南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行计算技术课程文档

Mindspore 中 Argminmax 算子的并行化研究

邵铭恺 2120240747

专业：计算机技术

指导教师：王刚

2025 年 7 月 6 日

摘 要

本报告是南开大学计算机学院《并行计算技术》课程的期末实验报告，主题为 Argminmax 算子的并行化研究。本报告中，首先我会介绍我的研究过程，定义问题内容，介绍需要解决的主要问题，对串行、并行算法的编程思路进行分析与具体设计，并介绍 Mindspore 项目中实际使用到的优化技术。通过对各个算法的性能测试与分析，我们得到并行化算法的性能能够优于 std::sort 库函数，并就之后的研究内容给出一定的展望。

关键词：并行计算 Argminmax 多线程 外部排序 SIMD

目录

一、 绪论

(一) 简介

本文档是南开大学 2025 春《并行计算技术》课程的期末研究报告，作者为邵铭恺，学号为 2120240747。文章将针对作业要求，对期末选题中的 ArgMinMax 算子进行编程实验，对算子使用向量指令、多线程、多进程（多核）以及 CUDA 的方法进行优化，考虑不同种方法对算子性能的影响，并结合算子项目本身选取合适的算法策略进行实现研究。

(二) 问题描述

我期末研究报告的选题是 Mindspore 中 Argminmax 算子的并行化研究。开题报告中，讲到该算子包含两种情况，第一种情况是在 topk 大小为 1 时，使用 top1 算法计算，遍历一次即可；另一个算法则是按照轴和偏移量取出对应待操作张量后，将其进行排序，然后直接取前 k 大的元素。由此可见，TopK>1 时，算法的主要计算时间瓶颈在排序的一步上，由此我们可以从排序算法并行化研究入手。

之前的单机编程中，我们对 TopK>1 情况下中的排序算法进行了研究，实现了双调排序并对其进行了并行化实验和验证，本次实验中将结合算子开发项目中的研究方法，对 TopK=1 和 TopK>1 的其他排序算法进行并行化研究，并与 Mindspore 中的原有实现在 6678 板子上的性能之间相对比，来证明我们算法的优化效果。最后，将多种算法融合后得到一款能够交付的 C 语言版本 ArgMinMax 算子来完成工作。

(三) 相关工作

1. TopK=1 算法

TopK=1 时，算法较为简单，即在一个一维数组中找出这一维的最大值，并将其返回。一个简单而朴素的算法就这么得出了，设置一个初始变量为对应数据类型的最小值，然后遍历一次一维数组，每遍历到一个比它大的元素，就将这个初始变量设置为这个比它大的值，到最后就能够得出执行结果。这种朴素方法的时间复杂度为 $O(n)$。

如果现在我们不满于这个算法的性能，我们可以使用向量指令，使用 SIMD 的方法对这个算法进行处理。Intel 和 ARM 公司提出的 AVX [1] 和 NEON [2] 支持我们通过掩码寄存器来实现极值查找的向量化，可以将算法的性能提升数倍。

仅取 TopK=1 的情况，我们也可以使用多线程来对其进行性能优化，可以将数据按照块划分或者循环划分的方式，将其划分为多段，分给单个线程执行一样的算法，最后让多个线程得出的局部最大值进行比较即可得到一个全局最大值。多种编程语言中，有多种并行化库可以做到这些点，如 c++ 的 std::thread，pthread，Java 和 Python 各自的 thread 类，以及 openmp 自动并行化等。

由上文描述，我们可以知道该算子的本质是一个归约操作，NVIDIA 公司在 NVIDIA 编程指南的一部分中介绍了树形归约 [3] 的方法，如图**??**它内部将每相邻的两个元素两两进行比较，然后利用 GPU 的强大算力将其迅速折叠缩小为一个数，这种方式非常适合具备众多核数的 GPU 来进行并行化。此外，2008 年 Dean 等人提出了 MapReduce 的概念 [4]，可以在 Map 阶段将其分为多块后，再在 Reduce 阶段进行结果的聚合和归约操作。现在多个大型 AI 计算框架中都具备各自进行优化后的的 ArgMinMax 算子，我们可以在实验中选取一种最适合的方法（或者是聚合方法）来完成我们的算法研究。

