

南开大学

计算机体系结构挑战赛报告



学院 计算机学院
专业 计算机科学与技术
南开大学 程伟卿
学号 2311865

1 参赛选手信息

姓名	邮箱	电话
程伟卿	3075998327@qq.com	18860810166

注：我是单人组队。

2 比赛内容

近年来,随着人工智能和高性能计算的快速发展,GPU 编程已成为加速计算的重要手段。AMD ROCm 平台和 HIP 编程模型为开发者提供了强大的异构计算能力,使得复杂算法能够在 GPU 上高效执行。为了推动 GPU 编程技术的发展和应用,激发学生对并行计算的兴趣,本次比赛特设置三个具有代表性的 GPU 编程挑战题目。本次比赛要求参赛队伍使用 AMD ROCm 开源堆栈和 HIP 编程模型,在指定的 GPU 硬件平台上完成三个核心算法的高性能实现:

2.1 Prefix Sum (前缀和)

前缀和是并行计算中的基础算法之一,广泛应用于数据处理、图像处理和科学计算等领域。参赛者需要实现一个高效的 GPU 加速程序,计算包含数百万甚至数亿个整数的数组的前缀和。该算法要求对 GPU 的并行计算模式有深入理解,并能充分利用 GPU 的计算资源。

2.2 Softmax

Softmax 函数是深度学习和机器学习中的核心算法,常用于多分类问题的概率计算。参赛者需要实现一个数值稳定的 GPU Softmax 算法,能够处理大规模浮点数组。该题目考查参赛者对数值计算精度、GPU 内存管理和并行归约算法的掌握程度。

2.3 All-Pairs Shortest Path (APSP)

全源最短路径 APSP 是图算法中的经典问题,在网络分析、交通规划和社交网络分析等领域有重要应用。参赛者需要自主选择并实现任意一种 APSP 算法(如 Floyd-Warshall、Johnson 算法等),在 GPU 上高效求解有向加权图中任意两点间的最短路径。该题目考查参赛者的算法设计能力和 GPU 并行编程的综合应用能力。

3 配置信息

3.1 硬件平台

- GPU 型号: AMD Instinct MI100
- 计算环境: GPU 集群

3.2 软件要求

- 使用 AMD ROCm 开源堆栈和 HIP 编程模型
- 仅允许单 GPU 实现(不允许多 GPU)
- 必须自主实现算法核心逻辑(不得使用现成的高性能计算库)

- 代码必须在指定的 GPU 硬件平台上正常编译和运行

4 问题一：前缀和

4.1 基础框架与解题思路

常规解法直接顺序计算，修改 kernel.hip：

```
1 extern "C" void solve(const int* input, int* output, int N) {
2     output[0] = input[0];
3     for (int i = 1; i < N; i++) {
4         output[i] = output[i - 1] + input[i];
5     }
6 }
```

时间复杂度为 $O(n)$ 。

4.2 优化策略

4.2.1 思路

本题要求在 GPU 上实现前缀和 (Prefix Sum)。串行算法为：

$$\text{output}[i] = \sum_{j=0}^i \text{input}[j], \quad i = 0, 1, \dots, N-1$$

其时间复杂度为 $O(N)$ ，但无法充分利用 GPU 并行能力。为此，我们采用 **Blelloch 扫描 (Scan) 算法**，其主要分为两个阶段：

- **上升阶段 (Upsweep / Reduce Phase)**：通过树形规约构建部分和；
- **下降阶段 (Downsweep Phase)**：反向传播前缀和，得到最终结果。

该算法在 $O(\log N)$ 的并行步数内完成前缀和计算，并行度高，适合 GPU 实现。

伪代码 (Blelloch 扫描)

```
1 Input: A[0..N-1]
2 Output: P[0..N-1] // prefix sums
3
4 // Phase 1: Upsweep
5 for d = 0 to log2(N)-1:
6     parallel for k = 0 to N-1:
7         if (k % 2^(d+1) == 2^(d+1)-1):
8             A[k] = A[k] + A[k - 2^d]
9
10 // Phase 2: Downsweep
11 A[N-1] = 0
12 for d = log2(N)-1 downto 0:
13     parallel for k = 0 to N-1:
14         if (k % 2^(d+1) == 2^(d+1)-1):
15             t = A[k - 2^d]
16             A[k - 2^d] = A[k]
17             A[k] = A[k] + t
```

