同濟大學
TONGJI UNIVERSITY

并行与分布式加速设计

课　　程：算　　法
任课教师：王 晓 年

学　　院：电子与信息工程学院
专　　业：自动化
组　　员：韩　意 2052902
　　孔维涛 2152952

目录

一、 任务描述

两人一组，利用相关 C++需要和加速（SIMD，多线程）手段，以及通讯技术（1.RPC，命名管道，2.HTTP，Socket）等实现函数（浮点数数组求和，求最大值，排序）。所有处理在两台计算机协作执行，尽可能挖掘两个计算机的潜在算力。

尽可能提高加速比，需要列出执行 5 次任务，并求时间的平均值。需要附上两台单机原始算法执行时间，以及利用双机并行执行同样任务的时间。

为了模拟任务，每次访问数据时，用 `log(sqrt(rawFloatData[i]))`拖慢访问速度。例如计算加法，用 `sum+=log(sqrt(rawFloatData[i]))`，而非 `sum+=rawFloatData[i]`。

二、 项目结构与环境说明

程序在两台运行 Ubuntu 22.04 系统的计算机下进行，其中 PC1（client 端）使用 WSL2，6 核+12 个逻辑处理器；PC2（server 端）使用 VMWare，8 核+16 个逻辑处理器。

编译使用 CMake 作为构建工具，GCC 作为编译器，编译模式为 Debug 模式。

两台计算机用 RJ-45 千兆六类网线连接，配置静态 IP 地址，PC2 必须使用桥接模式。

为了保证 CUDA 能够顺利通过编译，需要安装 NVCC 编译器来编译 CUDA 代码。执行以下代码可以统一安装 CUDA 工具包：

```
apt install nvidia-cuda-toolkit
```

项目文件目录树及相应解释如下：

```
.
├── CMakeLists.txt      # 项目构建文件
├── README.md
├── build.bash          # 编译脚本
├── docs
│   └── CHANGELOG.md    # 开发日志
├── include             # 头文件
│   ├── common.h
│   ├── cuda.cuh
│   ├── net.hpp         # 网络公用函数及client端、server端函数
│   ├── original.h
│   └── speedup.h
├── run_client.bash     # client模式运行脚本
├── run_local.bash      # local模式运行脚本
├── run_server.bash     # server模式运行脚本
└── src                 # CPP源文件
    ├── client.cpp      # client端函数
    ├── common.cpp      # 公用函数
    ├── cuda.cu         # CUDA
    ├── main.cpp        # 主函数
    ├── net.cpp         # 网络公用函数
    ├── original.cpp    # 加速前
    ├── server.cpp      # server端函数
    └── speedup.cpp     # 加速后
```

编译以及运行操作都已经编写成 bash 脚本文件。编译代码直接运行 build.bash，通过命令行参数来区分代码运行时的模式，本地加速运行 run_local.bash，client 端运行 run_client.bash，server 端运行 run_server.bash。需要注意，要先启动 server 端，再启动 client 端，以保证正确连接。端口号可以自定义，实践中选用 2345。

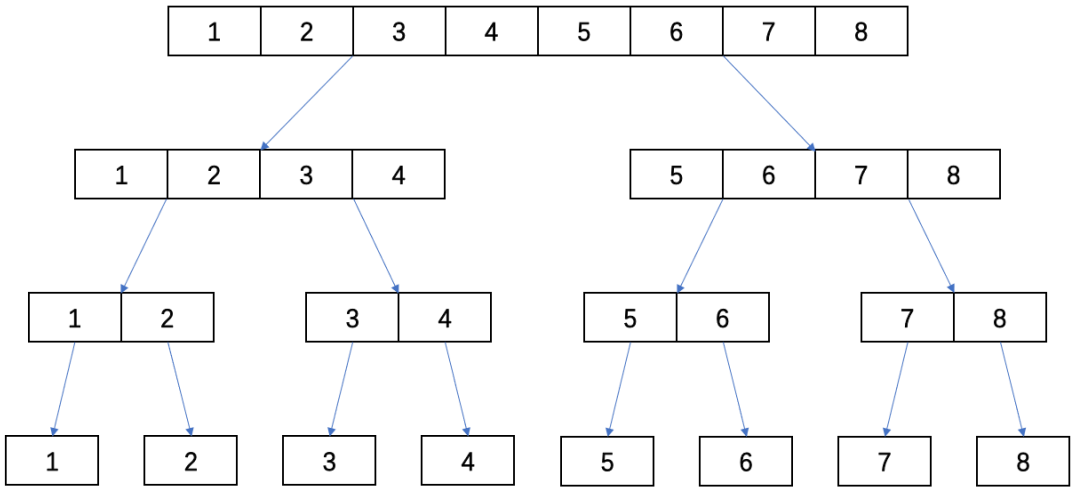
精确到日期的详细开发日志在 CHANGELOG.md 文件中。双人协作开发使用 GitHub 进行，现已设置为 public 状态。仓库地址为：https://github.com/Blattvorhang/SpeedUp-Computing。

