基于 GPU 加速的数值积分方法研究

李纪元, 天津大学智算学部

摘要: 数值积分是数值计算领域的经典问题，使用等距节点的插值型求积公式是求解该问题的常用方法。本文利用 GPU 对使用牛顿-科斯公式的复化求积法、变步长积分法，以及荣姆伯格算法进行了加速，并设计实验评估了三种算法加速后在不同计算参数下的加速比与效率。经过实验发现，三种算法经过 GPU 加速后，平均加速比分别为 0.002，4.7275，3.819，并且随着计算的复杂度增高，加速比逐渐增大。此外，本文还设计实验计算了加速后算法运行过程中的数据传输开销，经过实验发现，传输开销在算法运行过程中的平均占比为 82%。

关键词: 数值计算; 数值积分; GPU

1 引言

定积分的计算求解有着非常广泛的应用，也是软件开发过程中的常见问题。手动求解复杂积分往往十分困难，比如，对于下面这个积分，其求解结果十分复杂 [1]：

$$\int_{-30}^{30} \frac{\sin\left(\sqrt{x}+2\right)\ \mathrm{e}^{\sqrt{x}}}{\sqrt{x}} dx = 17.89856517604524 + 73.53152791318399i$$

除了求解困难外，使用计算机求解积分的另一个问题是，计算机中的数据是离散的，无法处理连续的量。因此，在计算机中，往往通过求近似解来计算定积分。使用等距节点的插值型求积公式 [2] 是求解定积分的常用方法，比如，使用梯形公式、辛普森公式、科斯公式等。为了提高解的精度，研究者进一步在这些求积公式的基础上提出了复化求积法、变步长积分法，以及龙贝格算法等 [3]。为了让求积公式具有更高的代数精度，往往需要增多求积节点。但是，当求积节点较多时，直接使用上述方法进行计算会降低计算效率，尤其是使用变步长积分法与龙贝格算法时，求积节点在算法迭代过程中会呈指数型增长，严重降低了求解效率。因此，本文使用 GPU 并行计算技术对复化求积法、变步长积分法、荣贝格算法进行了加速，并设计实验评估了三种算法加速前后的性能进行了全面的评估。

总结来说，本文做出了如下贡献：

（1）实现了 GPU 加速后的复化求积法、变步长积分法、荣姆伯格算法，本文所涉及算法的代码实现均可在附件中找到。

（2）设计实验对三种算法的加速比、效率，以及传输数据的开销进行了评估。

2 研究背景

本节将介绍关于 GPU 与数值积分的基础知识，以便读者理解本文的 GPU 加速算法。

Author's Contact Information: 李纪元, 1511416271@qq.com, 天津大学智算学部.

2.1 GPU

GPU（Graphics Processing Units）[4] 是用于图形与视频渲染的处理器，由于其计算单元的占比远高于 CPU，故非常适合用于承担计算密集型工作。GPU 的基本单元是线程（Thread），线程可以组织成线程块 (Block)，线程块又可以组织成线程网格（Grid）。每个线程都有一个唯一标识的全局序号，可以通过线程网格、线程块、以及线程的相对坐标按如下公式计算得出：

$$idx = threadIdx.x + blockDim.x * blockIdx.x \tag{1}$$

其中，$threadIdx.x$ 是指线程在当前线程块中相对坐标的横坐标，$blockDim.x$ 是指线程块 x 维度的大小，$blockIdx.x$ 是指线程块在线程网格中相对坐标的横坐标。在得到线程的全局坐标后，就可以根据全局坐标分配每个线程所要执行的任务。

2.2 插值型积分公式

假设对函数 $f(x)$ 在区间 $[a,b]$ 上求积分，取区间 $[a,b]$ 上 $n+1$ 个点：$x_0 < x_1 < ... < x_n$，代入函数并构造插值多项式：

$$L_n(x) = \sum_{k=0}^{n} l_k(x) f(x_k) \tag{2}$$

其中 $l_k(x)$ 是插值函数。

如果将 $L_n(x)$ 作为 $f(x)$ 的近似函数，那么：

$$\int_a^b f(x)dx = \int_a^b L_n(x)dx = \sum_{k=0}^{n} \int_a^b l_k(x)dx f(x_k) = \sum_{k=0}^{n} A_k f(x_k) \tag{3}$$