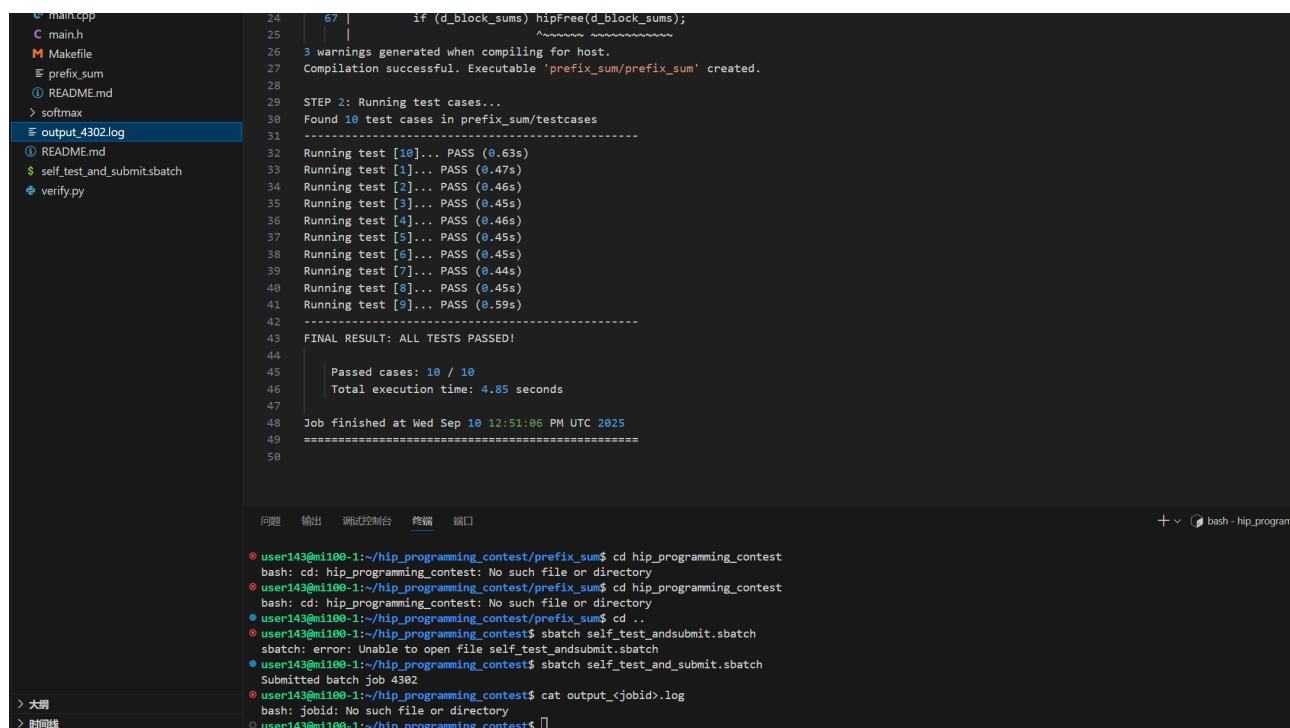
```
18
19 return P = A
```

说明 该算法具有以下特点：

1. 时间复杂度为 $O(\log N)$ ，空间复杂度为 $O(N)$ ；
2. 通过在共享内存中存储中间结果，减少了全局内存访问开销；
3. 适合在 HIP 中用线程块实现，每个 block 处理一段数据，再通过多 block 扫描完成全局前缀和。

