组当中, 并且对每个值出现的次数进行统计，最后反向填充目标数组。由于计数排序并不是比较排序算法，所以其时间复杂度和空间复杂度都是为 $O(n+m)$，其中 m 为数组的最小值和最大值的差值。

9. 桶排序

桶排序可以看做是一种改进的计数排序算法，其通过将数组按照值的范围放在不同的桶中，再在每个桶的内部进行排序，所以其的时间复杂度为 $O(n+m)$，空间复杂度为 $O(m\times n)$，其中 m 为桶的数目。

10. 基数排序

基数排序的基本思想在于按顺序对数的每一位进行排序。如对于十进制数组排序，首先根据各个数的个位数的值，将其放在十个“桶”当中，然后依次取出完成一次排序；再根据各个数的十位数的值，再将其放入十个桶当中，依次取出，如此往复，直到完成对于最高位的排序，按顺序取出的数组即为完成排序的数组。由基数排序的算法可以看出，它的时间复杂度仅仅取决于数的位数以及数组长度。因此，其时间复杂度为 $O(k \times n)$，其中 k 为数的位数。

综上，各种排序算法的时间复杂度、空间复杂度如表2.1所示。

表 2.1　各种排序算法的时间复杂度、空间复杂度

排序算法	时间复杂度	空间复杂度
冒泡排序	$O(n^2)$	$O(1)$
选择排序	$O(n^2)$	$O(1)$
插入排序	$O(n^2)$	$O(1)$
堆排序	$O(n \log n)$	$O(1)$
归并排序	$O(n \log n)$	$O(n)$
快速排序	$O(n \log n)$	$O(\log n)$
希尔排序	$O(n \log^2 n)$	$O(1)$
计数排序[1]	$O(n+m)$	$O(n+m)$
桶排序[2]	$O(n+m)$	$O(m \times n)$
基数排序[3]	$O(k \times n)$	$O(k+n)$

[1] 计数排序中，m 代表数据的最大值减最小值。
[2] 桶排序中，m 代表桶的个数。
[3] 基数排序中，k 代表数值中的“数位”个数。

二、k 路归并算法

除了以上传统的排序算法，我们还想介绍一种特殊的排序算法——k 路归并算法。k 路归并算法指的是利用归并排序的思想，对 k 输入数列进行排序的算法。k 路排序算法有很多变种，接下来我们进行逐一介绍和分析。

1. 传统多路归并排序

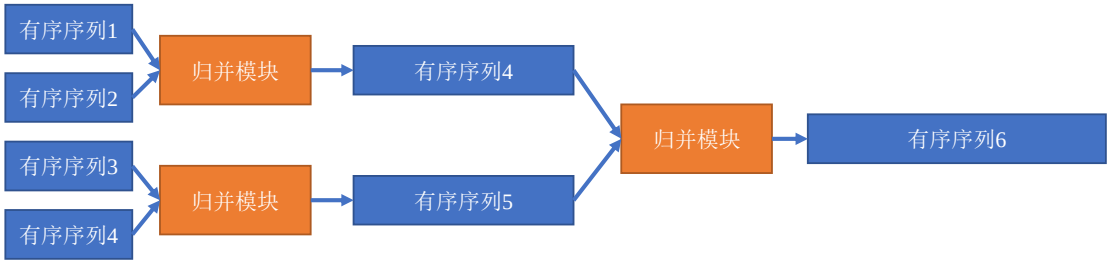

图 2.1　4 路归并排序示意。有序序列 6 即为最终输出有序序列

传统归并排序采用的 “Divide and Conquer”（分治法）进行排序，先将待排序数组分成等量子数组进行排序，分别形成有序数列。然后对有序数列进行排序

形成更长的有序数列，不断重复，直到待排序数组完全有序。对于传统的 k 路排序来说，我们可以直接利用归并排序的 “Conquer” 部分，即对 k 个有序数列进行两两排序，不断归并直到最后将 k 个有序数列归并成一个有序数列。传统 k 路排序的整体思路如图2.1所示。

2. 败（胜）者树归并

除了上述的传统归并排序，还有一种多路归并方法被广泛使用：败（胜）者树。败者树的思想在于首先将每个输入的有序序列的最小元素取出，根据其大小构建败者树。败者树的叶子结点（即 ls 数组）记录败者序列编号，使得较小数（胜者）能够不断爬升比较，直到根节点，此时败者树的根节点记录的序列编号的中的第一个数即为全输入序列中最小的数。将其取出放在输出有序数列第一位，在不断重复此过程，直到所有输入序列均已排序完成。败者树的工作过程如图2.2所示。在这个例子当中，第一次构建败者树的过程将输入序列 3 的第一个元素 6 取出，然后序列 3 中元素前进一位。第二次建树的过程将输入序列 1 的第一个元素 9 取出，作为输出有序序列的第二个元素，由此不断循环，即可得到五个输入序列的整体的有序数列。胜者树的思想和败者树类似，只是在建树过程中，叶子结点记录的是胜者的序列编号，使得败者能够不断上浮。胜者树构建出来的有序序列是逆序（从大到小）的。

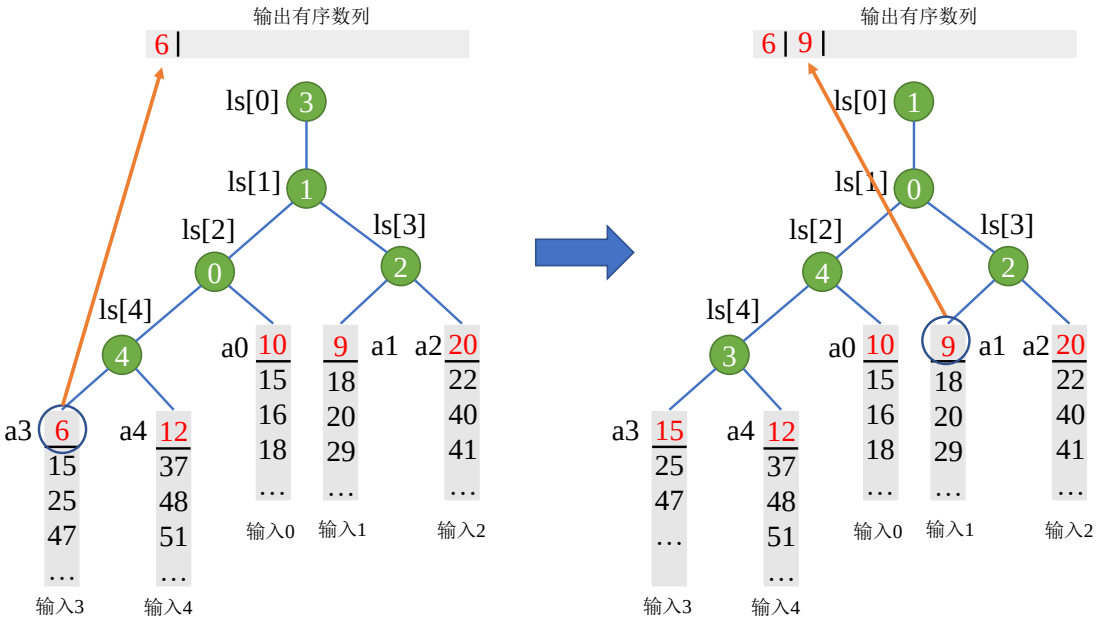

图 2.2　败者树示意图

第二节　针对大型数据集的排序算法所面临的问题

相对于小型数据集，要想针对面向大型数据集的排序算法进行加速，我们会遇到如下问题：

1. 对于大型数据集来说，FPGA 的片上资源无法一次性存储所有的数据，这就要求硬件加速架构必须分块从 DRAM 当中加载数据，这也会导致同时排序的过程中不可能一次性全部排序完成，只能先将每批次序列的数据排成有序数组，再对这些有序数组进行排序。
2. 对于大型数据集来说，即使是分批次载入数据，如果每个批次的数据量仍然较大，且硬件上面没有充分流水化/并行化的话，可能会造成严重的数据依赖，即排序过程中的每次循环过程都会很长，且直到该轮循环完成之前，下一轮循环都无法开始。这就会导致极高的延迟，从而大大影响硬件性能。
3. 对于大型数据集来说，算法本身的时间复杂度会极大地影响其效率，因此，我们需要比较不同时间复杂度的增长趋势，如图2.3所示。图中，横坐标表示待排序数的个数，纵坐标表示的事相对的时间复杂度，该图显示了 $O(n^2)$，$O(n\log n)$，$O(n\log^2 n)$, $O(kn)$ 时间复杂度的增长趋势。从图中可以看到，当 n 足够大时，$O(n\log n)$，$O(kn)$ 的增长速度明显要慢于其他的函数，因此，我们可以仅选择时间复杂度为 $O(n\log n)$ 和 $O(kn)$ 的算法，即堆排序、归并排序、快速排序以及基数排序。至于计数排序和桶排序，由于其时间复杂度和空间复杂度都取决于数据的最大值和最小值之差，当要排序的数据的最大值和最小值之差很大时（对于我们讨论的情形，待排序数据位宽为 32），其所需的时间和存储空间都是巨大的，因此我们不将其放入讨论范围。

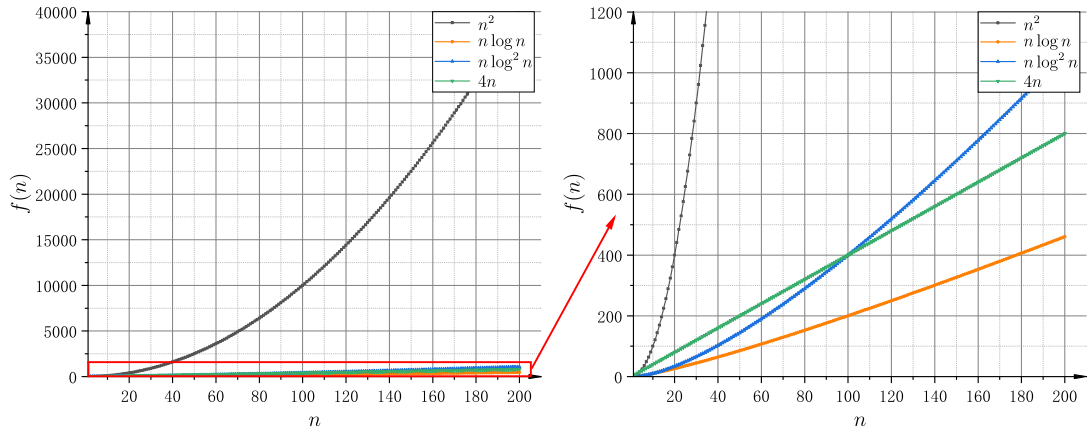

图 2.3　各个算法时间复杂度增长趋势

第三节　面向硬件的排序算法优化策略

大部分排序算法都需要用到嵌套循环语句。因此从硬件的角度，对于循环的优化必不可少。同时，HLS 提供了丰富的硬件优化策略来降低延迟以及提升吞吐量。下面我们选取几个常用的硬件优化策略进行介绍。