其中 A_k 与被积函数 $f(x)$ 无关，仅与求积节点有关。该求积公式被称为插值型求积公式。

可以使用代数精度衡量求积公式的求积精度。对一个求积公式，如果其对于次数不大于 n 个多项式函数均成立，则其代数精度为 n，$n+1$ 个求积节点的代数精度至少为 n。

在上述过程中，如果使用拉格朗日插值基函数 [5]，就得到牛顿-科斯公式：

$$\int_a^b f(x)dx = (b-a) \sum_{k=0}^{n} C_k^{(n)} f(x_k) \tag{4}$$

其中 $C_k^{(n)}$ 为科斯系数。一般取两点（梯形公式）、三点（辛普森公式）、五点（科斯公式）求积节点的公式较为常用。

2.3 复化求积法

$n \geq 8$ 时，牛顿-科斯公式的稳定性会下降，所以一般不使用 $n \geq 8$ 的牛顿-科斯公式。为了解决这个问题，可以使用复化求积法，即，将求积区间分为多个小区间，然后每个区间上分别使用牛顿-科斯公式进行求解。

2.4 变步长积分法

直接使用牛顿-科斯公式或者复化求积法，需要事先确定步长。为了避免这一点，可以先用较大的步长求解积分，然后逐次将步长减半，直到求解结果逐渐稳定。

首先将求积区间 $[a, b]$ 分为 n 个子区间，每个区间选择两个求积节点（梯形公式），步长 h 为子区间长度，利用复化求积法得到：

$$T_n = \frac{h}{2} \sum_{k=0}^{n-1} [f(x_k) + f(x_{k+1})] \tag{5}$$

然后，将步长减半，得到：

$$T_{2n} = \frac{T_n}{2} + \frac{b-a}{2n} \sum_{k=0}^{n-1} f(x_{k+\frac{1}{2}}) \tag{6}$$

利用上述公式，可以迭代求解被积函数在 $[a, b]$ 上的积分，当 $|T\_2n - Tn|$ 小于给定范围时，停止迭代。

2.5 龙贝格算法

通过将变步长求积公式进行线性组合，可以在计算量更少的情况下得到更精确的结果。

首先，根据变步长积分法求出梯形序列 $T_n, T_{2n}, ...$

将梯形序列的元素进行线性组合，可以得到辛普森序列：

$$S_n = \frac{4}{3} T_{2n} - \frac{1}{3} T_n$$

再将辛普森序列进行线性组合，可以得到科斯序列：

$$C_n = \frac{16}{15} S_{2n} - \frac{1}{15} S_n$$

最后，将科斯序列进行线性组合，得到龙贝格序列：

$$R_n = \frac{64}{63} C_{2n} - \frac{1}{63} C_n$$

由于新序列与原序列的差距越来越小，一般只求到龙贝格序列为止。

3 方法

本节将分别介绍使用 GPU 对复化求积法、变步长积分法、龙贝格算法进行加速的具体过程。

3.1 基于 GPU 加速的复化求积法

Algorithm 1: GPU 加速的复化求积法

Input: *block_num*: 线程块数量,*thread_num*: 每块线程块的线程数,l, r: 积分区间

Result: *res*: 积分计算结果

```
1  res_device = transfer_arr_to_device();
2  step = (r - l) / (block_num * thread_num);
3  multi_Cotes «block_num, thread_num»(l, r, step, res_device);
4  res = sum(download_res_from_device(res_device));
5  Function multi_Cotes(l, r, step, res_device):
6      idx = threadIdx.x + blockDim.x * blockIdx.x;
7      cur_l = step * idx + l;
8      cur_r = cur_l + step;
9      h = (cur_r - cur_l) / 4;
10     res_device[idx] = (cur_r − cur_l)/90 × (7 × f(cur_l) + 32 × f(cur_l + h) + 12 ×
         f(cur_l + 2 × h) + 32 × f(cur_l + 3 × h) + 7 × f(cur_r));
```