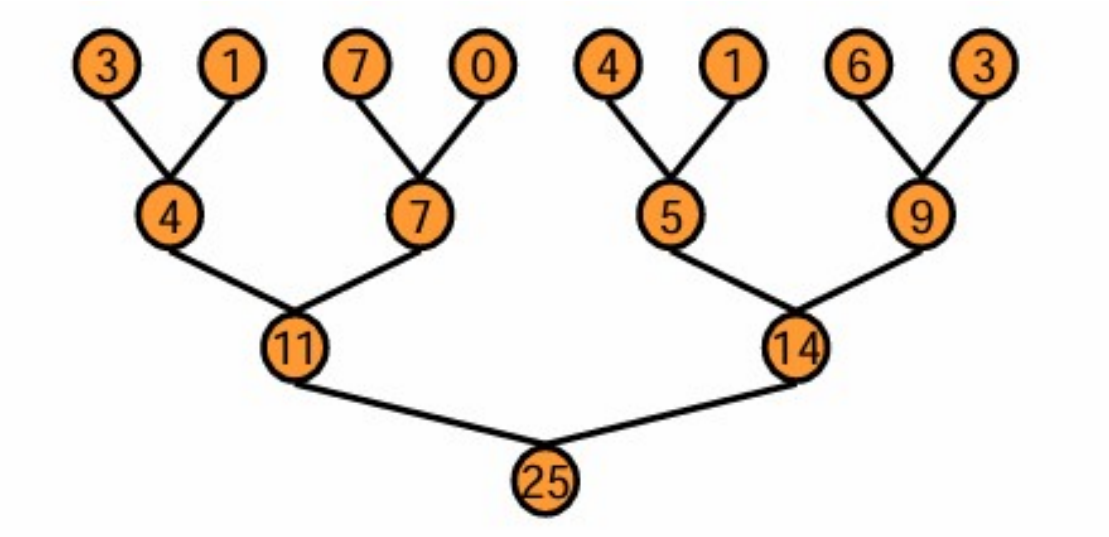

图 1: 树形归约示意图

2. TopK>1 算法

TopK>1 中算法的主要计算时间瓶颈在排序的一步上，由此我们可以从排序算法并行化研究入手。MindSpore 中为我们提供的实现中，内部调用了 C 语言标准库的 qsort 函数，使用串行方式的快速排序来完成算法，我们在这里可以考虑将其修改为并行版本，如并行快速排序，或者更换为更适合并行化的排序算法。排序算法种类繁多，并且有着不同的优缺点，许多研究者都对排序算法的并行化有一些研究，在算子开发项目中我们可以根据实际并行情况选取最合适的算法来进行应用。

关于 SIMD，对于快速排序而言，SIMD 可能难以适用，这里考虑其他的排序算法。我们可以考虑双调排序 [5] [6] 的方法来完成工作，此外也可以考虑归并排序和基数排序的优化。在数据量较大时，双调排序也可以充分利用 GPU 的多个计算单元，使用 CUDA 进行优化。

关于多线程的方法，可以选用快速排序和归并排序的方法，通过共享内存进行线程间通信来进行并行化（多核同理）。需要注意，实际设计排序算法时，可能会出现输入数据过大导致核内存储空间无法存储全部输入数据的情况，此时为充分利用缓存性能，或许需要采用与 MySQL 类似的外部排序算法来进行设计和实现。

二、 实验设计

本次实验主题为 Argminmax 算子的并行化研究，实验内容为使用多种并行方法对 Argminmax 算子进行并行编程实验，并选取最合适的并行化策略或融合方法得出最佳实现。实验中，首先对 Argminmax 算子涉及到的两种情况对应的算法进行实现，包括以下两种情况：

1. Topk=1：求数组中的最大值/最小值；
2. Topk=n：求数组中前 n 大/前 n 小的一些值。

实验中完成这两种情况的对应实现后，将按照这两种情况的思路对并行化算法进行设计，使用多种策略进行算法并行化，包含 SIMD、多线程、多核、DMA 应用等等，对两种串行算法分别进行并行算法的设计与实现，并进行性能的测量、对比、分析。此外，我们还将结合 Mindspore 算子开发项目，对 Mindspore 中 Argminmax 在 6678 和 7004 两种芯片上的一些针对性的性能

优化策略进行介绍，并进行单独的性能对比和分析（需要单独的环境，且只能使用纯 C 语言/线性汇编）。在充分测试之后，我们会使用最接近实际需求的并行化方法，将其应用到项目中，使该实验具备实际应用价值。

三、 Topk=1 算法实现

本节中对 Topk=1 算法的具体实现方法进行描述。

(一) 串行算法实现

Topk=1 情况下，串行算法实现的思路较为简单，我们设置一个局部变量来存储当前的最值，然后遍历数组，在循环中将其与当前最值进行比较。在取最大值的情况下，我们将该局部变量值设置为对应数据类型的最小值，当循环遍历到的值比该值大时，重置当前最大值并记录索引。按此流程，遍历到数组结束时就能够得到结果。

```
1  void topk1int32simple(const vector<int>& input, int& output, int&
       output_value, bool get_max) {
2    int n = input.size();
3    int index = 0;
4    int value = get_max ? INT_MIN : INT_MAX;
5    for (int i = 0; i < n; ++i) {
6      int value_tmp = input[i];
7      if ((value_tmp > value) == get_max) {
8        value = value_tmp;
9        index = i;
10     }
11   }
12   output = index;
13   output_value = value;
14   return;
15 }
```


(二) 并行算法实现

1. SIMD

与双调排序实验类似，我们在进行元素的比较操作时可以使用 SIMD 进行优化，使用掩码寄存器一次比较多个数字，这样也可以保证能够选出列表里最大的元素。所以，我们在这里可以借助 AVX256，一次性处理 8 个数据来对性能进行优化：

```
1  __m256i vb = get_max ? _mm256_set1_epi32(INT_MIN) : _mm256_set1_epi32
       (INT_MAX);
2  __m256i vb_index = _mm256_set1_epi32(0);
3  for (i = 0; i + 8 <= n; i += 8) {
4  __m256i va = _mm256_loadu_si256((__m256i const*)(&input[i]));
5  __m256i idxs = _mm256_set_epi32(i+7, i+6, i+5, i+4, i+3, i+2, i+1, i)
       ;
```

```
__m256i cmp = get_max ? _mm256_cmpgt_epi32(va, vb) :
    _mm256_cmpgt_epi32(vb, va);
vb = _mm256_blendv_epi8(vb, va, cmp);
vb_index = _mm256_blendv_epi8(vb_index, idxs, cmp);
}
```