一、对于循环的优化策略

1. 循环分块（Loop Tiling）

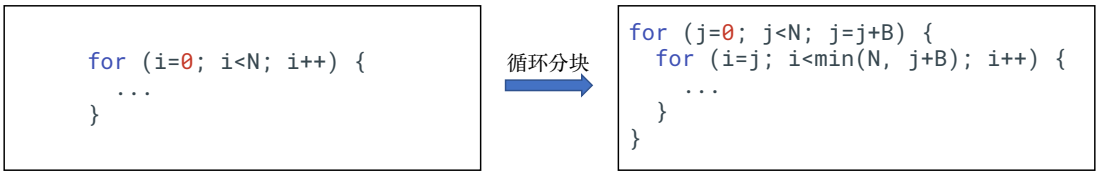

图 2.4　循环分块算法示意。本图将原循环分割成了长度为 B 的循环小块。

循环分块通过将“迭代空间”分割成更小的块，来确保循环中使用的数据被重复利用之前能够一直保留在 cache 中，防止因为单次循环过长导致要复用的数据被移出 cache。其算法的主要思想如图2.4所示。在 HLS 当中，循环分块可以直接在 C 语言文件中进行修改，而不需要额外插入 HLS pragma 语句。

2. 循环交换（Loop Interchange）

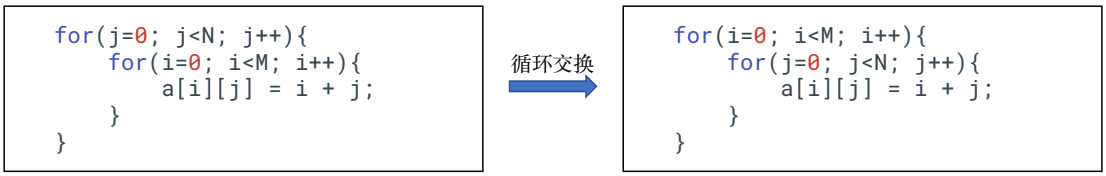

图 2.5　循环交换算法示意。左边是以行优先顺序遍历，右边是以列优先顺序遍历

循环交换即通过交换内外循环的顺序，使得代码具有更好的空间局部性。因为在 C 语言中，数据是以行优先的形式存储在 cache 当中的，所以在设计嵌套循环的时候，应该尽量以行优先遍历数据，从而达到更好的空间局部性。

3. 循环合并（Loop Fusion / Loop Merge）与循环分割（Loop Fission）

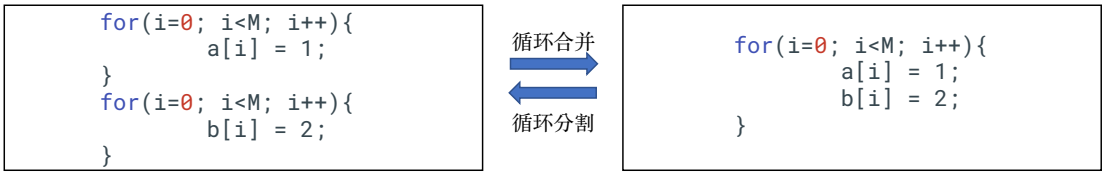

图 2.6　循环合并与循环分割算法示意

循环合并即将多个独立的循环过程合并到一个循环当中，而循环分裂则是相反的过程，将一个循环分割成多个独立的循环过程。循环合并通常是为了减

少在多个循环中数据被重复利用的情形下，cache miss 的概率，提升空间局部性，同时降低了循环控制结构的开销，消除可能存在的冗余分配，从而提升性能。在 HLS 当中，可以在循环语句之前添加`#pragma HLS loop_merge`来实现循环合并，并尽可能并行化。

4. 循环展开（Loop Unrolling）

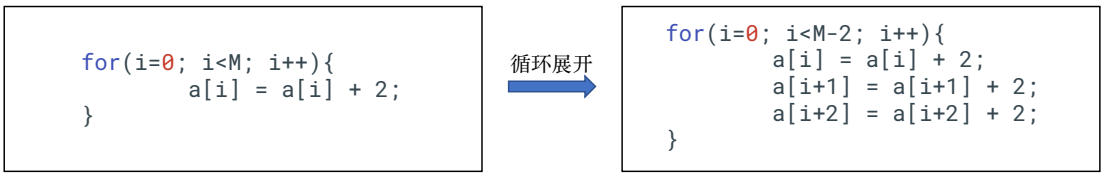

图 2.7　循环展开算法示意

循环展开通过提升每次循环中执行的操作数量来减少循环次数，降低分支预测的错误概率。同时，在 HLS 当中，通过循环展开，如果没有数据相关，每个循环内的操作可以并行执行，从而提升了并行度以及处理速度。在 HLS 当中，可以在需要展开的循环前面添加`#pragma HLS unroll`来将循环展开。

5. 循环扁平化（Loop Flattening）

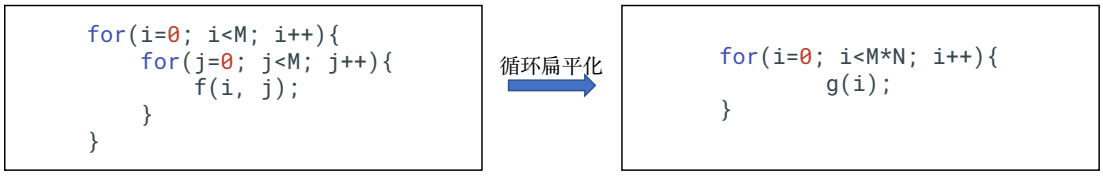

图 2.8　循环扁平化算法示意

循环扁平化将多层嵌套循环转为单层循环。从而能够更好的降低延迟和吞吐率。在 HLS 当中，可以在循环前面加上`#pragma HLS loop_flatten`来将循环扁平化。

二、流水线

流水线是一种常用的提升架构并行度以及吞吐率的措施，其主要思想在于将程序分解成几个独立的阶段，每个阶段互不干扰，每个阶段对应一个硬件模块来处理。以最简单的五段流水线举例，它将计算机执行指令的过程分为了取指令、译码、执行、访存和写回五个过程，当流水线被填满时，所有的阶段都在执行不同的指令，而不会互相干扰，从而达到了降低硬件资源开销，提升处理器的时钟频率和性能。Vitis HLS 提供了多细粒度的流水线选项，下面我们取最常用的指令级别流水线和任务级别流水线进行介绍。

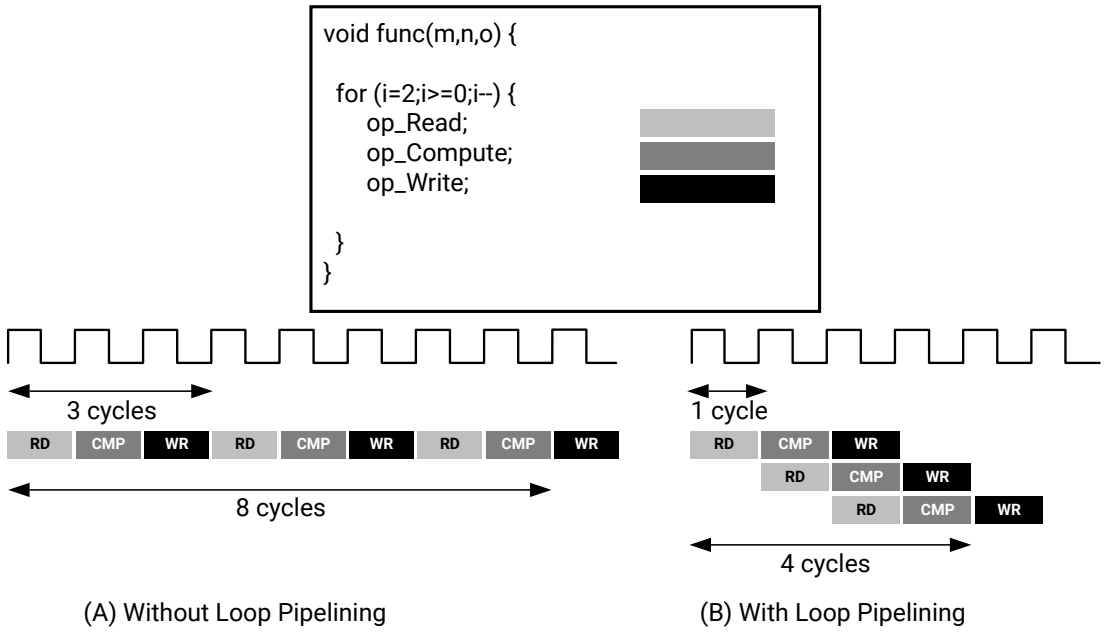

图 2.9　指令级别流水线示意图

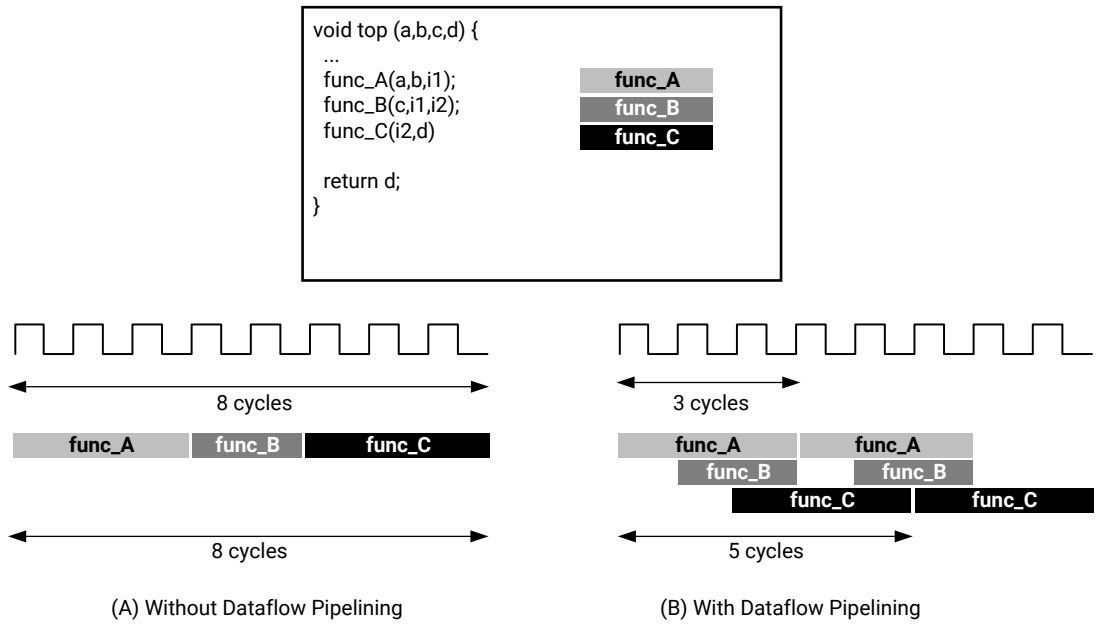

图 2.10　任务级别流水线示意图

1. 指令级别流水线

在 HLS 当中，我们可以使用`#pragma HLS pipeline`来实现指令级别的流水化，如图2.9[①]所示。在该图所示的例子当中，我们将每个指令的执行过程都分为了 RD, CMP, WR 三个阶段，加入`#pragma HLS pipeline`以后，我们可以将这三个指令的执行阶段流水化，从而将原本需要 9 个时钟周期完成的任务减少到 5 个时钟周期。