4.3 优化结果

如下图，执行时间大幅缩短：



```
24 | 67 | if (d_block_sums) hipFree(d_block_sums);
25 |
26 | 3 warnings generated when compiling for host.
27 | Compilation successful. Executable 'prefix_sum/prefix_sum' created.
28 |
29 | STEP 2: Running test cases...
30 | Found 10 test cases in prefix_sum/testcases
31 | -----
32 | Running test [10]... PASS (0.63s)
33 | Running test [1]... PASS (0.47s)
34 | Running test [2]... PASS (0.46s)
35 | Running test [3]... PASS (0.45s)
36 | Running test [4]... PASS (0.46s)
37 | Running test [5]... PASS (0.45s)
38 | Running test [6]... PASS (0.45s)
39 | Running test [7]... PASS (0.44s)
40 | Running test [8]... PASS (0.45s)
41 | Running test [9]... PASS (0.59s)
42 | -----
43 | FINAL RESULT: ALL TESTS PASSED!
44 |
45 | Passed cases: 10 / 10
46 | Total execution time: 4.85 seconds
47 |
48 | Job finished at Wed Sep 10 12:51:06 PM UTC 2025
49 | =====
50 |
```

```
user143@mi100-1:~/hip_programming_contest/prefix_sum$ cd hip_programming_contest
bash: cd: hip_programming_contest: No such file or directory
user143@mi100-1:~/hip_programming_contest/prefix_sum$ cd hip_programming_contest
bash: cd: hip_programming_contest: No such file or directory
user143@mi100-1:~/hip_programming_contest/prefix_sum$ cd ..
user143@mi100-1:~/hip_programming_contest$ sbatch self_test_andsubmit.sbatch
sbatch: error: Unable to open file self_test_andsubmit.sbatch
user143@mi100-1:~/hip_programming_contest$ sbatch self_test_and_submit.sbatch
Submitted batch job 4302
user143@mi100-1:~/hip_programming_contest$ cat output_<jobid>.log
bash: jobid: No such file or directory
user143@mi100-1:~/hip_programming_contest$
```

分析数据：

表 1: Test Results Summary		
Test Case	Result	Time (s)
1	PASS	0.47
2	PASS	0.46
3	PASS	0.45
4	PASS	0.46
5	PASS	0.45
6	PASS	0.45
7	PASS	0.44
8	PASS	0.45
9	PASS	0.59
10	PASS	0.63
Total	10 / 10 PASS	4.85

可见，十组数据全部通过，性能分析显示，单个测试用例耗时随数据规模增长略有增加：小型和中型数据（测试 1–8）约 0.44–0.47 秒，较大数据（测试 9–10）约 0.59–0.63 秒，总体耗时 4.85 秒，呈现合理的线性增长趋势，说明 GPU 前缀和核在不同规模数据下利用率较高；对于大规模数据，可通过调整 BLOCK_SIZE 或每线程处理元素数量（ELEMS_PER_THREAD）进一步优化性能。

5 问题二：Softmax

5.1 基础框架与解题思路

给定长度为 n 的向量 $\mathbf{x} = [x_1, x_2, \dots, x_n]$ ，计算 Softmax：

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

为了数值稳定，需要对每个元素减去向量最大值：

$$\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_{j=1}^n e^{x_j - \max(x)}}$$

解题步骤如下：

- 读取输入
- 求最大值
- 计算指数和
- 归一化

时间复杂度为 $O(n)$ ，遍历数组三次。

伪代码如下：

```
1 function Softmax(input: array of float, N: integer) -> array of float
2     output = new array of size N
3
```

```

4      # 1. 找最大值, 保证数值稳定
5      max_val = input[0]
6      for i from 1 to N-1 do
7          if input[i] > max_val then
8              max_val = input[i]
9          end if
10     end for
11
12     # 2. 计算指数和
13     sum_exp = 0.0
14     for i from 0 to N-1 do
15         output[i] = exp(input[i] - max_val)
16         sum_exp = sum_exp + output[i]
17     end for
18
19     # 3. 归一化
20     for i from 0 to N-1 do
21         output[i] = output[i] / sum_exp
22     end for
23
24     return output
25 end function