算法 1 描述了经过 GPU 加速后的复化求积法运行过程。由于 GPU 不能直接使用主机端的存储空间，因此，在进行 GPU 上的计算前，必须先将数据传输到 GPU（第 1 行），GPU 完成计算后，也需要将结果从 GPU 传输回 CPU（第 4 行）。multi_Cotes 是主要的计算过程，该函数是在 GPU 上并行运行的。首先计算当前线程的全局序号（第 6 行），然后根据全局序号计算当前线程负责的子区间，以及在当前区间取点的步长（第 7-9 行），最后，根据得到的参数对当前区间应用五点牛顿-科斯公式，并将结果存入结果数组（第 10 行）。将结果数组传回主机端后，求和即可得到积分结果（第 4 行）。

3.2 基于 GPU 加速的变步长积分法

Algorithm 2: GPU 加速的变步长积分法

Input: *block_num*: 线程块数量,*thread_num*: 每块线程块的线程数,*l*, *r*: 积分区间

Result: *res*: 积分计算结果

```
1  res_device = transfer_arr_to_device();
2  step = (r - l) / (block_num * thread_num);
3  trapezium_integration «block_num, thread_num»(l, r, step, res_device);
4  last = (download_res_from_device(res_device))*(step / 2);
5  res = ∞;
6  while |res - last| > 10 do
7      last = res;
8      res_device = transfer_arr_to_device();
9      step = step / 2;
10     block_num = change_block_num_by_step(step, thread_num);
11     do_integration_after_change_step «block_num,thread_num»(l, r, step,
         res_device, block_num*thread_num);
12     res = last / 2 + (step / 2) * sum(download_res_from_device(res_device));
13 end
14 Function trapezium_integration(l, r, step, res_device):
15     idx = threadIdx.x + blockDim.x * blockIdx.x;
16     cur_l = l + idx * step;
17     cur_r = cur_l + step;
18     res_device[idx] = f(cur_r) + f(cur_l);
19 Function do_integration_after_change_step(l, r, step, res_device, N):
20     idx = threadIdx.x + blockDim.x * blockIdx.x;
21     if idx%2 == 0 or idx ≥ N then
22         return;
23     end
24     res_device[idx] = f(l + idx * step);
```

算法 2 描述了 GPU 加速后的变步长积分法的计算过程。首先通过使用梯形公式的复化积分法求出初始步长下的求积结果（第 3-4 行）。然后不断将步长减半（第 9 行），利用 GPU 计算划分新的子区间后，新加入的点的和（第 11 行），并将结果代入变步长

积分法公式得到新的步长下的求积结果（第 12 行）。当新步长下的结果与上次迭代的结果相差小于定值时，停止迭代。

3.3 基于 GPU 加速的龙贝格算法

Algorithm 3: GPU 加速的龙贝格法

Input: *block_num*: 线程块数量,*thread_num*: 每块线程块的线程数,*l*, *r*: 积分区间
Result: *T*: 龙贝格序列

```
1  T = get_sequence_using_vary_step_method«block_num, thread_num»(l, r);
2  i = 0;
3  while i < 3 do
4      block_num = get_block_num_by_len(len(T), thread_num);
5      T_cuda = transfer_arr_to_device();
6      tmp = transfer_arr_to_device();
7      romberg«block_num, thread_num»(T_cuda, tmp);
8      T = download_res_from_device(tmp);
9  end
10 Function romberg(T_cuda, tmp):
11     idx = threadIdx.x + blockDim.x * blockIdx.x;
12     if idx > len(T_cuda) then
13         return;
14     end
15     m = pow(4, idx + 1);
16     tmp[idx] = (m * T_cuda[idx + 1] - T_cuda[idx]) / (m - 1);
```

算法 3 描述了 GPU 加速后的龙贝格算法的运行过程。首先利用上一节所介绍的变步长积分法求出最开始的梯形序列（第 1 行）。然后，将该序列送入 GPU（第 7 行），每个 GPU 负责根据龙贝格算法计算公式，利用第 idx 和 $idx+1$ 项得到新序列的第 idx 项（第 10-16 行）。该过程重复三次，即可得到龙贝格序列（第 3-9 行）。

4 实现

本文主要借助 Python 3.12.3 与 Numba 0.59.1 进行开发，分别使用 51 行、118、165 行 Python 代码实现了加速前后的复化求积法、变步长积分法与龙贝格算法。