2. 任务级别流水线

任务级别的流水线指的架构允许不同的任务同时进行，是一种细粒度更大的流水线，在 HLS 当中，我们使用`#pragma HLS dataflow`来实现该级别的流水线，如图2.10[①] 所示。在图中所示的例子当中，我们并没有考虑每个函数内部需要执行多少条指令，但是 HLS 会自动分析是否有可以复用的模块，从而在最大程度上实现多任务流水化。在本例中，加入`#pragma HLS dataflow`以后，我们可以将原本需要 8 个时钟周期完成的任务减少到 5 个时钟周期。

三、数组优化

1. 数组分区（Array Partitioning）

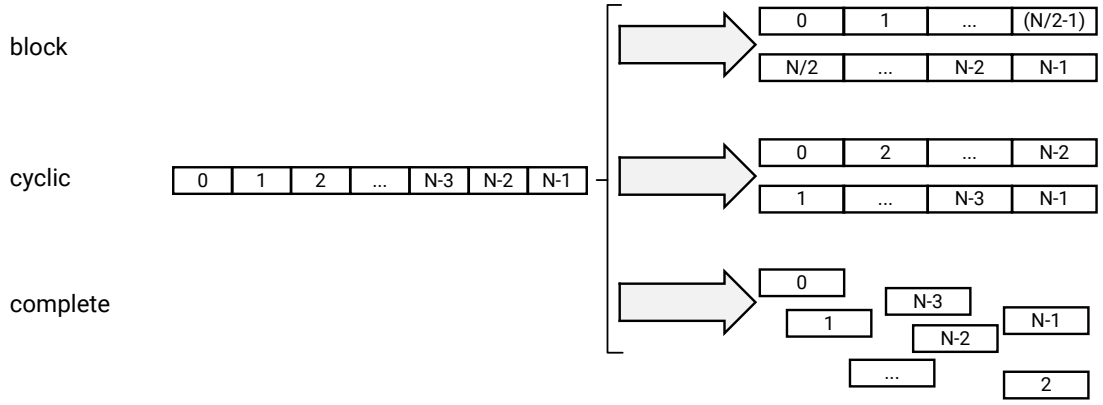

图 2.11　三种数组分区方式示意

在 Vitis HLS 当中，如果定义一个数组，HLS 会默认将其编译成一个双口 RAM，允许一个线程写一个线程读。但如果想对该数组进行同时写或者同时读等并行的操作，就会受到极大的限制。数组分区的功能就是将一个数组分割成多个小块数组，在编译时体现为将一个 RAM 块分割成多个小的 RAM 块或者寄存器，从而实现对数组的并行操作。

Vitis HLS 提供了三种数组分区的方式：block, cyclic, complete。block 将原数

[①]Xilinx Inc. Vitis High-Level Synthesis User Guide (UG1399), 2022.2

组分割成多个相同大小的块，在原数组中的相邻元素在块内也是相邻的；cyclic 也将原数组分割成相同大小的块，但是元素为交错排列；complete 将原数组完全打散，变成独立的元素，相当于将 RAM 形式转换成了寄存器形式。三种分区方式如图2.11所示。

2. 数组重组（Array Reshaping）

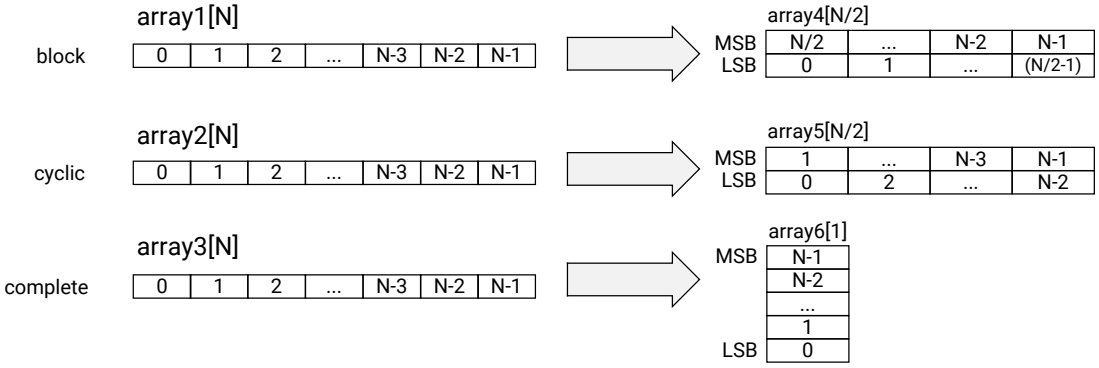

图 2.12　三种数组重组方式示意

数组重组旨在能够在单个时钟周期内能够读取更多的数据。同时减少 BRAM 的使用数量以及并行读取数据[18]。和数组分区类似，数组重组也有 block, cyclic 和 complete 三种方式，如图2.12所示。

第三章　硬件加速架构设计

第一节　硬件加速架构的搜索空间

如第一章第二节所述，在大型数据集的情形，我们从时间复杂度的角度出发，可以考虑的算法只能有堆排序、归并排序、快速排序以及基数排序。所以我们的硬件加速架构的搜索空间，也应该是基于这些算法来进行设计的。

在时间复杂度的同时，我们还需要考虑到各个算法的硬件友好程度。在这四种算法当中：堆排序可以被视为一种分治算法，它可以将一个大的输入序列分成多个较小的子序列并分别排序，最后将这些子序列合并成一个有序序列。这使得堆排序很容易被并行化，不同的处理单元可以同时对不同的子序列进行排序，而后将它们合并在一起；归并排序的并行化也易于实现。它将一个大的输入序列分成多个较小的子序列，并对这些子序列分别进行排序，然后将它们合并在一起。这种特性使得其能够很容易的被并行处理；基数排序是一种非比较排序算法，每次循环中根据元素在每个位上的值来进行排序，这个过程是完全独立的，可以并行执行，因此具有很高的硬件友好度；而对快速排序而言，由于其本质上是基于比较的算法，涉及到大量的非相邻元素的比较和交换，而为了进行比较和交换操作，需要访问数组中的随机元素，而这种随机访问在硬件中需要通过内存地址映射或多级缓存等技术来实现，增加了设计的复杂性和开销。此外，快速排序的递归结构也不利于在硬件中实现，因为递归需要消耗大量的存储器和控制器资源。而与此相对的，堆排序、归并排序和基数排序则都具有很好的局部性，即访存序列元素的时，可以访问一段连续的存储块，因此他们属于硬件友好型的算法，更加利于硬件加速。

基于上述的分析和讨论，我们的搜索空间当中只包含了归并排序、堆排序以及快速排序。但是这并不意味着硬件搜索空间中仅仅包含了三种选项：由于归并排序的特殊性，理论上来说，我们只需要生成多个小的有序序列，也可以通过之前所说的 k 路归并排序的方式进行排序；同时，基数排序根据进制的不同，也可以分为 2 进制、8 进制、16 进制的基数排序。我们的硬件架构搜索空间如表3.1所示，一共 13 个备选方案。

表 3.1　硬件架构搜索空间

混合算法		单一算法
2 进制基数排序 + 传统 k 路归并 8 进制基数排序 + 传统 k 路归并 16 进制基数排序 + 传统 k 路归并 堆排序 + 传统 k 路归并	2 进制基数排序 + 败者树归并 8 进制基数排序 + 败者树归并 16 进制基数排序 + 败者树归并 堆排序 + 败者树归并	归并排序 2 进制基数排序 8 进制基数排序 16 进制基数排序 堆排序

第二节　硬件加速架构的设计

根据第一节所阐述的搜索空间，我们可以发现我们的硬件架构可以由几个模块构成：基数排序模块、归并排序模块、堆排序模块，以及一些 IO 控制模块等。将这些模块进行组合，即可组成上述硬件架构搜索空间中的各种架构，实现对其性能的评估。

一、基数排序模块设计

1. 传统基数排序算法在 HLS 实现中面临的问题

对于传统的排序算法来说，在每一个循环中，我们按照每个数对应数位的值将其放入到“桶”中，再按照顺序将其取出，作为新的待排序数组。在取出过程中，要保证每个“桶”中的数取完之后就开始从下一个桶中取数，我们需要给每一个桶分配一个指针：每输入一个数指针的值加 1，每输出一个数指针的值减 1，当指针的值为 0 时，则说明该桶中的数已经全部取出，基本算法如图3.1所示：

```
input_bucket:
for (int j = 0; j < batch_size; j++) {
        <get ith_radix>
        <input number to bucket[ith_radix]>
        bucket_pointer[ith_radix] += 1;
    }
output_bucket:                              变上限循环
    for (int l = 0; l < 16; l++) {
        for(int m=0; m<bucket_pointer[l]; m++){
            sorted_data[k] = bucket[l][m];
            k = k + 1;
        }
    }
```

图 3.1　传统基数排序算法示意。红圈标出的即为变上限循环

然而，在这个过程中，会存在着若干问题，从而影响到硬件的执行效率：

1. 为了考虑最坏情况（所有数的第 n 位数都是一样的），每一个“桶”的大小都必须和待排序数组大小相同，这就造成了极大的空间浪费，对于 BRAM

来说，其本身存储空间就较小，当待排序数组增大时，这种资源浪费将急剧增加，让可以排序数组的数量的上限大大降低；

2. 在“桶”的输出循环当中，由于每个桶的指针值各不相同，所以该循环是一个变上限循环。对于 HLS 来说，如果使用变上限循环，则无法通过`#pragma HLS pipeline`或者`#pragma HLS dataflow`来实现硬件架构的流水化，这对于提升吞吐率和减低延迟来说是非常不利的；
3. 由于基数排序算法本身的限制，在“桶”输入和“桶”输出循环当中存在“写后读”的数据相关，而这个循环周期会随着输入数组的增大而增大，当输入数组特别大时，将会造成很长的等待和空转周期。

