# 基于CUDA的并行粒子群优化算法研究及实现

陈 风 田雨波 杨 敏
(江苏科技大学电子信息学院 镇江 212003)

**摘 要** 应用图形处理器(GPU)来加速粒子群优化(PSO)算法并行计算时,为突出其加速性能,经常有文献以恶化CPU端PSO算法性能为代价。为了科学比较GPU-PSO算法和CPU-PSO算法的性能,提出用"有效加速比"作为算法的性能指标。文中给出的评价方法不需要CPU和GPU端粒子数相同,将GPU并行算法与最优CPU串行算法的性能作比较,以加速收敛到目标精度为准则,在统一计算设备架构(CUDA)下对多个基准测试函数进行了数值仿真实验。结果表明,在GPU上大幅增加粒子数能够加速PSO算法收敛到目标精度,与CPU-PSO相比,获得了10倍以上的"有效加速比"。

**关键词** 粒子群优化,并行计算,图形处理器,统一计算设备架构
**中图法分类号** TP301.6 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2014.09.050

## Research and Design of Parallel Particle Swarm Optimization Algorithm Based on CUDA

CHEN Feng TIAN Yu-bo YANG Min
(Department of Electronic Information, Jiangsu University of Science and Technology, Zhenjiang 212003, China)

**Abstract** In the application of graphic processing unit (GPU) to accelerate particle swarm optimization (PSO) algorithm for parallel computing, many references worsen the performance of PSO algorithm on CPU side in order to highlight the acceleration performance. The concept of "Effective Speedup" was proposed in this paper to measure the achievement of GPU-PSO algorithm and CPU-PSO algorithm. The proposed method aims at accelerating the implementation to the target precision. The GPU parallel algorithm was compared with the best CPU serial algorithm, which does not require the same number of particles between CPU side and GPU side. Experiments based on several benchmark test functions using compute unified device architecture (CUDA) show that substantially increasing the number of particles on GPU side can significantly accelerate the accomplishment of PSO algorithm to the target precision. Compared with CPU-PSO, an "Effective Speedup" of more than 10 has been achieved.

**Keywords** Particle swarm optimization (PSO), Parallel computing, Graphic processing unit (GPU), Compute unified device architecture (CUDA)

## 1 引言

粒子群优化(Particle Swarm Optimization, PSO)算法是Kennedy和Eberhart于1995年提出的一种智能优化算法[1]。PSO算法简单易实现,具备较强的全局搜索和收敛能力,因此发展十分迅速,在诸多领域得到成功应用[2]。面对计算复杂度较高的优化问题时,收敛时间长是PSO算法的一大问题。利用群体中个体行为的并行性,将PSO算法并行化加速是解决该问题的有效方法。相比用计算机群[3,4]、多核CPU[5]或FPGA等专业并行设备[6,7]加速PSO算法,利用图形处理器(Graphic Processing Unit, GPU)并行加速PSO算法[8-18]具备硬件成本低的显著优势。特别是2007年NVIDA公司推出了统一计算设备架构(Compute Unified Device Architecture, CUDA),它不需要借助复杂的图形学知识,具备良好的可编程性,迅速成为当前最为流行的GPU编程语言。

2009年,Veronese和Krohling首次应用CUDA实现了对PSO算法的加速[8],掀起了GPU加速PSO算法的研究热潮,涌现出各种GPU加速PSO算法的版本,既有gbest全局拓扑结构[8],也有lbest邻域拓扑结构[10];既有单个粒子群[8,10],也有多粒子群协作[11,12];既有1个线程对应1个粒子[8,10],也有1个线程对应1个粒子的其中1维[13];既有普遍使用的同步PSO,也有特殊的异步PSO[14]。国内文献对GPU加速PSO算法的研究相对较少。张庆科等在文献[15]中概述了CUDA架构下包括PSO算法在内的5种典型现代优化算法的并行实现过程,在并行PSO算法部分给出了文献[10]中的实验结果。蔡勇等近期在文献[16]中给出了并行PSO算法较详细的设计过程和优化思路,其取得了90倍的加速比。

传统意义上的加速比通常以能否加速执行到目标迭代次数为判断依据。增加粒子数或线程数是当前在GPU上实现

到稿日期:2013-11-25 返修日期:2014-03-03 本文受船舶工业国防科技预研基金项目(10J3.5.2)资助。
陈 风(1989-),男,硕士生,主要研究方向为高性能计算与计算智能,E-mail:1019758367@qq.com;田雨波(1971-),男,教授,硕士生导师,主要研究方向为计算智能及其电磁学应用;杨 敏(1989-),女,硕士生,主要研究方向为计算智能等。