三、 加速方法

1. 多线程

由于求和与求最大值是典型的循环结构，适合使用 OpenMP 进行循环并行化，这里只需要在循环体外加一句`#pragma omp parallel` 即可，但要注意对求和结果以及最大值结果进行规约，规约运算符分别使用“`+`”和“`max`”，注意规约运算要求运算本身满足交换律，例如减法就不满足。这里不特别指定开辟的线程数，交由编译器进行处理，以便获得最优的线程数，因为过多的线程反而会带来额外的切换开销。

排序部分使用归并排序，具体而言，将排序过程视为一棵二叉树，这一步与归并排序的分治环节类似，如下图所示（以深度 4 为例，数字仅表示数组块编号，不表示数值大小，蓝色箭头表示拆分）：



每一次二分，递归地开辟两个线程，随后使用 join 函数阻塞等待两棵子树排序完毕后，进行归并。这样，二叉树上的每一个结点，都对应一个线程，总线程数n（包括根节点的主线程）与深度l（根节点深度规定为 0）的关系如下：

$$n = 2^{l+1} - 1$$

反过来有

$$l = \lfloor \log_2(n+1) \rfloor - 1$$

这样，叶节点上的每一块数组就能并行地进行排序（排序算法可以使用普通的归并排序），随后也能并行地对每两块排序结果进行归并。

查看任务管理器，使用多线程后，CPU 使用率达到 100%，说明充分利用了多核 CPU 的算力。

2. 算法

1) Kahan 求和算法

使用 float 类型对数组求和后，发现使用 OpenMP 前后，结果不一致。这是因为浮点数精度有限，加法不满足结合律，可能存在大数吃小数而丢失小数精度的问题。

为了解决累积精度误差问题，未加速的版本直接使用 double 类型来保存求和结果，最后再转为 float 类型返回。加速版本采用 Kahan 求和算法，该算法使用一个额外的变量 `c` 来保存并积累加法运算时引入的低位精度损失，初始化为 `0.0f`。

具体而言，每次访问数组中新的值时，将精度损失加给这一值后得到变量 `y`，再与求和结果 `sum_value` 相加，随后用相加后的结果减去相加前的结果，就可以得到 `y` 的高位，这是因为低位在大数加小数中丢失掉了，而用 `y` 的高位减去 `y`，就可以得到 `y` 的低位的负值，将这个值赋值给变量 `c`，这样就可以在下一次循环中将损失的低位补偿回去。关键代码如下：

```
for (int i = 0; i < len; i++) {
    float y = ACCESSF(data[i]) - c;
    float t = sum_value + y;
    c = (t - sum_value) - y;
    sum_value = t;
}
```

由于使用了 OpenMP，这里要注意同时对两个变量 `sum_value` 和 `c` 进行规约，以避免读写冲突。代码中的 `ACCESSF` 是 `logf(sqrtf(data))`的宏定义，不使用 `log(sqrt(data))`是因为统一用 float 类型计算，减少 float 和 double 相互转换引入的开销。

使用以上算法后，可以发现加速前后加速后的求和结果均为 1.13072e+09，对于 float 类型而言，6 位有效数字均相同就可以认为计算的结果准确了。

此外，成对求和（即使用 OpenMP 并行求和）这一操作本身也能减少浮点数加法大数吃小数带来的累计精度误差问题，这两种操作同时保证了 float 求和结果的精度。