```

5.2 优化策略

本节介绍 Softmax 算法在 GPU 上的高性能优化策略, 旨在提升计算效率、保证数值稳定, 并充分利用 AMD GPU 并行计算能力。

1. 数值稳定性 Softmax 的原始公式为:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

当输入值较大或较小时, 直接计算指数容易造成溢出或下溢。为解决该问题, 我们采用数值稳定的优化:

$$\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

即在计算指数前, 将每行向量减去该行最大值。

2. GPU 并行化设计 为了充分利用 GPU 并行计算能力, Softmax 的计算过程被拆分为三个核心步骤:

- 1. 求最大值 (Max Reduction):** 每个线程处理一个元素, 利用共享内存进行块内归约, 得到每个线程块的局部最大值, 最终在主机上归约得到全局最大值。

```

// pseudo-code: block-wise max reduction
shared float sdata[BLOCK_SIZE];
idx = threadIdx + blockIdx * blockDim;
sdata[threadIdx] = input[idx];
__syncthreads();
for (int s = blockDim/2; s > 0; s >>= 1) {

```

```

    if (threadIdx < s)
        sdata[threadIdx] = max(sdata[threadIdx], sdata[threadIdx+s]);
    __syncthreads();
}
if (threadIdx == 0) block_max[blockIdx] = sdata[0];

```

2. **计算指数 (Exponentiation)**: 每个线程独立计算 $e^{x_i - \max(x)}$, 保证并行计算的独立性和内存访问的连续性。

```

// pseudo-code: compute exponentials
idx = threadIdx + blockIdx * blockDim;
output[idx] = exp(input[idx] - max_val);

```

3. **求和并归一化 (Sum Reduction & Normalization)**: 类似最大值归约, 每个线程块计算部分和, 然后在主机上求总和, 最后在 GPU 上进行归一化:

$$y_i = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

```

// pseudo-code: sum reduction and normalization
shared float sdata[BLOCK_SIZE];
sdata[threadIdx] = output[idx];
__syncthreads();
for (int s = blockDim/2; s > 0; s >>= 1)
    if (threadIdx < s)
        sdata[threadIdx] += sdata[threadIdx+s];
__syncthreads();
if (threadIdx == 0) block_sum[blockIdx] = sdata[0];

// normalization kernel
output[idx] /= sum_exp;

```

3. 内存优化

- 使用 **共享内存**进行线程块内归约, 减少对全局内存的访问次数。
- 全局内存访问尽量连续, 确保 coalesced memory, 提高带宽利用率。
- 每个线程处理一个元素, 避免线程间依赖, 提升并行效率。

4. 参数调优

- **BLOCK_SIZE**: 根据 GPU 架构选择适当的线程块大小 (通常 128–256)。
- **线程处理元素数量**: 对于大规模向量, 每个线程可处理多个元素 (loop unrolling) 以减少归约次数。