这里实现与双调排序实验相类似的比较方式，将待比较值加载到向量寄存器，然后根据 get_max 判断方向，之后比较完再进行写回。这里需要把 index 也同时进行写回，来获取更多的返回值信息。在对齐版本中，将对应的 load 和 store 指令改为对齐版本（_mm256_load_si256 和 _mm256_store_si256）即可。

在比较完成之后，我们需要进行归约操作，首先分配两个内存对齐的缓冲数组，将向量寄存器中内容拷贝，然后进行最大值选取操作来完成归约。

```
alignas(32) int buf[8], buf_index[8];
_mm256_store_si256((__m256i*)buf, vb);
_mm256_store_si256((__m256i*)buf_index, vb_index);
for (int i = 0; i < 8; i++) {
  if ((buf[i] > value) == get_max) {
    value = buf[i];
    index = buf_index[i];
  }
}
```

此外，若待遍历数组的大小不是 8 的整数倍，我们需要将剩下的几个量退化到串行算法进行处理：

```
for (i; i < n; ++i) {
  int value_tmp = input[i];
  if ((value_tmp > value) == get_max) {
    value = value_tmp;
    index = i;
  }
}
```

这就是使用 SIMD 并行化 Topk=1 算法的主要流程。本次实验中涉及到的 SIMD 操作主要集中于比较归约操作，后续涉及到类似的方法，也是使用了同样的思路。

2. 多线程

在 Topk=1 算法中，我们可以使用多线程来进一步加速性能，我们将待操作数组均匀分成多份，将每一份交给一个线程进行操作，设置一个和线程大小相同的临时缓冲区保存每个线程得到的结果。最后令主线程对这个缓冲区进行归约，就可以得到最终结果。代码实现如下：

```
for (int t = 0; t < thread_num; ++t) {
  int start = t * batch;
  int end = start + batch < n ? start + batch : n;
  if (start >= end) continue;
```

```
5     results.push_back(pool.submit_task([&, start, end]() -> pair<int,
          int> {  // 回值的时候使用submit
6       ...
7     }));
8   }
9   ...
10  for (auto& f : results) {
11    auto [local_index, local_value] = f.get();
12    if ((local_value > value) == get_max) {
13      value = local_value;
14      index = local_index;
15    }
16  }
```

这里我们使用 BS::thread_pool 实现多线程，使用向上取整除法分配任务批次，并设置 end 的边界条件，将任务均匀分配给多个线程，并维护 results 这个缓冲区，最后进行归约操作，整体的实现思路也较为简单。

3. 融合方法

根据以上的研究, 我们可以使用多线程方法与 SIMD 融合, 完成一个综合性能很强的方法。我在这里使用与双调排序实验类似的方法，将 SIMD 块嵌入到 submit_task 中传入的这个 lambda 表达式中，最终与串行算法相比可以达到 14: 1 左右的加速比。

四、 Topk=n 算法实现

本节中对 Topk=n 算法的具体实现方法进行描述。对于 TopN 算法中，由于选择算法不便于并行化，我们则采用排序策略进行实验，那么该算法中主要优化的关键路径就在于排序算法上。我们之前已经实现了双调排序来作为排序算法的并行化研究，在这里再实现一些其他有潜力的排序算法，对其进行并行化，测量其性能，最后选取一种好的并行化算法来进行 TopN 的排序核函数的计算。

(一) 快速排序

快速排序的思想是，每次排序时选取一个基准，将比这个基准小的所有值放到左边，大的值放到右边，然后对左右子数组重复进行操作，递归到最后数组就是有序的了。串行算法的实现比较简单。

1. SIMD

关于快速排序的 SIMD 方法，一般来说比较难以使用，我们在这里使用一种简单的 SIMD 方法来进行实验。算法的主要是步骤是，比较的过程中，通过向量寄存器将基准值设置为待比较变量，然后一次多比较几个变量并通过比较结束之后的掩码寄存器得到比较结果。在比较结束之后，我们将掩码寄存器的内容取出到一个内存对齐的数组缓冲区中，然后去遍历这个数组缓冲区。由掩码寄存器的性质可知，数组中某一位为真，代表该位对应的两个比较元素结果是逻辑真，那

么我们就可以通过这个结果得到传入 swap 函数中的变量，这样就相当于一次进行了多个比较的操作，再单独对交换进行判断。

与之前一样的是，如果说数组大小不是 8 的倍数，那么我们就需要把最后的元素退化到标量来进行处理，通过这样的方法就可以得到快速排序的 SIMD 版本。但是这样的 SIMD 排序只是对判断做了一下向量化，并没有多少的性能优化效果。

2. 多线程

在设计并行化算法时，我们可以简单将中间的递归过程分配给多个线程，设置合理的递归深度和阈值来进行合理的多线程并行化方法。需要注意的是，在这里我们需要设置合理的递归阈值，比如递归深度或者数组当前的排序数组大小，能避免一次性产生过多个线程，导致的算法效率低下。对于快速排序而言，多线程带来的的性能提升比较有限，因为其负载往往不是很均衡，并且最快的时间复杂度可能退化到 $O(n^2)$。但是因为它的运行速度足够快，还是在小规模数据时被广泛使用。