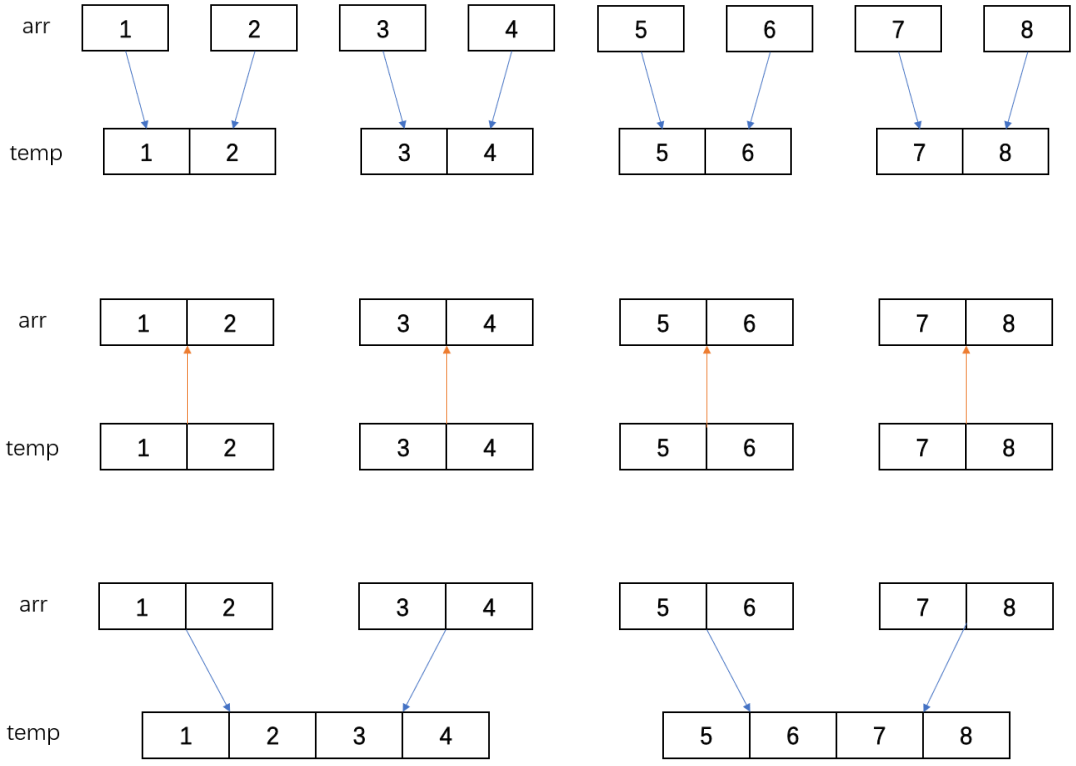
注意，虽然 Kahan 算法可以减少舍入误差，但它也会使代码变得更复杂，并

可能降低代码的性能。但 float 计算比 double 更快，这一牺牲可以换取更高的速度，且最终求和结果与 double 相同。

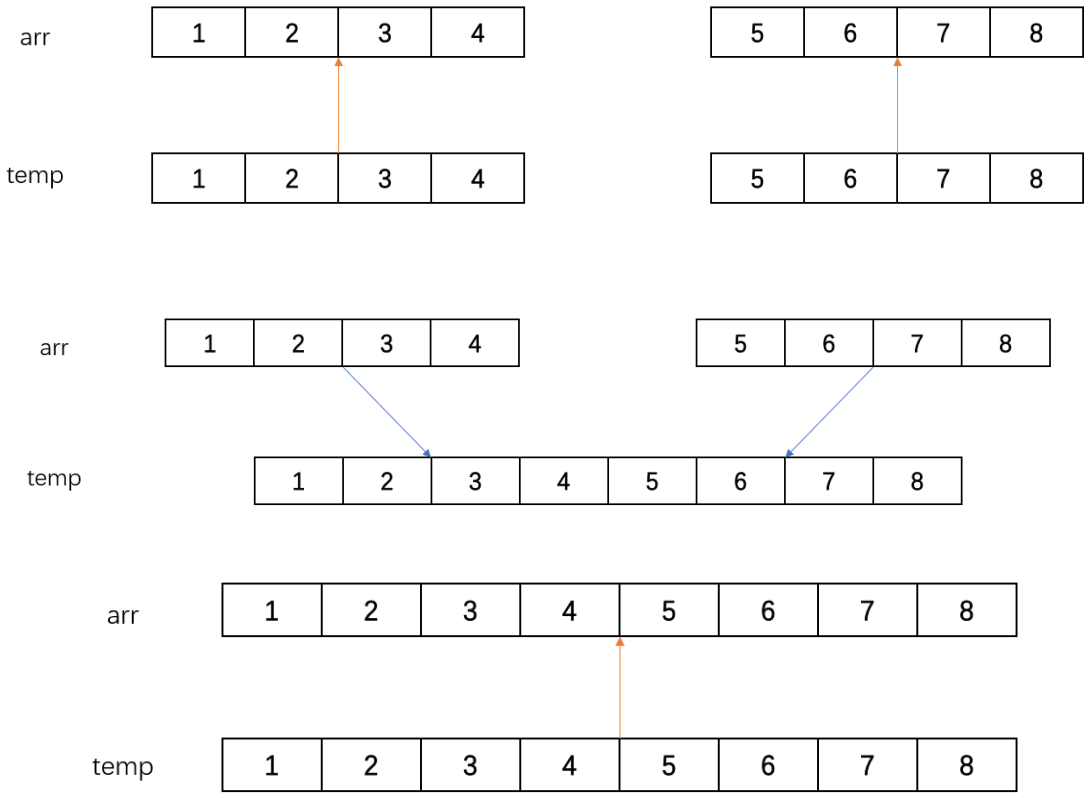
查阅文献[1]可知，Kahan 算法并不能保证绝对的精确。在使用顺序求和时，误差大小为$O(1)$，在使用二分的成对求和（pairwise summation）时，误差会上升到$O(\log_2 n)$，即误差呈对数增长，但在 6 位有效数字内，精度已经足够。