5 评估

本文将通过对 GPU 加速后的复化求积法、变步长积分法与龙贝格算法进行评估，来回答以下问题：

（1）使用 GPU 是否对本文所实现的三种算法在性能上有所优化？优化程度是多少？（5.2 节）

（2）使用 GPU 是否会带来其他方面的性能开销？这些开销占比多少？（5.3 节）

5.1 实验设置

本文的实验在一台搭载 Windows11 操作系统、12th Gen Intel(R) Core(TM) i7-12700H 处理器、16GB 内存、Nvidia Geforce RTX 3060 laptop GPU 的个人电脑上进行。

本文选择以下积分作为求积分的对象：

$$\int_{-20000}^{20000} 23\sin(x) + 21314\cos(x) - \frac{7}{2x^6+32} + \frac{1}{x^4+1} + 3x^4 dx$$

该函数包含多项式、分数、三角函数，手动求解定积分较为复杂。由于在使用不经过加速的变步长积分法与龙贝格算法时，有时会耗费较长时间，因此本文在对这两个算法进行评估时，令其中的变步长积分部分至多迭代 15 次，该设置不影响本文的评估结论。此外，为了令性能差距较为显著，本文使用 [-20000,20000] 作为求积区间。

为了去除随机性，本文对每种算法划分出不同子区间数量的情况（由于加速后每个子区间分配一个线程，因此子区间数量也是 GPU 线程数）均进行了实验，每个子区间数运行 3 次并取平均值。