2. 对应的优化措施

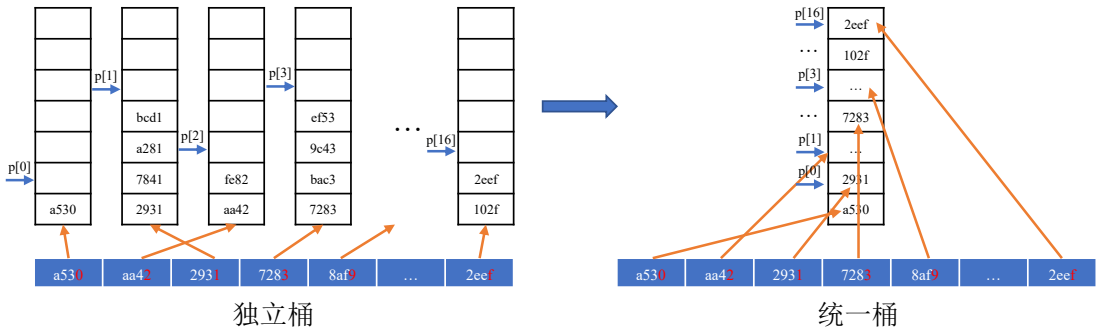

图 3.2　统一桶基数排序示意图

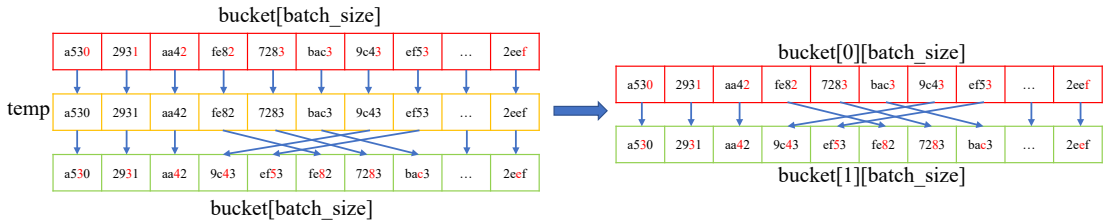

图 3.3　Ping-Pong buffer 示意图

针对上述提出的问题，我们创新性的给出如下优化措施：

1. 采用统一“桶”的方式，通过多个指针来分隔不同数位值的数。具体而言，如果采用 n 进制基数排序，则统一“桶”中有 2^n 个指针。通过使用统一“桶”，我们能够大大减少 BRAM 的使用量，使得片上大数据集排序成为可能。其基本思想如图3.2所示。
2. 在采用统一“桶”方式时，由于我们输出“桶”过程不需要考虑每个部分的指针位置，而是直接将“桶”中元素顺序读出，因此不存在变上限循环，这也使得我们能够通过`#pragma HLS pipeline`实现硬件流水化。
3. 采用 Ping-Pong buffer 技术，即设置两个统一“桶”，在一个循环中，一个

桶作为输出，一个桶作为输入。具体来说，在奇数循环中，排序模块从 0 号桶顺序读取数据，根据不同数的基数大小放入到 1 号桶当中；在该循环的下一个循环当中（即偶数循环），我们直接从 1 号桶中顺序读取数据，并按照不同数的基数大小放入到 0 号桶当中。这样做的一个好处是省去了中间数组的写入与读出，而是两个桶直接进行数据的输入和输出，从而大大降低了延迟和数据相关问题，使得硬件架构能够进一步流水化。其基本思想如图3.3所示。

需要注意的是，我们的基数排序模块是灵活的，可以适用于任意 2^n 进制数的排序。在接下来的实验当中，我们仅考虑常用的 2 进制、8 进制、16 进制排序。

二、归并排序模块设计

1. 传统归并排序模块设计

传统归并排序模块相对简单。对于两路有序序列输入，分别比较它们首位元素，将较小者选出放入输出序列当中，并将指针向后移位，最后重复上面的操作。需要注意的是，在最后需要判断数组元素是否已经排空，如果已经排空而另外一个数组仍有数，则将另外数组中的数有序输出即可。

2. 单一归并排序模块设计

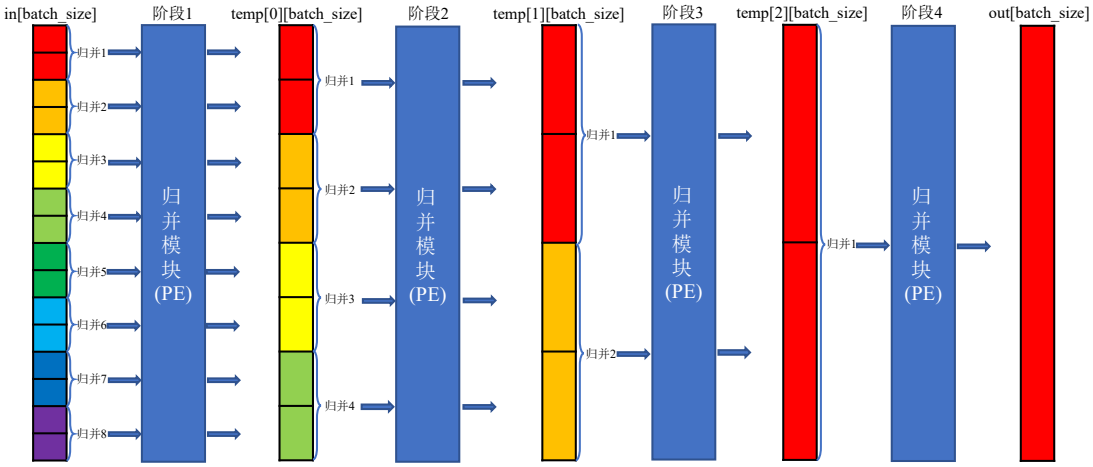

图 3.4　单一归并排序算法示意

如果我们想通过单一的归并排序模块来实现对一个无序序列的排序，我们还需要对归并排序模块进行修改以适应硬件特性。单一归并排序可以理解为首先将数组分割成多个长度为 2 的小块，分别对它们两两归并，然后增加步长到 4，再进行两两归并，最后不断重复的过程。如图3.4所示。但是不同于二输入归并模块，在单一归并模块中，我们只有一个输入，并存放在 BRAM 当中。由于 BRAM

最多只能允许同时两路访问，所以并不能完全并行化。因此我们尝试在每个阶段内部实现排序流水线，同时在不同阶段中间应用`#pragma HLS dataflow`。在实例化暂存数组`temp`时，采用`#pragma HLS array_partition`对其进行分割，将其实例成多个 BRAM 块，从而实现并行访问。这样下来，在综合的时候，硬件能够达到 II=1 的流水线，性能得到极大的提升。

3. 败者树归并模块设计

败者树算法的主要思想如图2.2所示。通过 HLS 编写算法时，由于其涉及到数据交换的操作，我们可以将败者树中的元素定义一个新的数据结构 Item，里面包含了该元素所在的败者树位置以及元素数值的大小。同时，在循环中，我们可以插入`#pragma HLS pipeline`来将其流水线化，从而达到降低延迟、提升性能的目的。在进行败者树归并的时候，我们需要注意在后期可能会有部分输入数组已经归并完成的情况，这个时候我们需要判断数组指针是否已经超过 batch_size，如果超过的话，则将该叶子结点设为 int 数据类型能够表达的最大值 (2147483647)，从而确保其不会参与到败者树归并过程当中。

三、堆排序模块设计

堆排序模块主要分为三个部分：构建大顶堆、交换/推出元素、重构大顶堆。我们可以分别定义交换元素函数 swap，构建大顶堆函数 maxHeapify；在重构操作中，可以复用构建大顶堆函数。需要注意的是，HLS 对于递归定义的函数的支持十分有限，而堆排序在一般的软件算法中，常采用递归定义的方式。因此，在写 HLS 代码时，我们应该避免采用递归定义：如在定义构建大顶堆函数时，我们可以让循环一直持续，直到根元素为最大元素，即为构建完成。同时，在循环当中，我们也利用了循环展开以及流水线的措施，实现了 II=1 的流水线结构。

四、混合排序算法设计

混合排序算法的主要模式分为两种：多路排序 + 传统 k 路归并排序、多路排序 + 败者树归并排序。这里的多路排序，可以是 n 进制基数排序或者堆排序。其基本思想如图3.5所示。对于多路归并排序来说，k 的值可以自由选择。综合考虑硬件资源、数据集大小等因素，我们统一设定 k=64。多路排序算法的并行度可以通过 Vitis HLS 中的 schedule viewer 直观的看出来。以 64 路基数排序 + 多路归并排序为例，图3.6展示的即为其在各个阶段的任务执行情况。可见在第一个时钟周期，64 路基数排序实现了并行执行；接下来的时钟周期均实现了并行