2) 交替归并

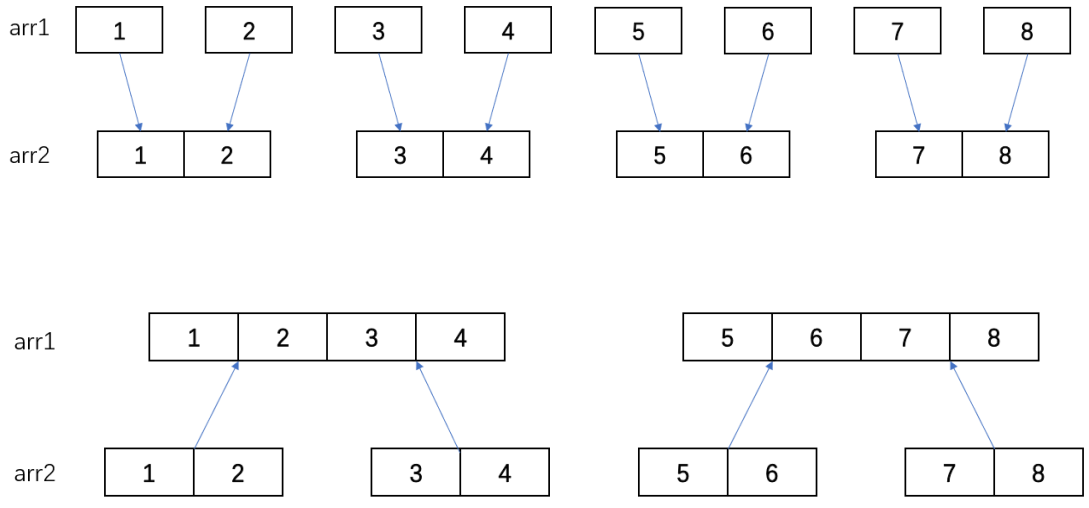
在归并操作中，需要额外的一倍空间作为临时数组 `temp`，来存放归并后的结果，并将 `temp` 中的最终结果写回原数组。显然，如果频繁地进行归并，就会存在频繁的复制操作。同样以深度 3 为例，归并操作如下图所示（蓝色箭头表示归并，橙色箭头表示复制）：

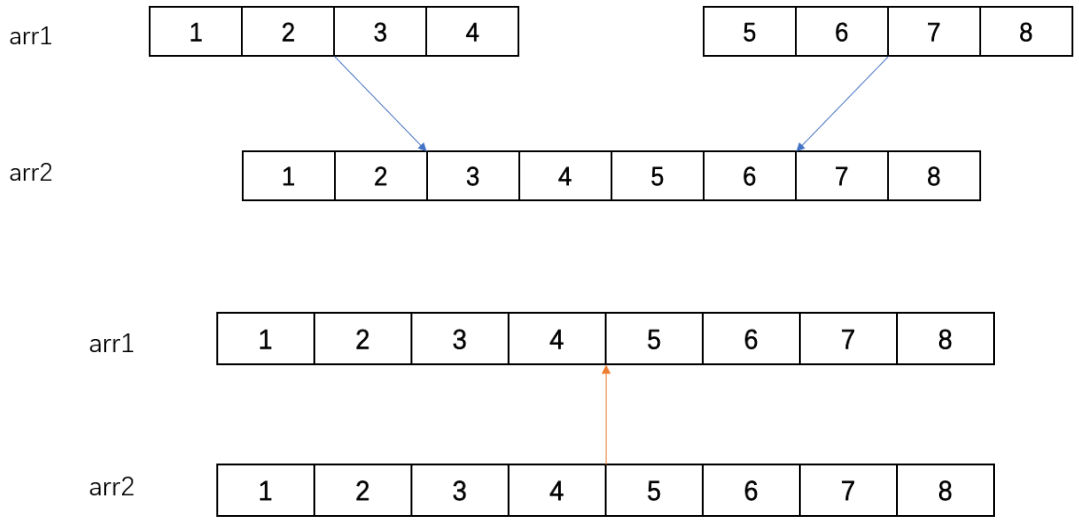


[1] https://en.wikipedia.org/wiki/Kahan_summation_algorithm



上述归并过程一共进行了 6 步，共 3 次完整数组的复制。如果交替地进行归并，即不复制回原数组，而是让两个数组交替存放结果，则可以减少次数。让 `arr1` 作为原数据先分治求得有序的块，然后进行归并，如下图所示：





优化以后，归并操作只有 4 步，少了 2 次复制，提升了计算效率。至于用 arr1 还是 arr2 来存归并结果，以及是否要把最终 arr2 的结果复制回 arr1，可以通过当前深度l的奇偶性来进行判断，当l为偶数时，把 arr2 的结果归并到 arr1；反之，当l为奇数时，把 arr1 的结果归并到 arr2。