3. 其他

由于 SIMD 方法没有得到很明显的加速，我们就不将其融合到算法内部了，使用多线程的快速排序来进行后续的实验。尽管我们自己编写算法没有获得一些成果，但是近年来对快速排序算法的研究其实有很多，包括 Berenger Bramas 等人发布的利用 AVX512 的混合排序算法研究，他们研究了一种新的分区算法，并使用基于无分支双调的排序方法对小块数据排序 [7]。此外，还有 Mark Blacher 等人提出的 vqsort 算法，他们的研究突破了单线程的、针对特定指令集、仅限于一小部分键类型这三个 SIMD 快速排序的重要限制，通过将该算法集成到最先进的并行排序器 ips4o 中，能够达成 1.59 的几何平均加速比 [8] 等。如果在算子项目中有必要使用，我们可以再进一步对排序算法的并行化进行调研，来进一步提升算法性能。

(二) 归并排序

归并排序相比于快速排序而言，尽管它也是采用分治的思想，而使用归并排序的任务划分方式往往相对稳定且负载均衡，是一个更加适合用来并行化的比较排序。归并排序的过程主要分为分裂和合并，这和双调排序有点类似，在运行归并排序时，我们拿到一个数组之后，不断将其分为一半，直到只剩最后两个元素时使用合并有序子序列的方法来进行合并，这样最后合并出来就是一个整体有序的数组。由此可见，我们可以对归并合并的操作使用 SIMD 指令优化，而对递归的分裂操作使用多线程。此外，通过修改归并排序的流程，我们还可以使用归并排序配合 DMA 完成外部排序的任务，后文中将详细叙述。

1. SIMD

归并排序中，我们主要将 SIMD 方法应用在合并的这一步，首先使用标量的合并流程来完成任务，而对剩余的情况使用 SIMD 进行处理。这里不能直接将 SIMD 用作比较，因为这只能比出两个向量寄存器间 8 个元素各个元素的大小关系，不能比出所有元素之间的顺序。

```
1 void topknsort_merge_avx256(std::vector<int>& arr, int left, int mid,
      int right, std::vector<int>& temp) {
2   int i = left, j = mid + 1, k = left;
3   while (i <= mid && j <= right) {
4     if (arr[i] <= arr[j]) temp[k++] = arr[i++];
```

```
5       else temp[k++] = arr[j++];
6     }
7     // 简单的SIMD作用有限，这里用于加速剩余部分的拷贝
8     while (i + 8 <= mid + 1) {
9       __m256i va = _mm256_loadu_si256((__m256i*)&arr[i]);
10      _mm256_storeu_si256((__m256i*)&temp[k], va);
11      i += 8;
12      k += 8;
13    }
14    while (j + 8 <= right + 1) {
15      __m256i vb = _mm256_loadu_si256((__m256i*)&arr[j]);
16      _mm256_storeu_si256((__m256i*)&temp[k], vb);
17      j += 8;
18      k += 8;
19    }
20    // 别漏了
21    while (i <= mid) temp[k++] = arr[i++];
22    while (j <= right) temp[k++] = arr[j++];
23    for (int l = left; l <= right; ++l) arr[l] = temp[l];
24  }
```

2. 多线程

在归并排序的多线程算法设计中，由于归并排序是一种分治排序算法，它在每一次分裂时，将原有的待操作数组分为原来的一半，那么我们就可以采取一种简单的方法进行设计，就是在每一次分裂的时候，将一个任务分给子线程，另一个任务由主线程执行。此外，与快速排序一致，我们仍旧设置小数组大小阈值和递归深度的限制，来切换到串行算法完成工作，这样就可以完成一个并行归并排序算法。

由于归并排序的分裂过程总是一半一半地进行分裂，那么我们多线程算法设计相比于快速排序而言，更容易达到合适的负载均衡，这是优于快速排序的地方。

3. 融合方法

在这里我们可以将上面提到的多线程设计策略与 SIMD 相融合来进行一个融合算法的设计，在分裂时将任务给子线程，并在合并时将使用 SIMD 优化，这样就可以得到一个多种策略的融合算法。此外，近年来也有一些对归并排序算法的进一步研究，我们可以根据需求选择合适的策略。

(三) 双调排序

在之前的单机实验中，我已经对双调排序进行过实现，但是相比于快速排序和归并排序，在个人 PC 上测试得出的性能优势不明显，该算法还是在排序网络上应用比较好。

按照上次实验的结果来看，实现的双调排序并行化算法的最优算法为迭代算法下 SIMD+ 多线程的融合方法，能够达到与快速排序相似的性能。该算法由于其容易并行化的优秀性质，在加大线程数之后能够带来较高的性能提升，这是比上面两种算法好的地方。

五、 Mindspore 算子优化

本节中，我将结合算子项目实际工作，介绍算子项目中对 Argminmax 算子的并行化方法策略。我们回顾 Mindspore 中的 Argminmax 算法流程，其中的输入包含轴参数，这表明可能同时有多个数组被传入该工作中。此外，Mindspore 中到 device 端的输入数组都是一维数组，而算法中又包含对四个维度的单独操作，这表明需要手动计算数组的待计算元素。

Algorithm 1 MindSpore ArgMinMax 算子执行流程

```
if param.topk == 1 then
    if param.get_max then
        ArgMaxTopK1(input, output, output_value, 轴维度参数)
    else
        ArgMinTopK1(input, output, output_value, 轴维度参数)
    end if
    return
end if
switch (param.axis)
case 0:
    ArgMinMaxDim0(input, output, output_value, compare_function)
case 1:
    ArgMinMaxDim1(input, output, output_value, compare_function)
case 2:
    ArgMinMaxDim2(input, output, output_value, compare_function)
case 3:
    ArgMinMaxDim3(input, output, output_value, compare_function)
end switch
return
```