的归并排序，可见其成功实现了算法的并行化。

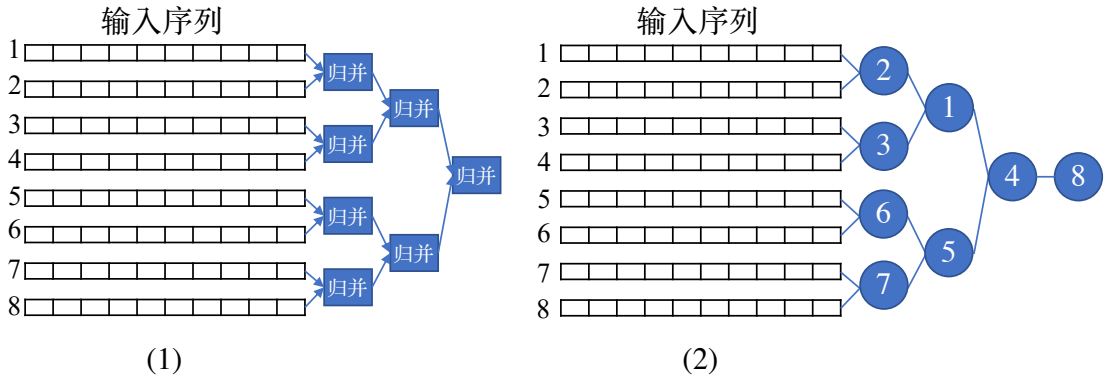

图 3.5　混合算法排序示意图。(1) 多路排序 + 传统 k 路归并排序；(2) 多路排序 + 败者树归并排序

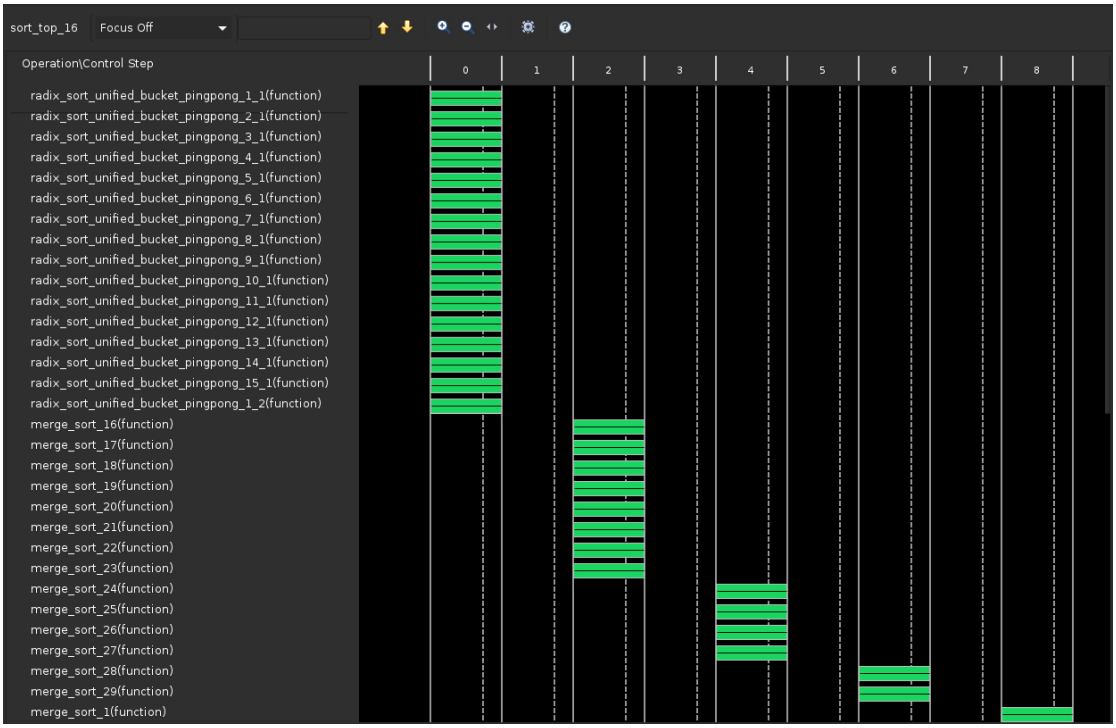

图 3.6　Vitis HLS Schedule Viewer 界面

第四章　硬件性能评估

在这一章里，我们将介绍我们评估硬件性能的方式与结果。包括数据集的构建方式，评估平台的配置，以及性能评估的结果，从而找到最优的硬件架构，以及测试最优的硬件加速比。

第一节　数据集的构建

对于数据集来说，我们设定所有的数为 32 进制数，通过 python 的 random 函数来随机生成 32 进制数，并将其存储到.h 文件中，方便软件和硬件仿真读取。为了比较软件算法和硬件架构在不同大小数据集的性能情况，我们创建了数据量为 100 万，500 万，1000 万数据集，每个给定大小的数据集，我们都采用随机函数生成了三组数据，一共 9 个数据集。在进行软硬件评估时，我们计算软件算法和硬件架构在三组数据集上面的平均性能，并以此作为性能评估的基准。

第二节　软件评估平台以及硬件评估平台配置

一、软件评估平台

我们的软件评估平台如下所示：

1. CPU：Intel Core i7-11800H 2.4GHz
2. 内存：Kingston FURY Beast DDR4 3200MHz 32G
3. 存储：Samsung 980 PCIe 3.0 NVMe SSD 1TB
4. C 语言标准：C99
5. Python 版本: 3.8

二、硬件评估平台

表 4.1　XCU280 FPGA 可利用资源表

LUT	Registers	DSP Slices	BRAMs	UltraRAMs	DDR total capacity	DDR maximum data rate
1.304×10^6	2.607×10^6	9024	2016	960	32GB	2400 MT/s

我们的硬件评估平台如下所示：

1. FPGA：Xilinx Alveo U280 Data Center Accelerator Card
2. EDA 软件：Xilinx Vitis HLS 2022.2

其中 U280 为加速卡，内部集成了 Xilinx XCU280 FPGA，其可利用的资源参数如表4.1所示。表中，LUT 为查找表，Registers 为寄存器，DSP Slices 为数字信号处理模块，BRAM 为片上 RAM，UltraRAM 为一种高性能 RAM，相较于 BRAM 来说有更大的存储密度和更低的静态功耗，且具有更低的访问延迟和更高的操作频率。

图 4.1　Xilinx U280 实物图

第三节　软件与硬件性能评估

基于上面生成的数据集，我们对我们的软件算法和硬件架构进行验证。对于软件算法而言，我们选择 Python 和 C 两种编程语言来实现基数排序、归并排序、堆排序等三种算法，通过 Python 和 C 语言标准库中的计时函数来测试完成排序的软件运行时间；对于硬件架构而言，我们使用 Vitis HLS 生成如表3.1所示的 13 种硬件架构，并将其部署在 U280 加速卡上，测试其完成排序的总耗时，作为衡量硬件性能的标准。

一、100 万级别数据集性能测试

将所有测试结果以柱状图形式绘出，如图4.2所示；其中，详细的软件测试结果如表4.2所示，详细硬件测试结果如表4.3所示。从上面的实验结果可以看出，在排序 100 万个数时，使用 16 进制基数排序 + 传统 64 路归并排序架构可以获得最好的性能，其仅需要 10.72ms 即可完成排序，相较于基于 C 语言最快的排序算法——16 进制基数排序快了 3.92 倍；相较于基于 Python 语言最快的排序算法——堆排序快了 96.08 倍。

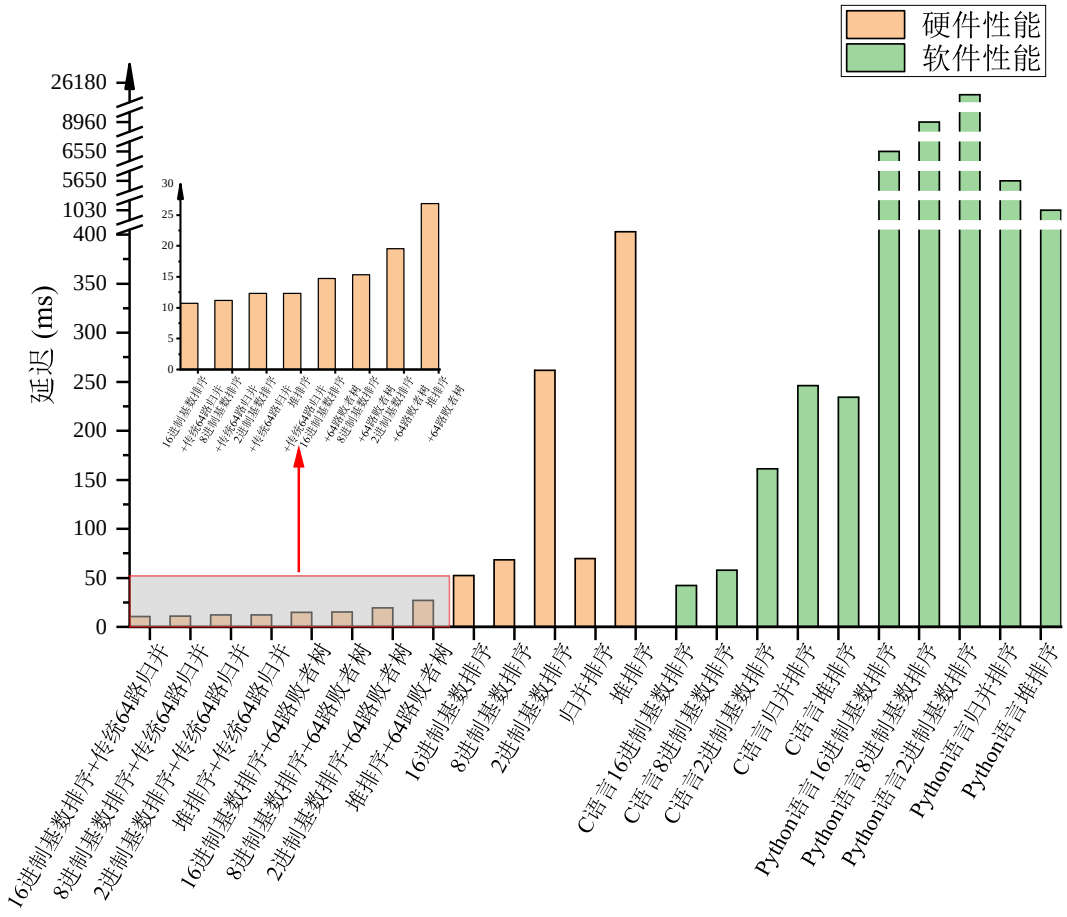

图 4.2　100 万数据集下不同平台性能对比

表 4.2　100 万级别数据集软件性能测试

算法	C 语言耗时 (ms)	Python 语言耗时 (ms)
16 进制基数排序	**42**	6550
8 进制基数排序	58	8960
2 进制基数排序	161	26168
归并排序	246	5650
堆排序	234	**1030**

表 4.3　100 万级别数据集硬件架构性能测试

硬件架构	硬件资源利用情况				性能		
	BRAM	DSP	FF	LUT	时钟周期	频率 (MHz)	总延迟 (ms)
16 进制基数排序 + 传统 64 路归并	3040	128	162639	179055	2125245	198.26	**10.72**
8 进制基数排序 + 传统 64 路归并	3040	128	99023	136431	2172099	194.74	11.15
2 进制基数排序 + 传统 64 路归并	3040	128	47183	117359	2500377	202.51	12.35
堆排序 + 传统 64 路归并	2016	0	55823	114673	3074622	250	12.30
16 进制基数排序 +64 路败者树	2048	128	158447	164386	1156549	78.4	14.75
8 进制基数排序 +64 路败者树	2048	128	94831	121762	1203403	78.4	15.35
2 进制基数排序 +64 路败者树	2048	128	42991	102690	1531681	78.4	19.54
堆排序 +64 路败者树	1024	0	51631	100004	2105931	78.4	26.86
16 进制基数排序	16	4	2424	2420	10000219	190.37	52.53
8 进制基数排序	16	4	1430	1748	13000195	190.37	68.29
2 进制基数排序	16	0	445	1327	65000452	248.63	261.43
归并排序	608	0	5420	14562	20000079	287.6	69.54
堆排序	0	0	803	1365	100740715	250	402.96

二、500 万级别数据集性能测试

类似上一节的做法，我们将所有测试结果以柱状图的形式给出，如图我们首先记录了软件性能测试结果和硬件测试结果，如图4.3所示。其中详细的软件测试结果和详细的硬件测试结果分别如表4.4和表4.5所示。

根据这些实验测得的数据，我们一样能够给出初步结论：在排序 500 万个数的时候，使用 16 进制基数排序 + 传统 64 路归并排序可以获得最好的性能，仅需 53.83ms 就可以完成排序，相较于基于 C 语言的最快算法——16 进制基数排序快了 4.12 倍；相较于基于 Python 语言的最快排序算法——堆排序快了 138.02 倍。