关键代码如下：

```
void parallelSortAux(float arr1[], float arr2[], const int len,
const int level) {
    if (level == 0) {
        mergeSort(arr1, len);
        return;
    }

    const int mid = len / 2;
    std::thread t1(parallelSortAux, arr1, arr2, mid, level - 1);
    std::thread t2(parallelSortAux, arr1 + mid, arr2 + mid, len
- mid, level - 1);

    t1.join();
    t2.join();

    if (level % 2)
        merge(arr2, arr1, arr1 + mid, mid, len - mid);
```

```
    else
        merge(arr1, arr2, arr2 + mid, mid, len - mid);
}
```

上述 level 表示当前深度(实际代码中根节点 level 最大,叶子结点 level 为 0，这是为了方便叙述)，可以看到代码中递归地开辟了两个线程，并等待执行完毕后，进行交替归并。

3. 网络

1) 任务量分配

首先说明两台计算机的任务量分配。两台计算机分为client端和server端，其中 client 端负责计算数组中一定比例α的数据，server 端负责剩下$1-\alpha$的数据，并将计算结果回传给 client。α的具体大小通过实验来进行调整，以选取出最佳的比例。

求和部分，client 接收到 server 端数据后，与本地计算结果相加，作为最终结果；求最大值部分，client 接收到 server 端数据后，与本地计算结果比较，选出较大的最为最终结果；排序部分，client 接收到 server 端数据后，需要在本地进行一次归并操作，把两个有序的数组合并为一个有序的数组。

2) 网络配置

项目中采用 RJ-45 千兆六类网线连接两台计算机，使其处于同一以太网下。由于 client 连接 server 端时需要相应 IP 和端口号，这里需要对两台计算机配置静态 IP、子网掩码以及网关。网络使用 192.168.0.0/24，client 端 IP 选择 192.168.0.1，server 端 IP 选择 192.168.0.2，由于不需要用到网关，这里设置为 192.168.0.254 即可。由于 Linux 系统运行在虚拟机下，需要对网络做特殊配置。其中 NAT 模式会进行端口映射，不容易连接到 server 端，应将其改为桥接模式，以便找到虚拟机的具体 IP 与端口号。

首先测试对整个数组的传输，数组大小为 2000000*64，类型为 float，即每个元素占 4 字节，计算如下：

$$\log_2(2000000 \times 64 \times 4) \approx 29$$

从而整个数组约占$2^{29}\mathrm{B} = 512\mathrm{MB}$的大小。使用后文提到的数组传输函数后，整个数组传输用时约为 4.6s，求得带宽为 111.3MB/s = 890.4Mb/s，说明千兆网线带宽基本占满，多余的开销可能是由 TCP 协议带来的。

client 连接到 server 后，针对求和、求最大值、排序三个操作分别创建三个 socket 作为数据通道，独立传输数据，以避免数据交叉造成误传。而基础的 clientSocket 则作为控制通道,用于 client 和 server 的连接建立和握手同步。程序运行完后，要记得将 socket 都关闭。

另外，即使 socket 都已经关闭，且程序结束运行，再重新运行时，同一端口号短时间内也不能再被使用。这是因为操作系统会为已关闭的端口保留一段时间，以确保所有与该端口相关的网络数据包都能完全传递完成，旧的连接数据包不会与新的连接数据包混淆，这被称为 TIME_WAIT 状态。

3) 握手同步

为了保证 client 与 server 同时开始运行加速版本的代码,需要引入一个同步机制，通过两次握手实现。server 阻塞等待 client 发送字符“s”，表示 synchronization，接收到该字符后，server 回传字符“a”给 client，表示 acknowledgement。这样就能保证同步结束后，两台计算机几乎同时开始执行后续的代码。该握手协议对应代码中的 `clientSync()`与 `serverSync()`函数。

4) 异步收发

程序在运行时,无法精确确定是 client 先计算完本地的任务量,还是 server 先计算完。如果是 server 先计算完，但 client 还在计算，就会造成 server 一直等待发送数据给 client 的局面，造成算力以及时间的浪费。

为了避免上述情形，采用异步收发的逻辑，即 client 端额外开辟一个线程用于接收 server 端发来的计算结果，还未接收到，该线程就处于阻塞状态被挂起，不影响本地数据的计算。这样就进一步提高了执行速度。

5) 丢包处理

收发数据使用 send()和 recv()函数，封装了错误处理后，命名为 safeSend()与 safeRecv()，BUFFER_SIZE 选择 1024，这是因为以太网的标准 MTU 通常为 1500 字节，低于这个大小可以减少分片带来的额外开销，降低网络负担。

经过实践发现，socket 在使用 TCP 协议连续发送大量数据时，可能存在收到的数据量不等于发送的数据量的情形，即发生了看似丢包的情况。但是再接收一次数据后，发现剩下的数据在下一个包中，并没有真的丢失，且保持原来的顺序。这就是所谓的粘包问题，会使划分数据变得复杂。