在项目对我们算子实现的要求中，存在着很多参数，包括 keep_dims、axis、in_strides 等多种参数，使得我们需要在一次算子的调用过程中实际调用多次 Argminmax 计算核心。axis 等参数的加入也使得原有的连续内存可能被破坏，在取出需要排列的序列时会多次进行内存的随机访问。此外，算子项目中还要求我们对多种数据类型的排序进行支持，这使得如基数排序的一些排序算法复杂度变高。基于以上种种问题，我们需要对算子项目中 dsp 芯片上做一些单独的优化，这些优化方法中的设计细节将在本节介绍到。

(一) 任务级多核并行

算子开发项目中，在 6678 芯片上我们使用 C 语言的多核方式完成，其中共有 8 个核心供我们使用，各个核心包含自己的 L2 Cache，核间共享内存 SMC 和核外内存 ddr。基于上述叙述，一次算子调用可能有多个任务待执行，因此我们可以执行任务级别的多核并行化，为每个核计算任务偏移量，让他们只对自己的任务进行处理。这样的好处是几个任务之间完全不具备数据依赖，能够达到极好的并行化性能，坏处是粒度过于粗，难以在内部对算法性能进行优化，且任务较少时无法利用到多个核。

```
1 void ArgMinMaxTopK1Int32DMA(const DATA_TYPE *input, void *output,
    DATA_TYPE *output_value, const ArgMinMaxComputeParam *param, int
```

```
    pre_axis_count, int axis_count, int after_axis_count, int
    core_mask) {
  int core_id = GetLogicCoreId(core_mask, DNUM);  // 获取核id
  int core_num = GetCoreNum(core_mask);  // 计算用到的核数
  int get_max = param->get_max_ == true ? 1 : 0;
  bool out_value = param->out_value_;
  DATA_TYPE *outputfp32 = (DATA_TYPE *)output;
  int32_t *outputint = (int32_t *)output;
  int i = 0, j = 0, k = 0;
  for (i = 0; i < pre_axis_count; ++i) {
    int output_offset = i * after_axis_count;
    int input_offset = output_offset * axis_count;
    int tasks = after_axis_count / core_num;
    int remain = after_axis_count % core_num;
    int start = core_id * tasks + (core_id < remain ? core_id :
       remain);
    int end = start + tasks + (core_id < remain ? 1 : 0);

    for (j = start; j < end; ++j) { //实际计算
      DATA_TYPE value = MIN_VALUE;
      int index = 0;
      for (k = 0; k < axis_count; ++k) {
        DATA_TYPE value_tmp = input[input_offset + k *
           after_axis_count + j];
        if ((value_tmp > value) == get_max) {
          value = value_tmp;
          index = k;
        }
      }
      //处理输出...
    }
  }
}
```

在 6678 和 7004 芯片上，都不具备在核内开单独多线程的支持，因此如果要实际上在任务较少时使用更细粒度的并行算法，就可以按照上述思路的设计将核当线程用，利用共享内存进行线程间同步，但是这样的并行化方法实现起来较为复杂，难以在线性汇编阶段进行实现，尤其是 Topk=n 时候的并行排序算法，因此我们在多核策略中仅使用任务级并行方法。

(二) 利用 DMA 提高缓存命中率

前面提到，我们系统依赖的 device 端涉及多级内存的设计，包括核内 L2 Cache、核间 SMC 和核外 ddr，内存空间依次增大，读取速度依次变慢。因此，我们在设计并行算法时，也可以通过考虑多级内存的特殊构造，利用 DMA 实现双缓冲的并发内存读取，将待计算元素依次搬入到 L2 Cache 中，通过形成流水线以提高 Cache 命中率的方式来提高算法性能，可以和计算操作并

发来隐藏延迟。

在实际的算子编程中，对于不同的 axis 参数，数据的访问较为跳跃，难以使用 DMA 进行大块的连续数据搬运的优化方法，只有在对数据最后一维进行聚合计算的时候才能使用 DMA 搬运连续大块内存进行优化。只对于最后一维的情况下，这里的优化思路类似于我负责的 All 算子中的概念，利用 DMA 将数据搬入 L2 Cache 提高 Cache 命中率的同时，让数据搬运和计算并发，从而对算法性能进行优化。

```
void ReduceAll##data_type##DMA(int outer_size, int inner_size, int
    axis_size, DATA_TYPE *src_data, DATA_TYPE *dst_data, int core_num)
     {
  ...参数准备
  for (g = start_group; g < end_group; g += group_step) {
    while (!DMA_TransState(0, core_id)); // 等待上一轮DMA传输完成
    curr_step = (g + group_step <= end_group) ? group_step : (
        end_group - g);
    const DATA_TYPE *outer_src = src_data + g * axis_size *
        inner_size;
    memcpy_dma((Uint32)inputtemp, (Uint32)outer_src, curr_step *
        group_bytes, core_id);  // DMA数据搬运
    // 在计算的同时搬运下一组
    memcpy_dma(下一组);
    for (j = 0; j < curr_step; ++j) {
      DATA_TYPE *in_group = inputtemp + j * axis_size * inner_size;
      DATA_TYPE *out_group = outputtemp + j * inner_size;
      for (k = 0; k < inner_size; ++k) {
        DATA_TYPE tmp = LOGICAL_TRUE;
        for (i = 0; i < axis_size; ++i) {
          tmp = LOGICAL_AND(tmp, in_group[i * inner_size + k]);
        }
        out_group[k] = tmp;
      }
    }
    while (!DMA_TransState(0, core_id)); // 等待DMA传输完成
    DATA_TYPE *outer_dst = dst_data + g * inner_size;
    memcpy_dma((Uint32)outer_dst, (Uint32)outputtemp, curr_step *
        inner_size * sizeof(DATA_TYPE), core_id);
  }
  while (!DMA_TransState(0, core_id));
}
```