5.2 性能评估

本文分别评估选取 1*4、2*8、4*16、8*32、16*64、16*128（线程块数 * 每块线程数）情况下三种算法加速前后的平均运行时间、加速比以及效率。

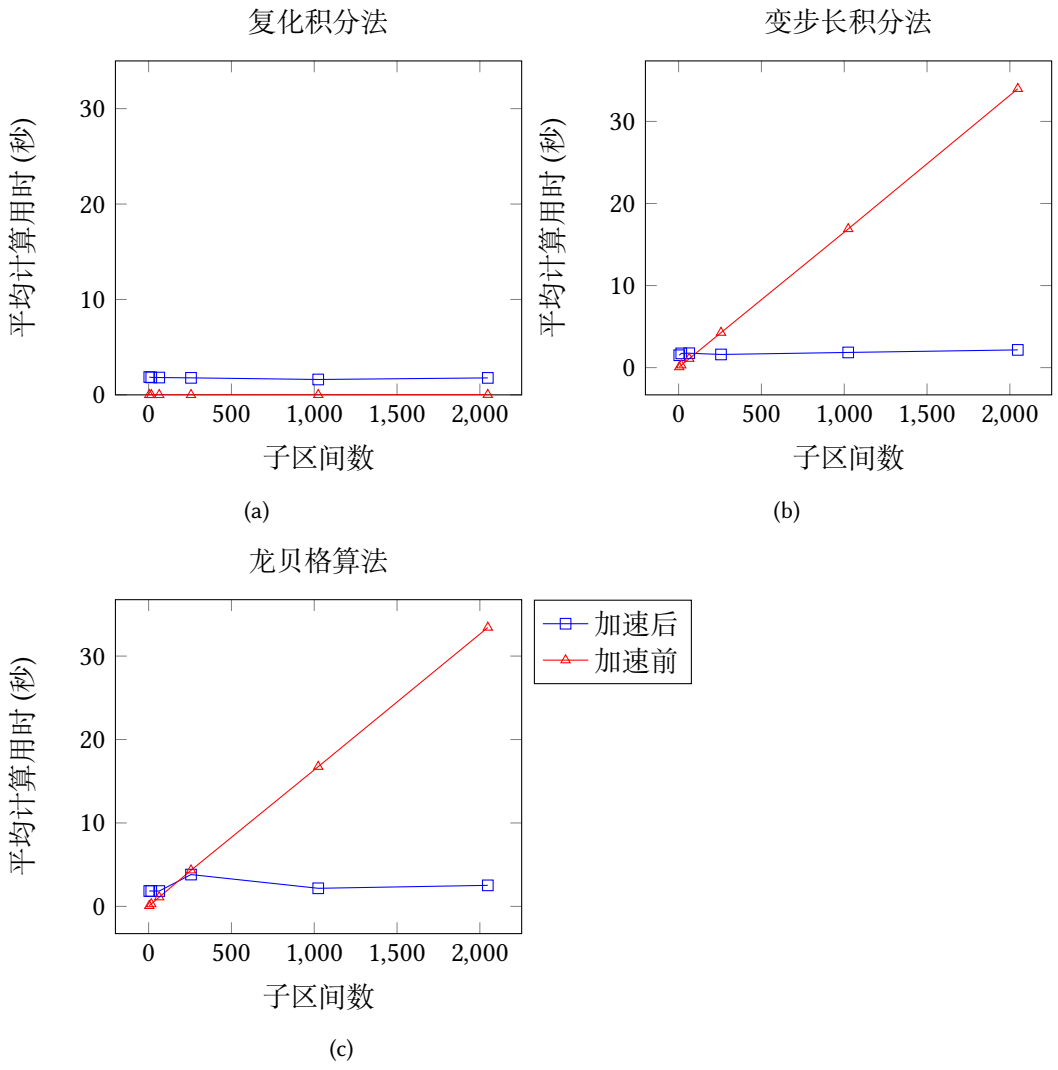

Fig. 1. **复化求积法、变步长积分法、龙贝格算法加速前后平均计算时长.**

图 1 展示了三种算法加速前后的平均计算时长。复化求积法并没有得到加速，反而性能变慢了。这是因为，复化求积法的形式本身比较简单，计算量也很小，因此，仅用 CPU 就足以完成计算。在使用 GPU 以后，反而因为传输数据的开销拖慢了运行速度。

变步长积分法在求积节点较少时，CPU 的速度是快于 GPU 的，但是由于其运行过程中求积节点会呈指数型增长，因此当求积节点增多时，GPU 便比 CPU 快得多了。龙贝格算法主要的时间开销也在复化求积法部分，因此结果和变步长积分法差距不大。