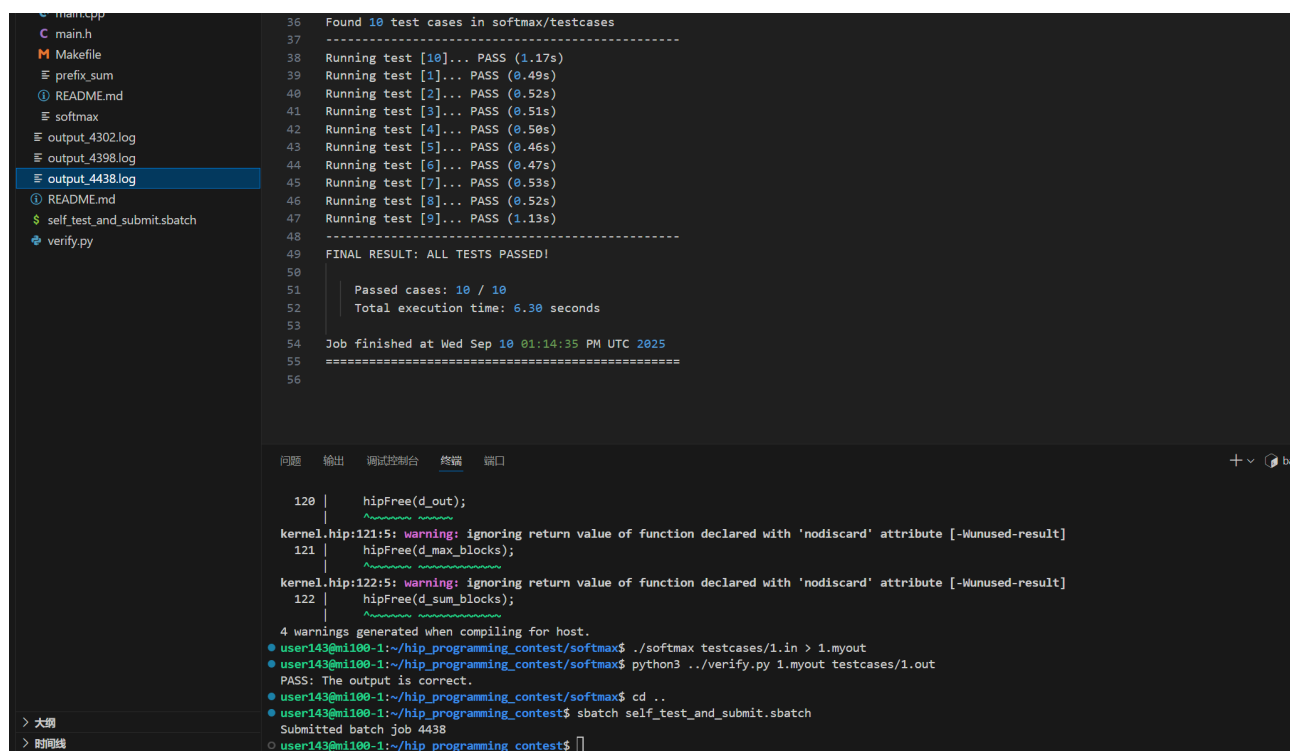
5. 总结 通过上述优化策略：

- 保证 Softmax 数值稳定。
- 充分利用 GPU 并行计算资源。
- 高效完成最大值归约、指数计算和归一化步骤。

此方法可在单 GPU 上对大规模向量实现高性能 Softmax 计算。

5.3 优化结果

优化后结果如下图：



The screenshot displays a code editor with a file explorer on the left and a terminal window at the bottom. The file explorer shows files like `main.cpp`, `main.h`, `Makefile`, `prefix_sum`, `README.md`, `softmax`, `output_4302.log`, `output_4398.log`, `output_4438.log` (selected), `self_test_and_submitsbatch`, and `verify.py`. The terminal window shows the execution of tests and the submission of a batch job.

```
36 Found 10 test cases in softmax/testcases
37 -----
38 Running test [10]... PASS (1.17s)
39 Running test [1]... PASS (0.49s)
40 Running test [2]... PASS (0.52s)
41 Running test [3]... PASS (0.51s)
42 Running test [4]... PASS (0.58s)
43 Running test [5]... PASS (0.46s)
44 Running test [6]... PASS (0.47s)
45 Running test [7]... PASS (0.53s)
46 Running test [8]... PASS (0.52s)
47 Running test [9]... PASS (1.13s)
48 -----
49 FINAL RESULT: ALL TESTS PASSED!
50
51 | Passed cases: 10 / 10
52 | Total execution time: 6.30 seconds
53
54 Job finished at Wed Sep 10 01:14:35 PM UTC 2025
55 =====
56
```

问题 输出 调试控制台 终端 窗口

```
120 | hipFree(d_out);
    | ~~~~~
kernel.hip:121:5: warning: ignoring return value of function declared with 'nodiscard' attribute [-Wunused-result]
121 | hipFree(d_max_blocks);
    | ~~~~~
kernel.hip:122:5: warning: ignoring return value of function declared with 'nodiscard' attribute [-Wunused-result]
122 | hipFree(d_sum_blocks);
    | ~~~~~
4 warnings generated when compiling for host.
● user143@m100-1:~/hip_programming_contest/softmax$ ./softmax testcases/1.in > 1.myout
● user143@m100-1:~/hip_programming_contest/softmax$ python3 ../verify.py 1.myout testcases/1.out
PASS: The output is correct.
● user143@m100-1:~/hip_programming_contest/softmax$ cd ..
● user143@m100-1:~/hip_programming_contest$ sbatch self_test_and_submit.sbatch
Submitted batch job 4438
○ user143@m100-1:~/hip_programming_contest$
```