如果要在其他维中利用 DMA 搬运，那么需要将实际的数据先逐个拷贝到一个缓存数组中，使访问连续，再使用 DMA 进行搬运，由于 Topk1 算法对于输入数组中的数据每个数据只访问一次，所以在这里没有实现的必要，会因为额外开销而导致没有优化效果。

(三) 外部排序

依赖多级内存的并行算法设计中，不仅可以利用 DMA 进行数据搬运来提高缓存命中率的计算，还可以利用多级内存上的 DMA 拷贝，使用双缓冲机制来实现一个边传边算的流水线，以此来提高算法的综合性能。

这也就涉及到了外部排序的概念。我们面向大小远超缓存大小的巨大数组，设计了一个利用双缓冲 +DMA 实现的外部归并排序方法，该算法共分为以下几个步骤:

1. 将大数组分成不超过缓冲区大小的若干个小块;
2. 利用快速排序方法将这些分块进行排序，然后写回外部内存;
3. 对分块排序之后，采用归并排序的设计思路，对有序的数据块进行合并，且在归并时利用缓冲区做 ping-pong 搬运;
4. 归并时用双缓冲（A、B、输出），归并后再 DMA 写回。

在分裂的过程中，我们将小数组使用快速排序进行排序，使用 DMA 将连续的小数组进行拷贝，提高缓存命中率。运行结束后，我们分成的小块都是有序的，下面就可以进行归并排序的合并操作。

在合并的过程中，我们将传入的 cache 均匀分成三块，设置两块为两边的输入缓冲区，一个为输出缓冲区。我们让这两个输入缓冲区拷贝到需要归并的小块的具体内容，并将归并结果保存到输出缓冲区。当输出缓冲区满时，将输出缓冲区结果使用 DMA 搬运到实际数组中，当输入缓冲区用完时刷新输入缓冲区数据，这样就能够完成实际的归并过程。这部分代码较长，详情请查看 github 代码中对应文件（external_sort_unfold.c)。

(四) 线性汇编优化方法

在 6678 上我们不涉及到 SIMD 向量指令的优化方法，而在线性汇编阶段我们就可以使用 SIMD 方法对其进行实现了。在 Topk=1 时，SIMD 方法的优化效果非常显著，我们可以主要将 SIMD 方法应用到该算法中来改善性能。而在 Topk=n 时，对双调排序而言，SIMD 优化效果较为显著，但是对于归并排序和快速排序就没有很大的效果，最终我们将采取合适的方法来进行线性汇编的整体实验。

此外，我们还可以对循环进行并行化，进行软件级别的循环展开和流水线调度，通过减少控制指令判断次数的方式来进行一定的算法性能优化，并且可以进一步利用多核和 DMA 进行操作。由于项目进度问题，目前还没有对本算子的线性汇编方法进行进一步实现，后续会对线性汇编的优化方法设计进行进一步探究。

六、 实验结果

本节中将对以上算法实现的性能数据进行呈现，并简要分析和对比它们之间的性能差距。本节中我们对每一类算法的性能分开进行分析，先将每一种方法的实验数据进行展示，然后在它们之间进行对比分析，来证明并行化方法的有效性，并分析什么算法在什么区间具备着优势，为之后的研究和实现打下基础。

(一) Topk=1 算法

对于 Topk=1 算法而言，我们让数据规模按照 2 的幂次方增长，然后对每个算法的执行时间进行测量。我们设置简单串行算法为 Baseline，测量 SIMD、多线程、SIMD+ 多线程三个优化算法的性能和基础算法进行对比，对比结果如图2所示。