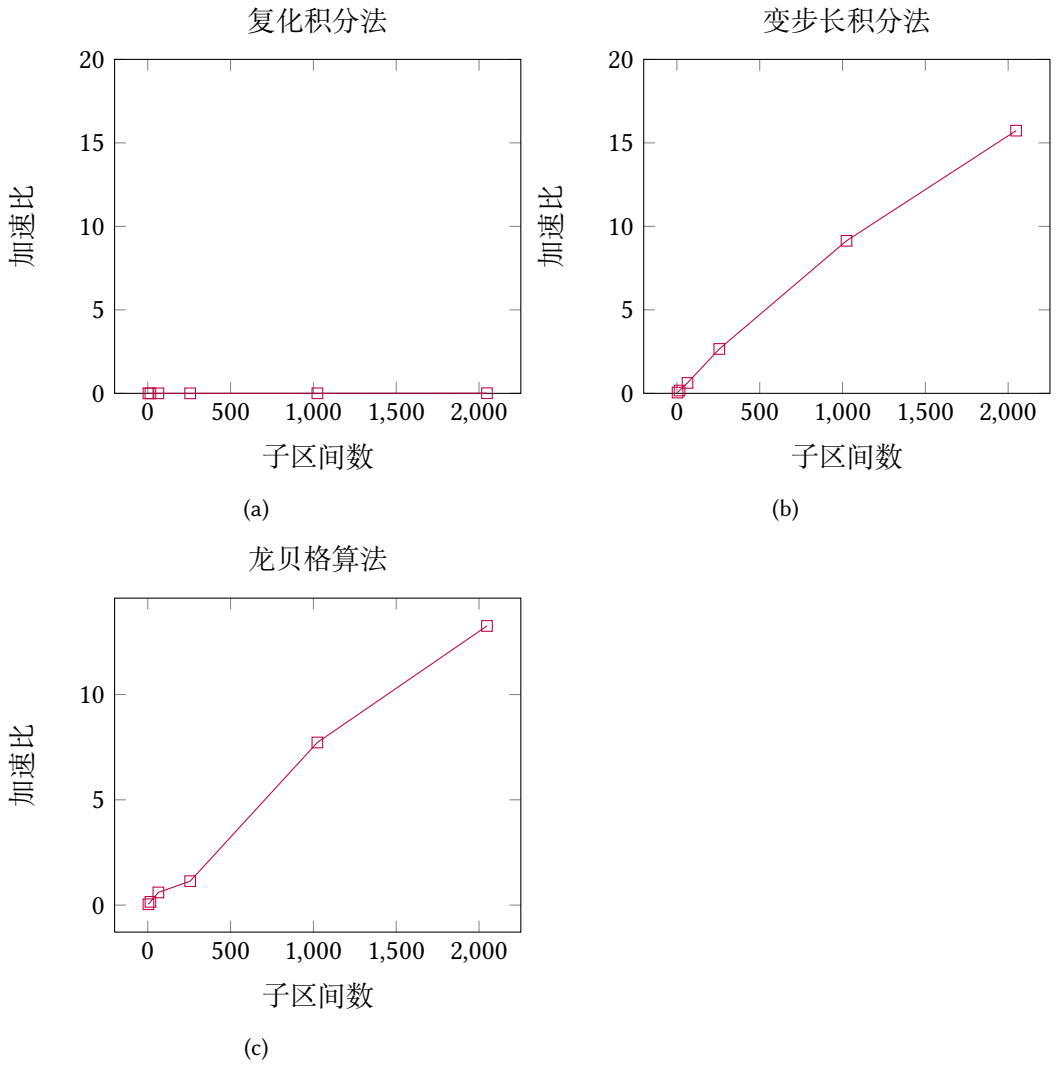

Fig. 2. **复化求积法、变步长积分法、龙贝格算法加速比.**

图 2 展示了三种算法的加速比。可以看出，当计算复杂度增高时（变步长积分法、龙贝格算法），加速比会大幅增加；反之，如果计算复杂度本身就很低，盲目使用 GPU 优化反而会降低运行效率（复化求积法）。