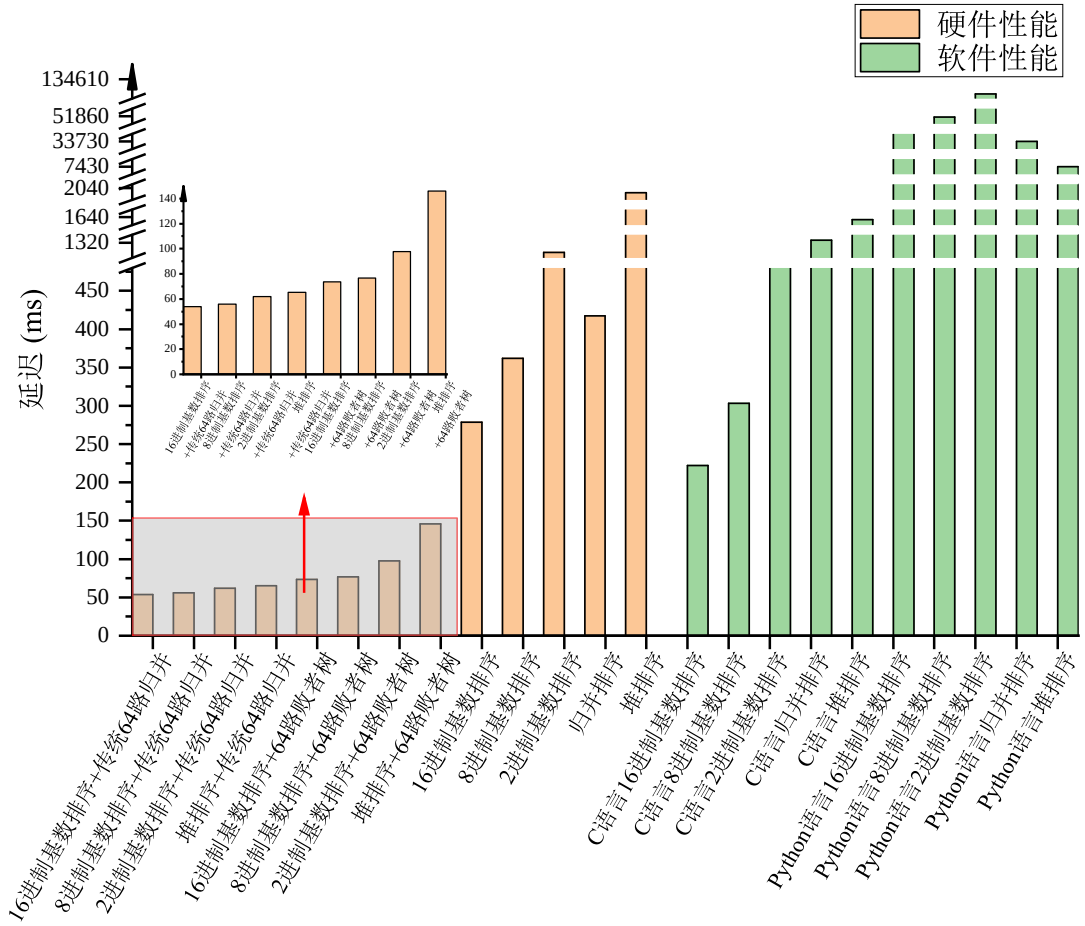

图 4.3　500 万数据集下不同平台性能对比

三、1000 万级别数据集性能测试

根据图4.4、表4.6、表4.7所示的数据，我们可以有如下结论：在对 1000 万数据进行排序的过程中，仍然是使用 16 进制基数排序 + 传统 64 路归并具有最好的性能，仅需 107.78 毫秒即可完成排序，相较于 C 语言最好的算法——16 进制基

表 4.4　500 万级别数据集软件性能测试

算法	C 语言耗时 (ms)	Python 语言耗时 (ms)
16 进制基数排序	**222**	34324
8 进制基数排序	303	51859
2 进制基数排序	795	134591
归并排序	1323	33730
堆排序	1637	**7430**

表 4.5　500 万级别数据集硬件架构性能测试

硬件架构	硬件资源利用情况				性能		
	BRAM	DSP	FF	LUT	时钟周期	频率 (MHz)	总延迟 (ms)
16 进制基数排序 + 传统 64 路归并	3040	128	164553	180848	10625245	197.39	**53.83**
8 进制基数排序 + 传统 64 路归并	3040	128	100937	138032	10859599	193.91	56.00
2 进制基数排序 + 传统 64 路归并	3040	128	49097	119152	12500377	201.61	62.00
堆排序 + 传统 64 路归并	2016	0	59273	116146	16283732	250	65.13
16 进制基数排序 +64 路败者树	2048	128	159986	165927	5781549	78.4	73.74
8 进制基数排序 +64 路败者树	2048	128	96370	123111	6015903	78.4	76.73
2 进制基数排序 +64 路败者树	2048	128	44530	104231	7656681	78.4	97.66
堆排序 +64 路败者树	1024	0	54706	101225	11440041	78.4	145.92
16 进制基数排序	16	4	2448	2501	50000219	179.53	278.51
8 进制基数排序	16	4	1454	1826	65000195	179.53	362.06
2 进制基数排序	16	0	445	1327	325000452	248.63	1307.17
归并排序	736	0	6648	17594	120000095	287.6	417.25
堆排序	0	0	851	1387	508562774	250	2034.25

数排序快了 4.38 倍；相较于 Python 语言最好的算法——堆排序快了 163.48 倍。

表 4.6　1000 万级别数据集软件性能测试

算法	C 语言耗时 (ms)	Python 语言耗时 (ms)
16 进制基数排序	**472**	73390
8 进制基数排序	634	97979
2 进制基数排序	1622	269123
归并排序	2706	75360
堆排序	1637	**17620**

四、总结

基于上述的实验结果，我们可以初步得出以下几个结论：

1. 在常用的 2 进制、8 进制、16 进制基数排序当中，无论是软件算法还是硬件架构的实验结果都显示出 16 进制基数排序速度最快，8 进制次之，2 进制最慢。基于此我们可以推测其原因主要是对于 16 进制基数排序来说，其一次可以排序 4 个比特位，因此对于最外层循环来说（以 32 位整数为例），只需要 8 次循环即可完成排序。而与之对应的 2 进制基数排序则需要 32 次循环才能完成排序，因此 16 进制基数排序相对于 2 进制基数排序要更快；
2. 使用多路混合排序架构的性能会优于单一排序算法的加速架构，其原因也是显然的：使用多路混合排序，能够大大提高排序过程的并行性，从而提升总体的性能；

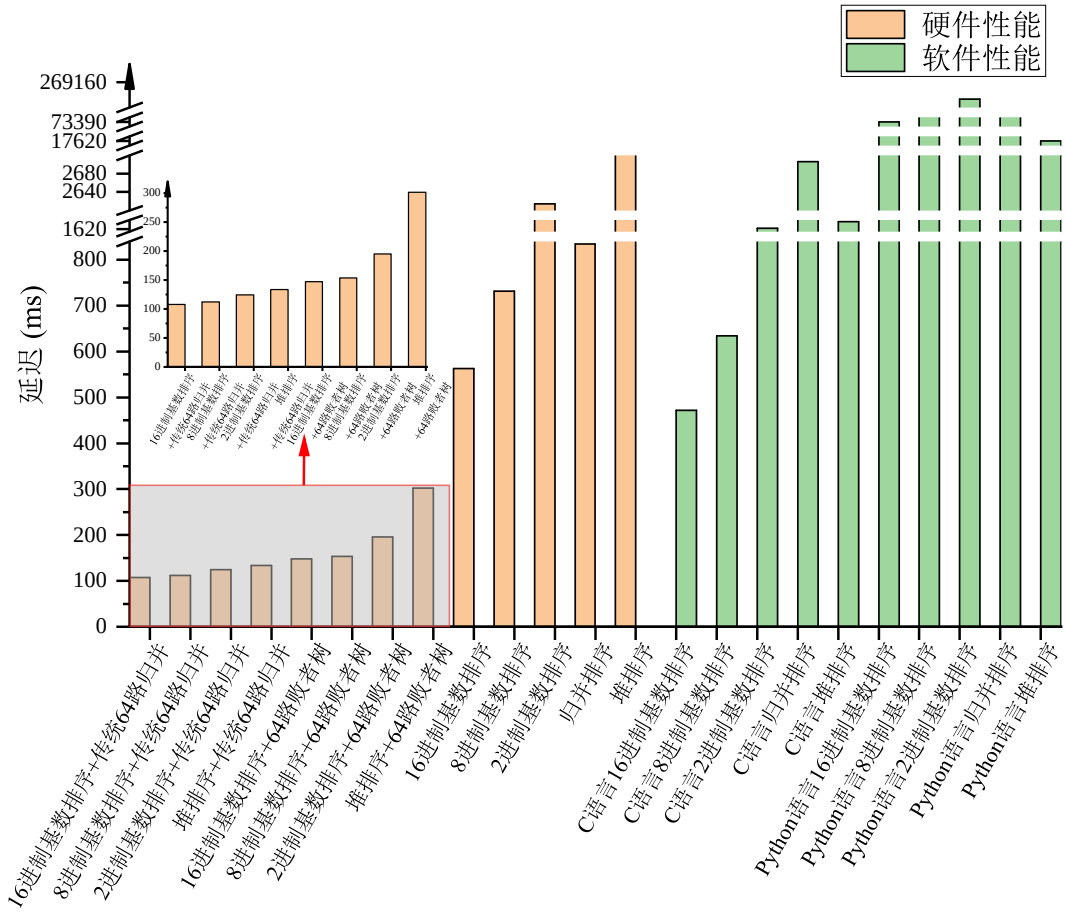

图 4.4　1000 万数据集下不同平台性能对比

表 4.7　1000 万级别数据集硬件架构性能测试

硬件架构	硬件资源利用情况				性能		
	BRAM	DSP	FF	LUT	时钟周期	频率 (MHz)	总延迟 (ms)
16 进制基数排序 + 传统 64 路归并	3040	128	165191	180441	21250245	197.16	**107.78**
8 进制基数排序 + 传统 64 路归并	3040	128	101575	138521	21718974	193.69	112.13
2 进制基数排序 + 传统 64 路归并	3040	128	49735	119705	25000377	201.37	124.15
堆排序 + 传统 64 路归并	2016	0	60423	116635	33346887	250	133.39
16 进制基数排序 +64 路败者树	2048	128	160499	166376	11562799	78.4	147.48
8 进制基数排序 +64 路败者树	2048	128	96883	123496	12031528	78.4	153.46
2 进制基数排序 +64 路败者树	2048	128	45043	104680	15312931	78.4	195.32
堆排序 +64 路败者树	1024	0	55731	101610	23659446	78.4	301.78
16 进制基数排序	16	4	2456	2528	100000219	177.75	562.59
8 进制基数排序	16	4	1462	1852	130000195	177.75	731.37
2 进制基数排序	16	0	450	1330	650000452	248.63	2614.33
归并排序	736	0	6696	17618	240000095	287.6	834.49
堆排序	0	0	867	1392	1018510822	250	4074.04