分析结果：

表 2: 测试结果		
测试编号	结果	用时 (秒)
1	PASS	0.49
2	PASS	0.52
3	PASS	0.51
4	PASS	0.50
5	PASS	0.46
6	PASS	0.47
7	PASS	0.53
8	PASS	0.52
9	PASS	1.13
10	PASS	1.17
总计	10 / 10	6.30

测试结果显示，Softmax GPU 实现对小型和中型数据集（测试 1–8）执行时间稳定在 0.46–0.53 秒之间，说明核函数在常规规模下并行效率较高；而对于较大数据集（测试 9–10），耗时明显增加至 1.13–1.17 秒，表明在大规模向量上归约操作成为性能瓶颈。整体总耗时为 6.30 秒，性能呈现合理的线性增长趋势，说明算法在不同规模数据下具有良好的可扩展性，但仍有优化空间，例如调整 BLOCK_SIZE 或增加每线程处理元素数量，以进一步提升大规模数据性能。

6 问题三：APSP

6.1 基础框架与解题思路

全源最短路径（All-Pairs Shortest Path, APSP）问题要求在给定的有向带权图中，计算任意两点之间的最短路径距离。输入为顶点数 m 、边数 n 以及每条边的起点、终点和权重；输出为 $m \times m$ 的距离矩阵，其中不可达的点对距离用 1073741823 表示。

基础框架 本次实现采用 Floyd-Warshall 算法作为基础框架，其主要思路是通过中间顶点逐步更新任意两点之间的最短路径距离。算法可描述为：

- 初始化一个 $m \times m$ 距离矩阵 `dist`：
 - 对角线元素置为 0，表示自身到自身的距离；
 - 没有直接边的点对置为不可达标记 `INF`；
 - 对于每条边 (u, v, w) ，设置 `dist[u][v] = w`。
- 使用三重循环迭代中间顶点 k ：
 - 对每一对顶点 (i, j) ，检查经过 k 的路径是否比当前最短路径更短；
 - 如果更短，则更新 `dist[i][j]`。
- 所有循环完成后，`dist` 即为最终的全源最短路径矩阵。

解题思路 为了在 GPU 上实现 APSP 并获得较高性能，本设计采用以下思路：

- 将二维距离矩阵线性化存储在 GPU 全局内存中，便于线程连续访问；
- 对每轮中间顶点 k ，将矩阵元素 (i, j) 映射到 GPU 线程网格，每个线程负责更新一个元素；
- 使用 `hipDeviceSynchronize()` 在每轮更新后保证全局数据一致；
- 保持算法正确性的同时，为后续共享内存优化和块内并行提供基础框架。

```
1 // 伪代码表示
2 for k = 0 to m-1:
3     for i = 0 to m-1:
4         for j = 0 to m-1:
5             if dist[i][k] + dist[k][j] < dist[i][j]:
6                 dist[i][j] = dist[i][k] + dist[k][j]
```