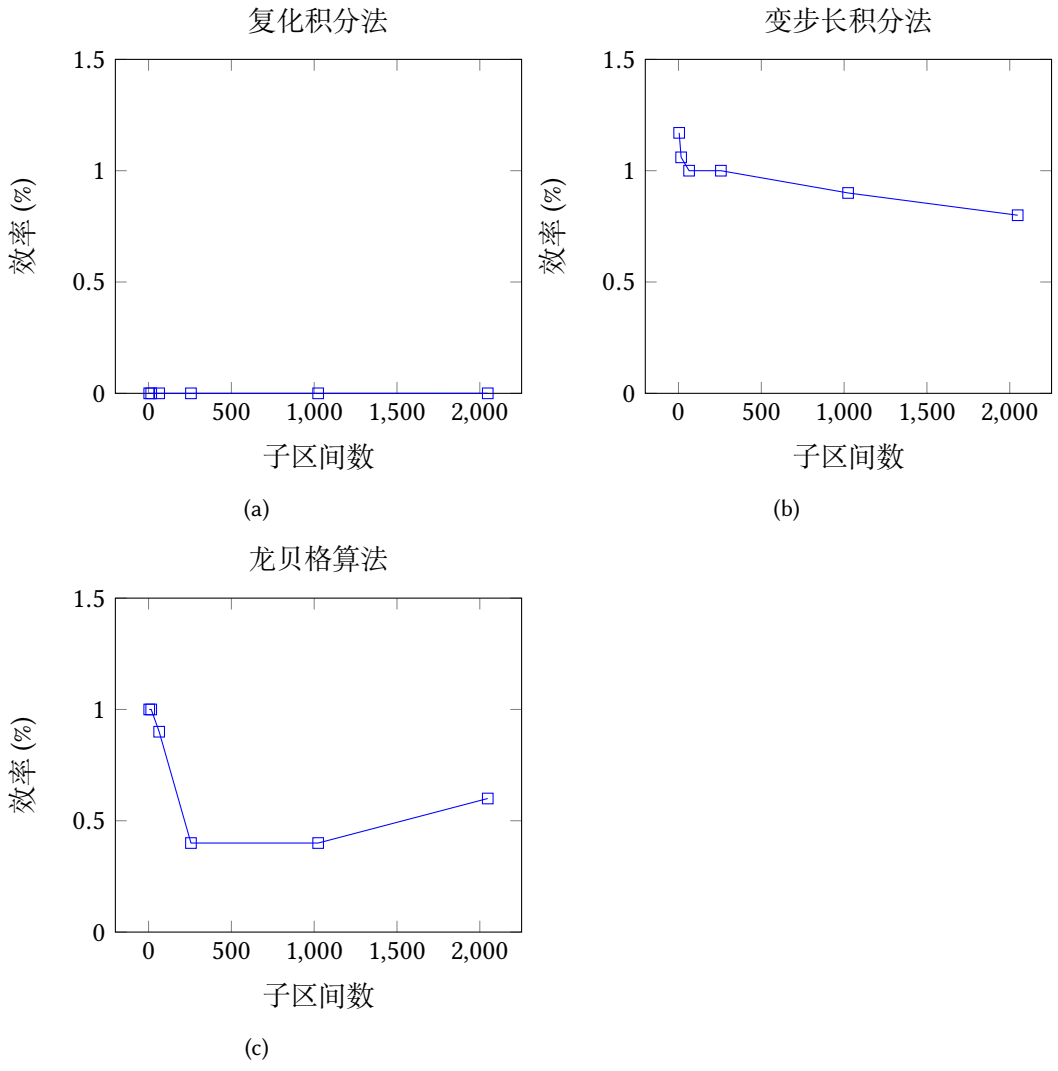

Fig. 3. **复化求积法、变步长积分法、龙贝格算法加速效率.**

图 3 展示了三种算法的加速效率。从总体上来说，三种算法的加速效率都很低，这意味着线程并没有得到充分的利用。这是因为使用 GPU 后，计算过程虽然已经非常快了，但是仍然有数据传输开销存在，拖慢了整体运行速度。即使线程再加数倍，这部分开销依然是无法避免的。本文将在下一节进行更详细的阐述。

5.3 GPU 性能开销评估

本文在实验过程中观察到，从主机端到设备端的数据传输开销占总体开销的一大部分。因此，本文以操作最复杂的龙贝格算法为例，研究了其在 1*4、2*8、4*16、8*32、16*64、16*128 线程下数据传输开销占总开销的比重。

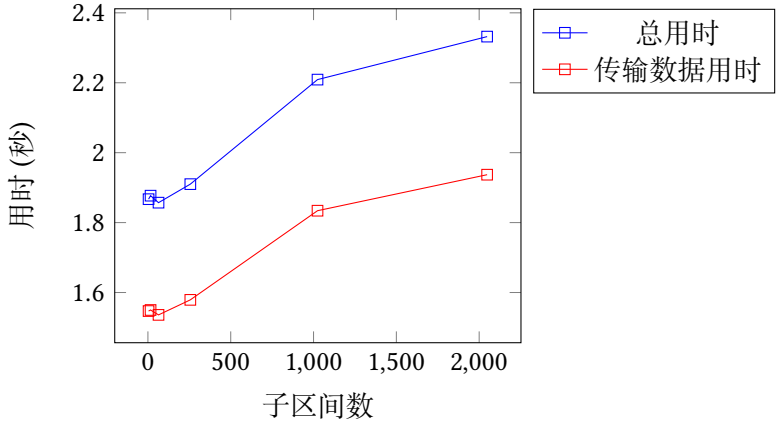

Fig. 4. **龙贝格算法总用时与传输数据用时.**

图 4 展示了不同线程数下数据传输开销占总开销的比重。随着子区间数（即线程数）的增加，数据传输开销也在以相同的趋势增加。并且，数据传输开销平均占总开销的 82%，这意味着大部分的时间都在进行主机端与设备端的数据传输，而真正的计算用时仅占不到 20%。该实验结果说明，尽量减少主机端与设备端的数据传输，是优化 GPU 并行计算的一大重点。

6 结论

本文针对复化积分法、变步长积分法与龙贝格算法三种数值积分方法设计了 GPU 优化算法。经过实验，使用 GPU 加速对变步长积分法、龙贝格算法有加速效果。本文还设计实验研究了数据传输开销占算法总开销的比重，经过实验发现该占比高达 82%，这说明优化数据传输过程是设计 GPU 并行计算算法的要点。

REFERENCES

[1] David Scherfgen.Integral Calculator[CP].(2024-04-24)[2024-04-24].https://www.integral-calculator.com/
[2] Wikipedia.Numerical integration[DB/OL].(2024-02-23)[2024-04-24].https://en.wikipedia.org/wiki/Numerical_integration.
[3] 天津大学数学系编写组. 工程数学基础教程. 天津: 天津大学出版社,2016.
[4] Wikipedia.Graphics processing unit[DB/OL].(2024-04-22)[2024-04-24].https://en.wikipedia.org/wiki/Graphics_processing_unit
[5] Wikipedia.Lagrange polynomial[DB/OL].(2024-04-13)[2024-04-24].https://en.wikipedia.org/wiki/Lagrange_polynomial