3. 在多路排序当中，我们使用了 64 路传统归并算法和 64 路败者树两种方法来对前序排序架构生成的 64 路有序数组进行排序。从实验结果我们可以看到相较于 64 路败者树，64 路传统归并算法的优势更为明显。其延迟相对较低，且能够使 FPGA 以更高的频率工作。而从败者树归并的原理我们就可以发现，其过程难以去发掘潜在的并行性，且含有大量的比较和交换的过程，从而造成其整体的工作频率不高（仅有约 78.4MHz），败者树的架构也限制了前序归并排序和堆排序的工作频率，造成整体效率偏低。
4. 相较于 Python，使用 C 语言编写的程序具有更加高的效率。其原因主要有以下几点：首先，Python 是解释型语言，即在运行时需要通过解释器逐行解释并执行。而 C 语言是一种编译型语言，在运行之前，C 代码会被编译器编译成机器码，因此在运行时不需要额外的解释过程。编译型语言通常比解释型语言更快，因为它们避免了运行时解释的开销；其次 Python 是一种动态类型的语言，这意味着变量在运行时才确定其类型。这增加了运行时的开销，因为解释器必须检查每个变量的类型并执行相应的操作。而 C 语言是静态类型语言，在编译时就确定了变量的类型，因此运行时无需进行类型检查，提高了执行效率；另外，Python 的自动内存管理也会对性能造成一定损失；与此同时，C 语言具有高度优化的编译器，可以生成高度优化的机器码，而 Python 的解释器在该方面的能力有限。
5. 在上述三组实验当中，采用 16 进制基数排序 + 传统 64 路归并具有最高的硬件效率。相较于软件算法中最快的 C 语言 16 进制归并排序可以有 4 倍左右的提升；而相较于 Python 语言中最快的堆排序算法，则可以有最多近 200 倍的提升幅度。因此，我们可以认为，该架构是我们的硬件搜索空间当中性能最强的架构。在接下的一章中，我们将采用该架构来解决实际问题。
6. 从三类不同大小的数据集测试数据中我们可以发现，硬件相对于软件的加速比是不断提升的：最优硬件架构“16 进制基数排序 + 传统 64 路归并”相对于 C 语言 16 进制基数排序的加速比从 100 万数据时的 3.92 倍增长到 1000 万数据集时的 4.38 倍；相对于 Python 语言堆排序的加速比从 100 万数据时的 138.02 倍增长到 163.48 倍。这说明我们的硬件加速架构在数据集越大的时候越有优势，符合我们设计的初衷。

第五章　排序算法加速器的应用——大数据集八叉树构建过程加速

在本章中，我们将之前的实验结果运用到实际问题——构建大数据集的八叉树过程当中。以此来举例排序算法加速器的可能应用场景。

第一节　八叉树简介

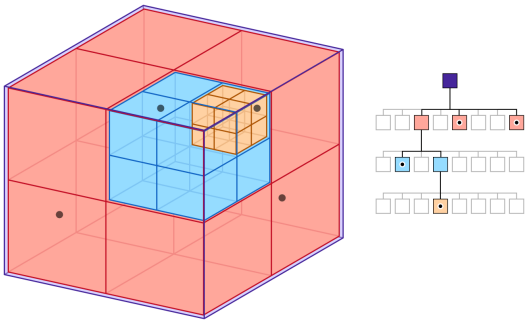

图 5.1　八叉树示意图

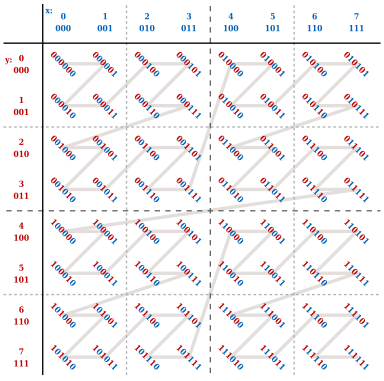

图 5.2　2 维情况下的莫顿码编码顺序

八叉树（Octree）是一种树形的数据结构。其特点在于每个内部结点都有 8 个子节点。而与此同时，如果在三维直角坐标系中对三维空间进行划分，也可以划分成 8 个卦限。因此，八叉树经常被用于分割三维空间，通过增加子节点的层级可以将空间无限递归划分[19]。显然，在构建八叉树的过程中，我们需要确定空间划分的最小尺度（分辨率），从而设定八叉树的最大递归深度。八叉树的优点在于其能够快速进行空间查询。而空间查询是计算机视觉邻域中一个重要的过程：在碰撞检测的粗检测中需要运用到空间查询；在射击游戏中，需要运用空间查询来进行邻近查询；在光线追踪领域当中，也需要利用空间查询来确定光线的入射位置以及反射方向。

在构建八叉树的过程中，我们常用莫顿码[20]来对节点坐标进行编码。使用莫顿码表示节点坐标有诸多好处。首先，它将三维坐标转化为二进制编码，压缩了空间，且方便了结点的索引；其次，莫顿码的 Z 字形遍历编码方式使得在空间中相邻的结点具有相近的值，使得其与八叉树具有直接映射的关系（$3k$ 比特的莫顿码映射到八叉树的第 k 层[21]），这样就相当于将建树问题转换成了排序问题，使得算法更加简洁。

综上，使用莫顿码来对空间的点进行构建八叉树的步骤如下所示：

1. 将点的直角坐标转化为莫顿码；
2. 对莫顿码进行排序；
3. 逐个比较相邻两个莫顿码的高阶位相等位数，寻找当前点与上一个点在八叉树中的共同祖先节点的深度；
4. 循环执行第三步，完成八叉树的构建。

第二节　八叉树构建硬件加速

为了更加全面的测试硬件加速性能在不同大小数据集下的表现，我们仍然生成了包含 100 万、500 万和 1000 万个点的数据集，每组大小的数据集生成三组，共 9 组数据来进行测试。同时，我们在软件程序当中，对每个步骤进行计时，从而测得不同阶段所花费的时间。我们将建树过程分为三个阶段：编码阶段、排序阶段和建树阶段。根据三个阶段的用时情况以及随着数据集增长而增长的速率，我们就可以得出建树过程中的瓶颈位于哪个阶段。

在这里，我们直接选取相对高效的 C 语言作为我们的编程语言，同时在莫顿码的排序过程中，我们也采用最高效的 16 进制基数排序作为排序手段，对三种不同大小的数据集进行测试后，我们得到分阶段的平均用时如表5.1所示。

表 5.1　不同大小数据集下使用 C 语言构建八叉树各阶段平均用时

数据集大小	编码阶段 (ms)	排序阶段 (ms)	建树阶段 (ms)	总用时 (ms)
1000000	**223**	**45**	24	292
5000000	**1144**	**252**	133	1529
10000000	**2243**	**479**	233	2955

根据上面的实验数据，我们不难看出，在这三个阶段当中，编码阶段占用了最多的时间，排序阶段也占用了相当的一部分时间，建树阶段占用的时间最少。因此，我们在设计硬件加速器的过程中，主要关注编码阶段以及排序阶段。对于排序阶段而言，我们也采用之前验证过的最高效的架构——16 进制基数排序 + 传统 64 路归并排序架构来进行排序。

对于编码阶段来说，其算法中包含了大量的移位操作以及逻辑运算，因此十分适合硬件加速。在编写 HLS 代码当中，我们在循环中插入了`#pragma HLS pipeline`操作，实现了 II=1 的流水线架构，从而大大减少了延迟。

我们对我们的硬件架构进行了测试，测试的结果如表5.2所示。为了能够更加

直观地对比，我们将软件测试结果和硬件测试结果以柱状图形式汇出，如图5.3所示。

表 5.2　不同大小数据集下使用硬件加速加速器构建八叉树各阶段用时

数据集大小	工作频率 (MHz)	编码阶段 (ms)	排序阶段 (ms)	建树阶段 (同软件)(ms)	总用时 (ms)
1000000	219.3	**4.56**	**9.69**	24	38.25
5000000	218.25	**22.91**	**48.68**	133	204.59
10000000	217.96	**45.88**	**97.5**	233	376.38

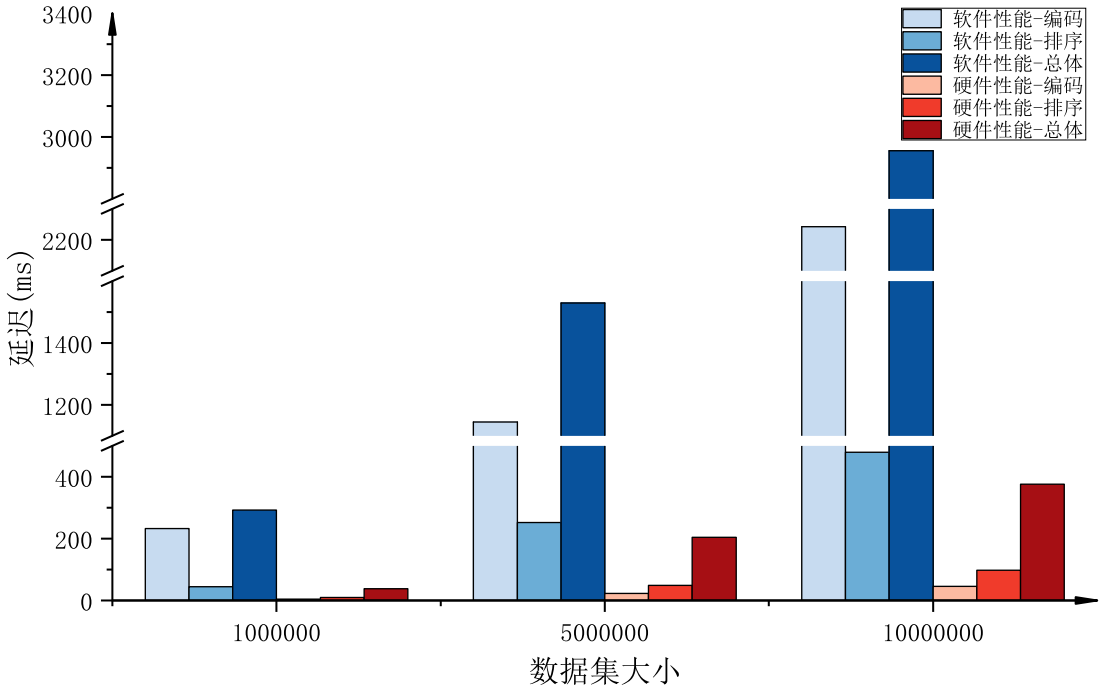

图 5.3　在不同大小数据集下硬件性能与软件性能的对比

由上可见，采用我们设计的硬件加速器以后，建树效率大大提升，相较于软件算法而言，我们的硬件加速算法能够让全过程耗时减少约 87%，特别是对于软件算法性能的瓶颈——编码和排序阶段，加速效果尤为显著。由此可见，经过搜索空间的选择以及针对性的优化以后，我们给出的硬件加速器是非常高效的，可以用于解决实际问题。