这种现象通常是由于 TCP 协议的工作机制引起的。TCP 是面向流的协议，它提供了一个可靠的、面向连接的数据流，正因如此，TCP 并不关心数据的边界，它只是负责将数据流分割成合适的块，并将这些块传递给上层应用。因此，发送方可能会将多个小的写操作合并成一个大的 TCP 包发送，而接收方则需要根据数据包的大小进行解析。

为了解决上述问题，规定在发送数组前，发送端先发送一个消息长度字段，表示数组的长度 len，但要注意，这里规定的长度是数组长度而非字节数，这一步转换已经封装到了 sendArray()和 recvArray()函数中。随后，发送端开始在 while 循环中每次发送一个大小为 BUFFER_SIZE 的数据块，同时检测自己真正发送出去的量（可以通过 send()函数返回值得知），以此更新发送数据的起始索引，直到整个数组被发送出去为止。接收端同样使用 while 循环而非 for 循环接收数据，直到收到 len 长度的数据后结束循环。这样就解决了粘包的问题。

此外，为了得到更快的传输速度，可以舍弃 TCP，选用 UDP，并使用 QUIC 协议保证数据的可靠、有序以及完整。发生一次丢包时，重发三个相同数据包，如果再丢包，则重发十个。由于该协议相对复杂，本次项目中不采用。

4. CUDA

在我们的 CUDA 和并行编程实现中，我们采用了一种混合方法，将数据集分成两部分，一部分在 GPU 上使用 CUDA 加速处理，另一部分在 CPU 上使用 OpenMP

和 SIMD 进行优化处理。具体来说：

CUDA 部分：利用 sortSpeedUpCuda 函数，将数据集的一部分传输到 GPU 内存并执行归并排序。这一步骤充分发挥了 GPU 的并行计算能力，适合处理大规模数据。

CPU 部分：同时，在 CPU 上，我们根据算法特性选择了基数排序或并行排序。这些算法通过 OpenMP 实现多线程加速和 SIMD 指令集优化，以提高处理速度。

数据合并：最后，GPU 和 CPU 各自处理的结果通过 mergeomp 函数合并，确保得到完全排序的最终数据集。

这种协同工作方式在加快数据处理的同时，还确保了准确性和效率。通过这种策略，我们显著提高了对大规模数据集处理的速度，展示了 CUDA 和并行编程在数据处理方面的强大能力。

在我们的 CUDA 实现中，我们专注于使用 GPU 加速归并排序算法。首先，我们在 GPU 上为输入数据和辅助数组分配内存，并将数据从主机复制到 GPU。我们的核心功能是 mergeSortKernel，它在每个线程中处理数组的不同部分，以并行方式执行排序。

在这个过程中，我们采用了迭代方法，每次迭代中数组处理的部分大小逐渐增加。我们通过交替使用输入和输出数组来有效地合并这些部分，从而实现排序。每次迭代后，我们同步设备以确保所有操作都已完成。

最后，排序后的数据被复制回主机内存，并释放 GPU 上分配的内存。这种方法利用了 GPU 的并行处理能力，显著加快了排序过程，尤其是对于大规模数据集。通过这种方式，我们可以观察到与传统 CPU 方法相比，在处理速度和效率上的显著提升。

让我们深入分析 CUDA 代码中的关键部分：

1. **内存分配与数据传输：**

   使用 cudaMalloc 为输入和输出数据在 GPU 内存中分配空间。

   cudaMemcpy 将数据从主机复制到 GPU。

2. **归并排序内核**（mergeSortKernel）：

   每个线程处理数组的一个特定段，实现并行排序。

   blockIdx.x * blockDim.x + threadIdx.x 计算每个线程的全局

索引。

利用归并排序的逻辑，将数据合并到输出数组。

`if (start >= len) return;` 确保不处理超出数组界限的部分。

3. **迭代排序过程：**

在主机上，通过循环逐渐增加每次处理的数组段大小（由变量 `width` 控制）。

在每次循环中，调用 `mergeSortKernel` 并等待其完成（`cudaDeviceSynchronize`）。

在每次迭代后，交换输入和输出数组的指针，为下一次迭代做准备。

4. **最终数据传回主机：**

使用 `cudaMemcpy` 将排序后的数据从 GPU 内存复制回主机内存。

释放在 GPU 上分配的内存（`cudaFree`）。

这种 CUDA 实现有效利用了 GPU 的并行处理能力来加速数据排序过程，特别适合于处理大规模数据集。通过将数据分块并在多个线程上并行处理，大大提高了排序的效率和速度。

5. SIMD

在我们的代码中，SIMD（单指令多数据流）部分的实现侧重于通过 Intel 的 immintrin.h 库，采用 SSE 指令集来优化浮点数数组的并行处理。这一部分包括以下几个关键步骤：