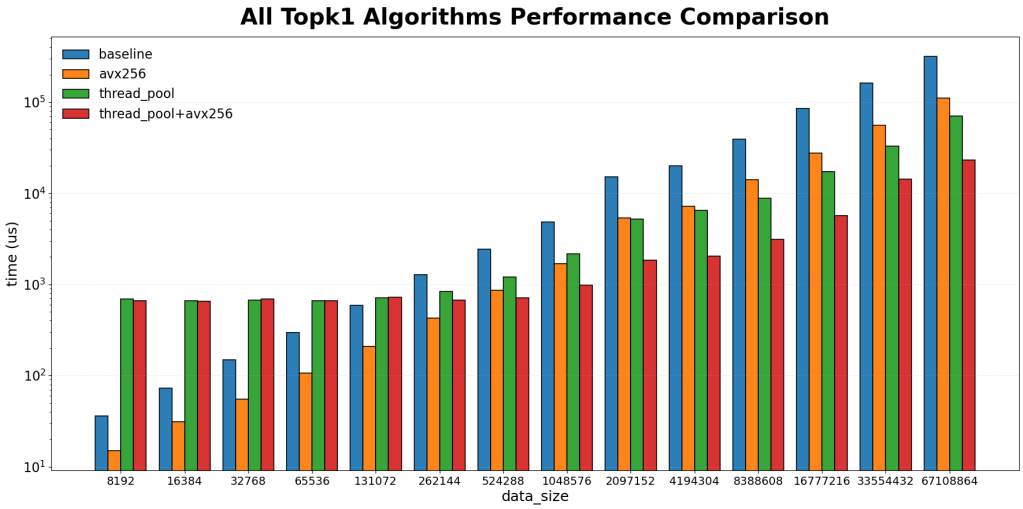

图 2: Topk=1 时，所有算法之间的性能数据对比（无编译优化）

可以看到，数据规模较小时，算法没有明显优势，但在数据规模增长时，多线程、SIMD 和它们的融合方法都取得了一定的性能提升，最大可以达到接近 15：1 的加速比，说明对于 Top1 算法而言，我们设计的并行算法均有一定的成效，但是在数据规模小的时候，使用多线程的开销占据了时间大头，不过在测量范围内的任何时候 SIMD 都有效果。所以我们在小数据上可以使用 SIMD 方法，当性能达到平衡点（131072 262144）之后，将算法切换为融合方法进行使用即可。

此外，在开启编译优化（-O2、-O3）之后，由于编译器对串行算法有着较为激进的优化，造成了性能平衡点的后移，O2 的情况下后移到了数据规模为 2^{20} 左右，而 O3 的情况下后移到了 2^{21} 左右。

方法\ 数据规模	编译选项\执行时间 (us)								
	O0			O2			O3		
	2^{16}	2^{20}	2^{26}	2^{16}	2^{20}	2^{26}	2^{16}	2^{20}	2^{26}
baseline	298	4868	316253	45	668	43326	42	952	41788
avx256	107	1679	110001	35	586	36319	42	796	37002
thread	659	2165	69855	3427	779	14106	709	1200	14712
融合	661	981	23203	1183	770	11970	760	1024	10705

表 1: Topk=1 算法中，不同编译选项对性能带来的影响

开启编译优化之后，在数据量大时算法普遍取得了更佳的性能，尤其是串行算法获得了很大的优化，并行化之后的算法也性能也得到了很大优化，得到了 2.16 的加速比。

(二) Topk=n 算法

在 Topk=n 时，我们使用之前的双调排序的性能测量方法对新实现的算法性能进行测试，将 Baseline 设置为我们实现的简单快速排序算法和 STL 中的 sort 函数，测量三种排序算法的并行化性能，对比结果如图3所示。

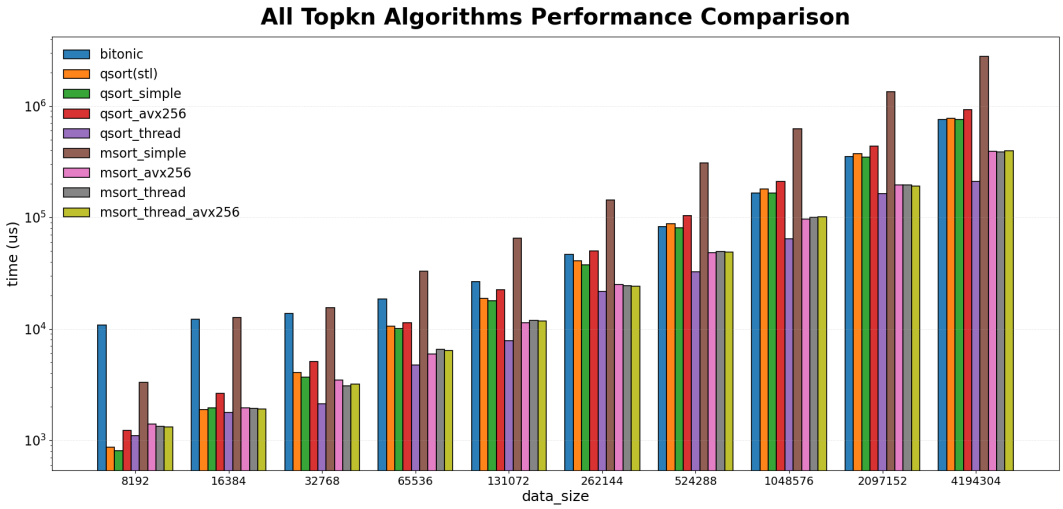

图 3: Topk=n 时，所有算法之间的性能数据对比（无编译优化）

其中，bitonic 为上次双调排序实验中测试得到的最佳性能，qsort(stl) 为 stl 中的 sort 函数，qsort_simple 和 msort_simple 分别是快速排序和归并排序的串行实现，其他是多种并行化方法。

从数据中可以得知，快速排序的多线程版本是这些算法中最优秀的版本，其次是归并排序的三种并行化方法，得到了差不多性能。多种排序算法的并行化版本都相比 stl 中的 sort 函数有着 2：14：1 不等的加速比，说明以上的并行化方法都具备一定的有效性。

输出数据中体现了在数据规模较小时，多线程反而会带来较大负担，不如串行算法好；简单的 SIMD 策略对快速排序几乎没有什么优化，还是需要调研论文来复现他们的方法来达到进一步优化；串行归并排序是这些算法中最慢的，但是也是并行之后加速比最高的（双调排序不算）；线程数达到硬件负担时（电脑为六核心十二线程），再增加线程数量也不会提高并发性；使用混合方法的归并排序和单独进行计算的方法并没有多少性能差距，这或许是因为 SIMD 和多线程对资源占用都比较多导致的。总而言之，对于多线程和 SIMD 的排序，如果要考虑性能，我们还是优先选择快速排序；如果考虑并发性，有更强大的并行硬件可以被使用的话，我们可以考虑使用归并排序或者双调排序来达到更好的加速比。