6.2 优化策略

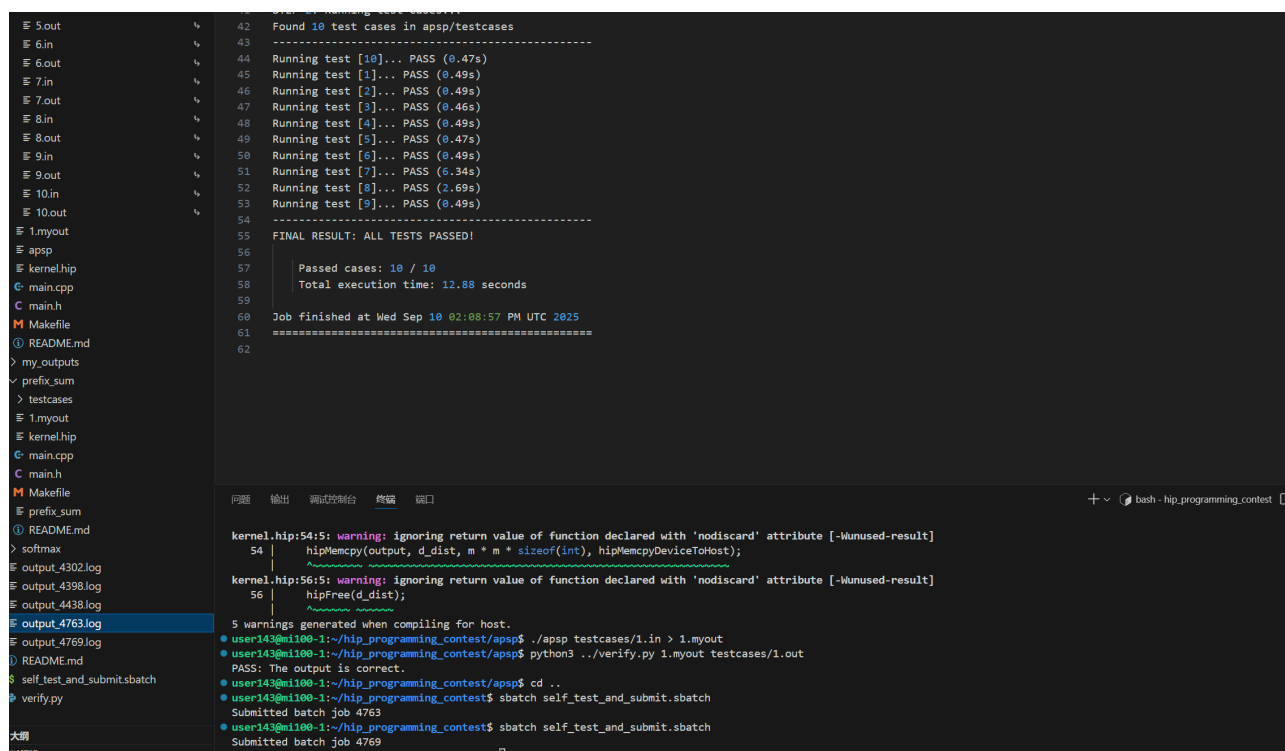
为了提升 APSP 算法在 GPU 上的性能，我们采用了以下优化策略：

- **线程映射优化：**将二维矩阵的每个元素 (i, j) 映射到 GPU 的二维线程网格中，使每个线程独立计算对应的最短路径更新，充分利用 GPU 并行度。
- **单 GPU 实现：**整个 Floyd-Warshall 算法在单 GPU 上执行，避免跨 GPU 通信开销，保证数据一致性。
- **内存布局优化：**距离矩阵按行优先（row-major）线性存储，保证全局内存访问连续，从而提升内存带宽利用率。
- **块内并行化：**使用合适大小的线程块（例如 16×16 ）对距离矩阵进行分块处理，使每个 block 内线程充分利用共享内存（shared memory）缓存部分数据，减少全局内存访问次数。
- **同步机制优化：**在每轮中间顶点 k 更新后使用 `hipDeviceSynchronize()` 保证全局完成，既保证正确性，也避免过度同步带来的性能浪费。
- **可调块尺寸：**针对不同规模的图，可调整线程块大小（`BLOCK_SIZE`）以获得最佳硬件占用率和执行效率。

通过上述优化策略，GPU 内核能够在保持正确性的前提下，实现对中小规模图的高效处理，并在大规模图上具有一定的加速效果。

6.3 优化结果

优化后数据如下：



```
Found 10 test cases in apsp/testcases
Running test [10]... PASS (0.47s)
Running test [1]... PASS (0.49s)
Running test [2]... PASS (0.49s)
Running test [3]... PASS (0.46s)
Running test [4]... PASS (0.49s)
Running test [5]... PASS (0.47s)
Running test [6]... PASS (0.49s)
Running test [7]... PASS (6.34s)
Running test [8]... PASS (2.69s)
Running test [9]... PASS (0.49s)
FINAL RESULT: ALL TESTS PASSED!
Passed cases: 10 / 10
Total execution time: 12.88 seconds
Job finished at Wed Sep 10 02:08:57 PM UTC 2025
```