快速对数计算：使用 `fast_log2` 函数，借助 SSE 指令，将浮点数转换为整数格式，提取其指数部分，并对小数部分应用近似多项式计算，最终得到对数值。

自然对数转换：在 `_mm_log_ps` 函数中，通过 `fast_log2` 函数计算对数后，将其转换为自然对数，方法是乘以 ln(2)的值。

并行和的计算：在 `sumSpeedUp` 函数中，通过 SSE 指令同时处理数组中的四个元素，执行加载、应用对数与平方根等计算，并将这些处理后的元素累加起来，最终将它们的和存储在一个浮点数数组中。

这种 SIMD 优化方法能够在单个 CPU 指令周期内同时处理多个数据元素，极大提高了数据处理的效率和速度，尤其适用于进行大规模数批量计算中，如对数

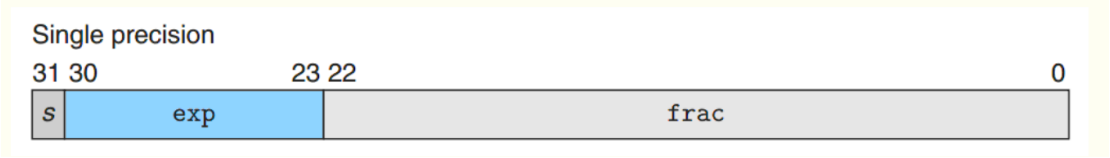
组进行数学函数的应用。通过这种方式，我们的实现能够充分利用现代 CPU 的高级指令集，以实现数据处理任务的高效并行执行，从而显著提高应用程序的整体性能。

但是由于 GCC 编译器本身并不支持 _mm_log_ps 函数，而只支持 _mm_sqrt_ps，我们只能手动实现_mm_log_ps 函数，这一操作拖慢了运行速度，最终计算时间时，并没有采用 SIMD（speedup.cpp 中的 SSE 宏定义置为 0）。

6. 基数排序

基数排序(radix sort)的基本思想是，从最低位开始，按照当前位的数值进行排序得到一个新的序列，对新序列依据高一位的数值重新排序，这样依次循环直到最高位，得到的数列就是一个有序数列

对于 IEEE 754 其存储形式如下图，最高位表示数字的符号，8 位表示指数，23 位表示尾数



IEEE float 有一个特性，除了最高的符号位，从 0 位到 30 位对数值的权重依次增加，这些位与 32 位无符号整数的排序方法相同，针对符号位可做如下的预处理：对于正浮点数，将最高的符号位取反(由 0 转化为 1)。对于负浮点数，全部位取反，这样便可应用整数的基数排序方法(对浮点数应用位运算前需要将其转化为整形), 排序完成后再将其转化。

基数排序的核心原理是按照数字的每个位进行排序，通常从最低有效位开始，逐步到最高有效位。这种方法不依赖于数据的整体比较，而是通过分布式的方式，将数字按位分组排序。

在实现上，我们首先将浮点数转换为整数表示，以便按位

来排序。这一步骤使用了两个关键函数：floatToUInt 和 uintToFloat，分别用于转换浮点数到无符号整数和反向转换。接着，我们对每个字节进行计数排序（countingSortByByte），这是基数排序的核心步骤，它根据每个字节的值进行排序，并且使用一个累积和数组来确定每个值的最终位置。

基数排序的高效之处在于它的分布式处理方式，每次只关注一个字节，而不是整个数字。通过这种方式，我们能够高效地处理大量数据，而不受数字大小的限制。

四、 加速比评估

仅使用 CPU，不使用 CUDA 以及 SIMD 进行双机并行加速，五次实验后，得到的用时以及加速比数据如下：

序号	Client			Server			并行			加速比		
	sum	max	sort	sum	max	sort	sum	max	sort	sum	max	sort
1	1.23	1.43	103.48	1.12	1.16	110.74	0.13	0.14	15.80	9.57	10.22	6.55
2	1.36	1.46	104.52	1.15	1.22	110.75	0.10	0.13	16.07	13.17	11.55	6.50
3	1.45	1.53	104.56	1.07	1.12	111.77	0.12	0.17	16.45	12.05	9.09	6.36
4	1.82	2.12	110.09	1.11	1.12	110.94	0.10	0.12	15.59	17.82	17.07	7.06
5	1.43	1.48	103.34	1.07	1.10	110.51	0.10	0.13	15.78	13.71	11.11	6.55
平均	1.46	1.60	105.20	1.10	1.14	110.94	0.11	0.14	15.94	13.27	11.81	6.60