第六章　总结与展望

在这个项目当中，为了解决 CPU 在对大型数据集排序过程中所面临的瓶颈，我们尝试针对该过程设计加速器。面对多种多样的排序算法以及海量的可能的加速器架构，我们使用了架构搜索的方法：首先根据时间复杂度以及硬件友好程度来排除掉部分算法以及部分架构，达到缩小搜索空间的目的，然后针对性的设计算法的加速模块，再根据不同的并行度对这些加速模块进行组合和验证测试，从而在最后搜索出最优的排序加速架构——16 进制基数排序 + 传统 64 路归并排序架构，并将其运用在了加速构建八叉树的过程当中，将建树过程缩短了 87%，大幅提升了效率。

在设计排序模块的时候，我们也采用了很多创新性的手段。首先，在设计基数排序算法加速模块的过程中，我们创新性地采用了“统一桶”架构，大大节省了 BRAM 的使用，提升了空间利用效率，也解决了变上限循环无法优化的问题，将架构高度流水化；与此同时，我们还采用了 Pingpong-buffer 架构，通过两组“桶”直接通信，省去了中间数组的存入和读出的时间，进一步节省了排序时间。其次，在设计归并排序模块的时候，我们通过采用数组分割的手段，提升了架构的并行度。而在设计其他排序算法的加速模块的过程中，我们也运用了 pragma 进行了大量的优化。最后，我们采用 HLS 来进行硬件架构的设计，相较于编写 verilog 来说更加高效，体现了设计自动化的优越性。

然而，本工作也并非完美。首先，我们的搜索空间仍然很小，仅有 13 个架构；搜索的过程也是直接枚举评估，并没有采用当今架构搜索 (NAS) 常用的强化学习搜索、进化算法搜索等方法，实际上并没有利用到 NAS 思想的核心所在；实际上，这些搜索算法，也需要在含有大量架构的搜索空间中才可以体现优势，较小的搜索空间仍然限制了搜索自动化思想的应用；其次，由于 1000 万数据仍然能够在 U280 的 BRAM 当中存储，我们并没有设计与内存通信读取的模块，这限制了将我们的架构运用到更大的数据集上的可能性，因为 BRAM 一定是有存储上限的。最后，我们的架构相对于 CPU 来说，加速比仍然不够多。对于较底层的变成语言（如 C 语言），我们也仅仅能够达到 4 倍左右的加速比。这说明我们的架构仍然具有优化的空间。

基于目前的研究进展，我们觉得在未来我们可以有以下研究方向：

1. 增加硬件架构的搜索空间，尝试囊括更多先进的排序算法如 Timsort, Bitonic

sort 等，并尝试利用 NAS 的快速搜索方法，提升搜索的效率与精度；

2. 尝试设计适用更大型数据集（如包含 1 亿数据的数据集）的加速架构，这就要求我们需要考虑与内存的通信，以及可能需要分块排序，重复从内存中加载数据；
3. 尝试将我们的架构与 GPU 进行比较，对于 GPU 来说，其并行度更高，对于我们架构的优化会是更高层次的要求；
4. 尝试运用我们的架构来解决更多实际问题，如碰撞检测，光线追踪，大规模地理数据的处理等方面。

对于大型数据集排序的加速具有很高的实际意义与应用价值，对于这一方面的研究仍然有很多值得令人深究的方向，希望在未来能够基于目前的研究成果，在这一方向做出一定的贡献。

致　谢

从研究课题的确定到最终完成毕业论文，前前后后大约过去了近半年的时间。在这半年当中，有太多值得铭记的时刻：2022 年的 12 月 8 日，科大终于解封了，回归了久违的正常生活；2022 年的 12 月 11 日，提交完了申请季最后一所学校的网申表；2023 年 1 月 24 日，收到了 EPFL 的无面试录取——申请季的第一封录取信；2023 年 2 月 1 日，去上海领事馆面签，顺利通过；2023 年 2 月 10 日，收到了普林斯顿大学的录取，申请季正式结束；2023 年 3 月 25 日，拖着尚未痊愈的身体，独自一人启程飞跃重洋；2023 年 4 月 11 日，第一次在组会中作报告，介绍自己的工作；2023 年 4 月 21 日，第一次参加了学术研讨会；2023 年 4 月 30 日，也就是今天，四月的最后一天，我终于完成了自己的毕业论文。

我一直觉得，自己是极度幸运的：在今年的申请季，作为一个转专业的选手，基础不甚扎实的情况下，仍然被普林斯顿录取，实在是无法用“实力足够”来进行解释；在 UCI 的项目报名截止以后，仍然通过国合部的老师帮忙完成了报名；在 Irvine 的这些日子里，老师同学们的热情善良让在异国他乡的我无比温暖；更不用说前三年在科大的时光，我收到了多少的鼓励与帮助……这一路走来，是多么的幸运啊！因此，我十分珍惜我现在拥有的一切，希望自己能够对得起自己的运气，不辜负如此难得的机会。

在自己的科研学习过程中，我收到了很多老师和同学的帮助。我首先想感谢的是金西老师，正是他让我明确了自己的人生目标，知道了“未来到底需要做什么”，帮助我从零开始迈入科研的大门；我还想感谢我的暑研老师——佐治亚理工学院的郝聪老师，她善良乐观的性格深深地影响着我，她还身体力行的告诉我要成为一个“学术社牛”，多去结交朋友，多去建立人脉，共同在学术上产生新的突破；另外我想感谢的就是加州大学尔湾分校的黄思陶老师，在我做毕设期间给予了全力的支持和帮助：他将最好的加速卡给我使用，每周还会跟我聊天，探讨毕设中所遇到的问题，真的让我非常感动。另外，SoC 设计室的学长袁伟、贵雨宸、佐治亚理工学院博士生陈涵秋、Rishov Sakar、杜克大学博士生李苇航、加州大学尔湾分校博士生徐浩成、徐嘉涔等都在我申请或是科研过程当中给予了相当重要的帮助，在此一并感谢。

最后还是想感谢我的家人们，感谢你们对我无私的爱和无限的包容，我永远爱你们。

2023 年 5 月

参 考 文 献

[1] DEMUTH H B. A report on electronic data sorting[M]. Stanford Research Institute, 1956.

[2] MCILROY P. Optimistic sorting and information theoretic complexity[C]// SODA. 1993: 467-474.

[3] BENDER M A, FARACH-COLTON M, MOSTEIRO M A. Insertion sort is o (n log n)[J]. Theory of Computing systems, 2006, 39: 391-397.

[4] DE MICHELL G, GUPTA R K. Hardware/software co-design[J]. Proceedings of the IEEE, 1997, 85(3): 349-365.

[5] KUON I, TESSIER R, ROSE J, et al. Fpga architecture: Survey and challenges[J]. Foundations and Trends® in Electronic Design Automation, 2008, 2(2): 135-253.

[6] B. Carrion Schaefer. High-Level Synthesis Made Easy[M]. highX Technlogies, 2023.

[7] Kastner R, Matai J, Neuendorffer S. Parallel Programming for FPGAs[A]. 2018. arXiv: 1805.03648.

[8] CONG J, LAU J, LIU G, et al. Fpga hls today: successes, challenges, and opportunities[J]. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2022, 15(4): 1-42.

[9] ŻUREK D, PIETROŃ M, WIELGOSZ M, et al. The comparison of parallel sorting algorithms implemented on different hardware platforms[J]. Computer Science, 2013, 14(4)): 679-691.

[10] JAN B, MONTRUCCHIO B, RAGUSA C, et al. Fast parallel sorting algorithms on gpus[J]. International Journal of Distributed and Parallel Systems, 2012, 3(6): 107.

[11] GEORGOPOULOS K, CHRYSOS G, MALAKONAKIS P, et al. An evaluation of vivado hls for efficient system design[C]//2016 International Symposium ELMAR. IEEE, 2016: 195-199.

[12] CHEN R, SIRIYAL S, PRASANNA V. Energy and memory efficient mapping of bitonic sorting on fpga[C]//Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2015: 240-249.

[13] MATAI J, RICHMOND D, LEE D, et al. Resolve: Generation of high-performance sorting architectures from high-level synthesis[C]//Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2016: 195-204.

[14] PURNOMO D M J, ARINALDI A, PRIYANTINI D T, et al. Implementation of serial and parallel bubble sort on fpga[J]. Jurnal Ilmu Komputer dan Informasi, 2016, 9(2): 113-120.

[15] ROMANOUS B, REZVANI M, HUANG J, et al. High-performance parallel radix sort on fpga[C]//2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2020: 224-224.

[16] QIAO W, GUO L, FANG Z, et al. Topsort: A high-performance two-phase sorting accelerator optimized on hbm-based fpgas[J]. IEEE Transactions on Emerging Topics in Computing, 2022.

[17] JAYARAMAN S, ZHANG B, PRASANNA V. Hypersort: High-performance parallel sorting on hbm-enabled fpga[C]//2022 International Conference on Field-Programmable Technology (ICFPT). IEEE, 2022: 1-11.

[18] XILINX. Vitis High-Level Synthesis User Guide[Z]. 2022.

[19] MEAGHER D. Octree encoding: A new technique for the representation, manipulation and display of arbitrary three dimensional objects by computer[J]. Rensselaer Polytechnic Institute, Troy, NY, 1980.

[20] MORTON G M. A computer oriented geodetic data base and a new technique in file sequencing[M]. International Business Machines Company New York, 1966.

[21] KARRAS T. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees[C]//Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics. 2012: 33-37.

在读期间发表的学术论文与取得的研究成果

已发表论文

无

待发表论文

1. Gui Y, Wei B, Yuan W, et al. Hardware Acceleration of Sampling Algorithms in Sample and Aggregate Graph Neural Networks[J]. arXiv preprint arXiv:2209.02916, 2022.
2. Wei B, Hao C. ReG-NAS: Graph neural network architecture search using regression proxy task[EB/OL]. 2023. https://openreview.net/forum?id=t7HIN3fUAUu.

研究报告

1. 魏博逸. 探究填充复合介质的电容器内部电场、电势物理量的分布以及等效电容条件. 校电磁学小论文比赛，2020.
2. 魏博逸. 探究肥皂膜光须现象. 新生研讨课，2021.
3. 魏博逸. 对氢原子 2p-1s, 3p-1s 跃迁的爱因斯坦辐射系数已近跃迁概率比的计算. 原子物理研讨课，2021.
4. 关浩森，连泓坤，徐文蔚，魏博逸. 金刚石 NV 色心测磁平台搭建和地磁观测. 大学物理-研究性实验，2021.