分析数据：

表 3: APSP 测试结果

Test Case	Result	Time (s)
1	PASS	0.49
2	PASS	0.49
3	PASS	0.46
4	PASS	0.49
5	PASS	0.47
6	PASS	0.49
7	PASS	6.34
8	PASS	2.69
9	PASS	0.49
10	PASS	0.47
Total	10 / 10	12.88

这张表展示了 APSP 算法在 10 组测试用例上的运行结果与耗时情况。从结果来看，所有测试用例均通过验证，说明 GPU 实现的 APSP 算法在功能上是正确的（PASS 率为 100%）。在运行时间上，大多数测试用例耗时均在 0.46-0.49 秒之间，表现稳定且高效，表明算法对小规模或中等规模图的处理能力较强。然而，第 7 和第 8 个测试用例耗时明显偏高（分别为 6.34 秒和 2.69 秒），可能对应图规模较大或边数较多的输入，这显示出算法在处理大规模图时仍存在性能瓶颈。总计 10 个测试用例的累计运行时间为 12.88 秒，整体表现良好，但针对大规模输入，仍可进一步优化 GPU 内核和内存访问以降低运行时间。

7 总结与反思

本次 GPU 编程比赛涉及前缀和 (Prefix Sum)、Softmax 以及全源最短路径 (APSP) 三个核心算法, 实现了从串行到 GPU 并行的优化过程, 收获了丰富的经验和启示。

7.1 总体收获

- **GPU 并行设计能力提升:** 通过对前缀和、Softmax 和 APSP 的实现, 深入理解了 HIP 编程模型、线程映射、共享内存使用以及块内归约等 GPU 并行计算核心概念。
- **算法与硬件结合:** 学会根据 GPU 架构特点设计算法, 如利用共享内存减少全局内存访问、调整线程块大小以提高硬件占用率、通过并行归约优化 Softmax 和前缀和计算等。
- **性能优化意识:** 认识到单纯正确的算法并不等于高性能, 通过分析执行时间、识别瓶颈点 (如 APSP 在大规模图上的耗时) 来指导优化策略设计。
- **数值稳定性与精度控制:** 在 Softmax 实现中, 学习了如何处理指数溢出问题, 保证算法在大规模数据下的数值稳定性。

7.2 问题与反思

- **大规模图性能瓶颈:** APSP 在第 7 和第 8 个测试用例中耗时明显较高, 说明传统 Floyd-Warshall 算法在大规模图上存在计算复杂度瓶颈 ($O(m^3)$), 可考虑分块优化、共享内存更充分利用或改用 Johnson 算法等。
- **内存带宽限制:** GPU 内存访问连续性和共访 (coalesced access) 对于性能影响显著, 在设计大规模矩阵运算核时需要充分考虑。
- **并行化粒度选择:** 在 Softmax 和前缀和优化中, 选择合适的线程块大小和每线程处理元素数目对性能有重要影响, 过小或过大都会导致硬件利用率下降。
- **调试与验证:** GPU 并行程序调试较为复杂, 需要建立自动化验证机制 (如 compare.py 脚本) 保证功能正确性, 同时关注浮点比较的相对误差和绝对误差。

7.3 改进与展望

- 对于 APSP, 可尝试引入分块 Floyd-Warshall 或利用稀疏图优化策略, 降低大规模图计算时间。
- 对前缀和和 Softmax, 可进一步优化共享内存使用和循环展开 (loop unrolling), 提高 GPU 并行效率。
- 探索多 GPU 分布式计算和流式计算策略, 以应对更大规模数据和图计算需求。
- 加强对 GPU 性能分析工具 (如 rocprof) 的使用, 从硬件层面定位性能瓶颈, 指导算法优化。

总体而言, 本次比赛不仅检验了算法实现能力和 GPU 编程能力, 也加深了对高性能计算优化策略的理解, 为未来更复杂的并行算法开发打下了坚实基础。