最终得到的 sum、max、sort 的平均加速比分别为 13.27，11.81，6.60。

使用 CUDA 进行单机加速得到的一次加速比数据如下，单机运行时间在 10s 以内，加速比 12，已经超过双机理论速度（双机最快：10s/2+5s(千兆网传输所有数据)=10s）：

```
--- Original version ---
Sum time consumed: 1.16718
Max time consumed: 1.20018
Sort time consumed: 116.004
Total time consumed: 118.371
sum: 1.13072e+09
max: 9.33377
Result is sorted.
--- Speedup version ---
Sum time consumed: 0.159357
Max time consumed: 0.183435
Sort time consumed: 9.58962
Total time consumed: 9.93241
sum: 1.13072e+09
max: 9.33377
Result is sorted.
--- Speedup ratio ---
Sum speedup ratio: 7.32432
```

Max speedup ratio: 6.54282
Sort speedup ratio: 12.0968
Total speedup ratio: 11.9177

五、 代码解读

初始化操作使用如下代码：

```
for (size_t i = 0; i < DATANUM; i++)
{
    rawFloatData[i] = float(i+1);
}
```

数组本身已经是一个有序的数组，在这种情况下，如果使用冒泡排序，时间复杂度将会下降到$O(n)$，而快速排序则会达到最坏的情况$O(n^2)$，这样的排序是不够客观的。为了保证排序部分的可比较性，需要先对数组进行打乱，这里采用洗牌算法，调用库函数 `std::shuffle`，为了保证两台计算机上打乱的结果一致，使用相同的随机数种子 42 进行打乱。

打乱过程需要的时间较长，为了提高打乱速度，可以分块进行并行打乱，每一个块内的数据打乱，而其中一个块的所有数据可以大于另一个块的所有数据。这里只需要两台计算机打乱方法保持一致即可，对应代码中的`parallelShuffle()`函数。

common.h 文件中，包含了一些公用的定义，如下：

```
#define MAX_THREADS 64
#define SUBDATANUM 2000000
#define DATANUM (SUBDATANUM * MAX_THREADS)   /* total number of
data */

#define ACCESS(data) log(sqrt(data))
#define ACCESSF(data) logf(sqrtf(data))

enum RunningMode {
    LOCAL,
```

```
    CLIENT,
    SERVER
};

enum ProcessingType {
    ORIGINAL,
    SPEEDUP
};
```

这里 ACCESS 和 ACCESSF 是为了拖慢数据访问的速度而定义的宏函数，以降低网络传输开销的占比。RunningMode 和 ProcessingType 均为枚举类型，分别表示运行的模式以及是否加速。

client 和 server 模式的 IP 以及端口号作为命令行参数传入程序，这样可以避免反复编译带来的麻烦。在 main 函数中，对输入的参数具有错误处理，例如使用正则表达式对 IP 的合法性进行判断。

main 函数中，具有 3 个宏定义：

```
#define INIT_SHUFFLE 1
#define TEST_NUM 1
#define SKIP_ORI 0
```

其中 INIT_SHUFFLE 表示初始化时是否打乱数组；TEST_NUM 表示执行同样任务的次数，便于在同一进程下重复执行多次任务，以求用时平均值；SKIP_ORI 表示是否跳过本地执行，这是为了加快对加速时间的测试。

网络连接建立一次，持续保留各个 Socket，直到整个程序运行结束才关闭，因此任务执行的用时不把建立连接包括在内。

net.hpp 中有两个全局常量，如下：

```
const double SEP_ALPHA = 0.5; // client proportion of data
const int BUFFER_SIZE = 1024;
```

这里 SEP_ALPHA 表示 client 端处理的数据占比α，经多次测试得到较优的比例为 0.5。BUFFER_SIZE 选择 1024，原因已在前文中说明。

六、 任务分工

代码整体框架逻辑——韩意

基础未加速部分——韩意

本地多线程、算法加速——韩意

CUDA——孔维涛

SIMD——孔维涛

网络通信——韩意、孔维涛

基数排序——孔维涛

七、 学习体会

本次大作业充分考查了学生对并行计算以及分布式的理解，在完成这次算法期末大作业的过程中，我们深刻体会到了并行计算和分布式加速设计的强大能力。特别是在多线程，CUDA 和 SIMD 部分的应用中，我们亲眼见证了 GPU 和 CPU 在处理大规模数据时的高效能力。

通过这次实践，我们不仅提高了对并行算法的理解，还学会了如何有效地将理论知识应用到实际项目中。通过并行和分布式技术的深入实践，我们对这些领域有了更深刻的理解。多线程，CUDA 和 SIMD 的使用尤其让我们印象深刻，因为它们在处理大规模数据时展示了显著的效率提升。我们学到了如何将理论知识有效地应用在实践中，应用这些先进技术来解决实际问题，我们深刻理解了理论与实践结合的重要性。通过多线程，CUDA 和 SIMD 的应用，我们体验了在处理大规模数据集时显著提高效率的过程。这次经历不仅加深了我们对并行计算和分布式系统的理解，也提高了我们的编程技巧和问题解决能力。