(三) 算子优化实现

在本节中，我们更换实验环境，去算子开发中实际使用的 Code Composer Studio 中使用 Ti C6678E 模拟器进行编译，并在模拟器环境下测试算法的性能。该部分的算法性能以时钟周期为单位，Baseline 改为 Mindspore 中原有实现的算法性能。

在不同的数据规模下，使用任务级并行的算法性能数据如图4所示，在任何数据规模下，我们都能够得到算法的性能提升，加速比约为 8：1，因为算法利用了 8 个核进行计算。

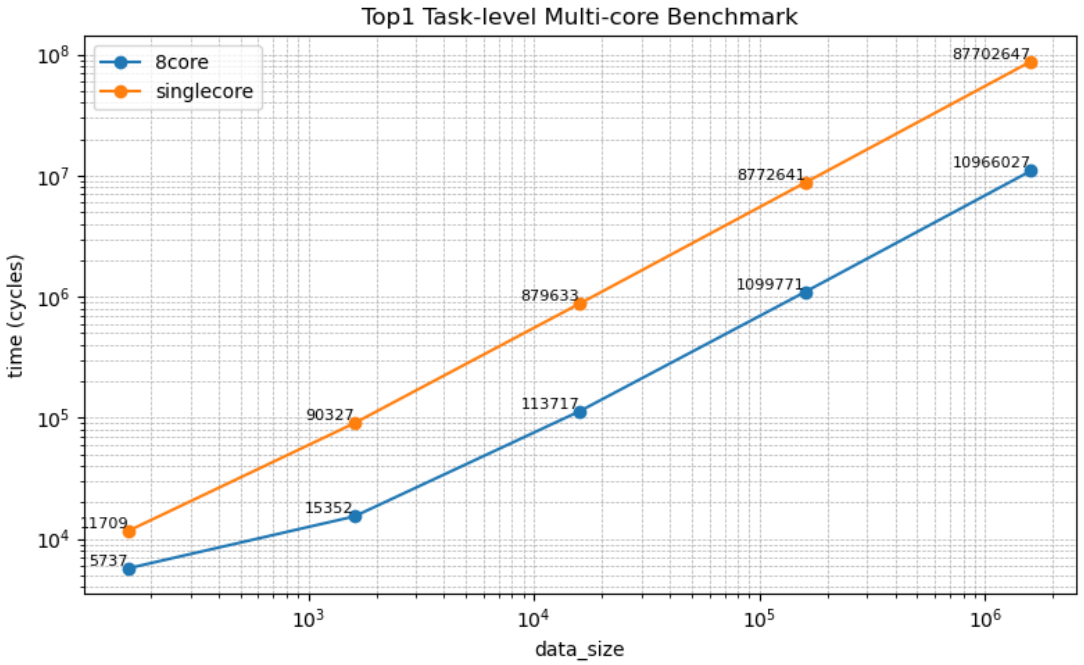

图 4: 任务级并行算法与单核算法性能对比

由于算子开发项目进度问题，对于 DMA 策略、外部排序这两种策略只进行了大概实现，目前还没有具体的性能测试数据，只有初步测试的一些简单结果。将数据使用 DMA 优化之后，得到的性能略有提升，需要进一步调优。关于外部排序的设计，在模拟器上算法性能无明显提升，后续考虑将其应用到开发板上进行开发实验。另外，线性汇编阶段任务刚开始，正在进一步对线性汇编的算法策略进行设计和优化。

七、 总结与展望

本次实验中，我们针对 Argminmax 算子的要求分析了实验的内容，然后针对 Topk=1 和 Topk=n 两种情况分别设计了并行算法，并结合实际的算子项目进行多核、DMA 优化和线性汇编算法设计。然后我们对优化算法进行了实验，对多种并行化策略进行了性能分析。对于我们设计的算法，在数据规模较大时，一些较好的并行算法性能可以优于 C++: std 里面的快速排序算法，并且可以优于上次的双调排序实验结果，具备非常重要的研究和应用价值。后续我们会对算子项目中的重要优化点进行进一步实验和研究，以达到项目要求。

八、 AI 使用说明

与单机并行编程使用情况相同。

参考文献

[1] Intel® Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. Accessed: 2025.

[2] Neon –Arm®. https://www.arm.com/technologies/neon. Accessed: 2025.

[3] Optimizing Parallel Reduction in CUDA. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. Accessed: 2025.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal on Computing*, 18(2):216–228, 1989.

[6] Mihai F Ionescu and Klaus E Schauser. Optimizing parallel bitonic sort. In *Proceedings 11th International Parallel Processing Symposium*, pages 303–309. IEEE, 1997.

[7] Berenger Bramas. A novel hybrid quicksort algorithm vectorized using avx-512 on intel skylake. *arXiv preprint arXiv:1704.08579*, 2017.

[8] Jan Wassenberg, Mark Blacher, Joachim Giesen, and Peter Sanders. Vectorized and performance-portable quicksort. *Software: Practice and Experience*, 52(12):2684–2699, 2022.