PSO算法并行加速的主要方法之一。然而，一般情况下100～200个粒子数目已足够应付较复杂的优化问题，过多增加粒子数会显著恶化CPU端PSO算法的性能，造成加速比虚假现象。针对传统加速比指标的不合理之处，本文定义了“有效加速比”这一较为客观的加速指标，用GPU并行PSO算法与最优CPU串行PSO算法作性能比较，以能否加速收敛到目标精度为判断依据，并在CUDA架构下对Sphere、Rosenbrock、Rastrigrin、Griewangk这4个基准测试函数进行了数值测试。结果表明，在GPU上大幅增加粒子数能够加速PSO算法收敛到目标精度，与CPU-PSO相比，获得了10倍以上的“有效加速比”。

## 2 基于CUDA的并行PSO算法

### 2.1 标准PSO算法

本文使用的算法版本为带惯性权重、全局拓扑结构的PSO算法[19]。粒子群由$N$个粒子组成，每个粒子的位置代表优化问题在$D$维搜索空间中的一个潜在的解。算法的速度更新和位置更新公式如下：

$$V_{id}(t+1)=wV_{id}(t)+c_1r_1(P_{id}(t)-X_{id}(t))+c_2r_2(P_{gd}(t)-X_{id}(t)) \quad (1)$$

$$X_{id}(t+1)=X_{id}(t)+V_{id}(t+1) \quad (2)$$

其中，$i=1,2,\cdots,N$，$d=1,2,\cdots,D$；$c_1$和$c_2$是学习因子，非负的常数；$r_1$和$r_2$是介于[0,1]的均匀分布的随机数；$V_{id}(t)\in[-V_{max},V_{max}]$，$V_{max}$限制了粒子飞行的最大速度，$X_{id}(t)\in[-X_{max},X_{max}]$，$X_{max}$限定了粒子搜索空间的范围，可设定$V_{max}=kX_{max}$，$0\leqslant k\leqslant 1$；$w$是惯性权重，介于[0,1]，用来平衡粒子的全局探索能力和局部开发能力。

标准PSO优化算法的流程如下：

(a)随机初始化每个粒子的位置$V_{id}(t)$和速度$X_{id}(t)$。

(b)初始化个体最优位置$P_{id}(t)$和全局最优位置$P_{gd}(t)$。

(c)更新每个粒子的速度$V_{id}(t)$和位置$X_{id}(t)$。

(d)计算每个粒子对应的适应度值$F(X_i)$。

(e)更新个体最优位置$P_{id}(t)$及对应的适应度值$F(P_i)$；更新全局最优位置$P_{gd}(t)$及对应的适应度值$F(P_g)$。

(f)若满足算法结束条件，则结束，否则返回步骤(c)。

### 2.2 CUDA编程模型

CUDA编程模型将CPU作为主机，GPU作为协处理器，两者协同工作，各司其职。CPU负责进行逻辑性强的事务处理和串行计算，GPU则专注于执行高度线程化的并行处理任务。CUDA采用单指令多线程(Single Instruction Multiple Threads，SIMT)执行模式。内核函数(kernel)执行GPU上的并行计算任务，是整个程序中的一个可以被并行执行的步骤。CUDA将线程组织成块网格(Grid)、线程块(Block)、线程(Thread)这3个不同的层次，并采用多层次的存储器结构：只对单个线程可见的本地存储器、对块内线程可见的共享存储器、对所有线程可见的全局存储器等。一个kernel函数中存在两个层次的并行，即同一Grid中的Block之间不需要通信的粗粒度并行，同一Block中的Thread之间允许通信的细粒度并行。CUDA计算流程通常包括CPU到GPU数据传递、kernel函数执行、GPU到CPU数据传递3个步骤。

### 2.3 基于CUDA的并行PSO算法设计

标准PSO算法的可并行性本质上是因为群体中个体行为的并行性，体现在以下(a)(b)(c)3个方面，以及CUDA架构特有的并行性，体现在(d)：

(a)粒子速度和位置的更新是可并行的；

(b)粒子适应度值的计算是可并行的；

(c)粒子个体最优位置的更新是可并行的；

(d)粒子全局最优位置更新中寻找最小适应度值，可利用CUDA架构下的并行规约(Reduction)算法。

每次迭代过程中(a)(b)(c)(d)4步必须依次顺序执行，因为下一步的运算依赖于上一步的结果。所有粒子个体最优更新完成后，才能进行全局最优更新，即步骤(c)后必须执行所有线程同步(对应kernel函数的结束)操作后才能执行步骤(d)，因此一次迭代过程至少需要2个kernel函数(第一个kernel函数对应步骤(a)(b)(c)，第二个kernel函数对应步骤(d))。考虑到通俗简明和可扩展性，可将第一个kernel函数拆分成多个kernel函数。

根据以上分析，可采用粒子与线程一一对应的并行策略，GPU-PSO的算法流程设计如图1所示。

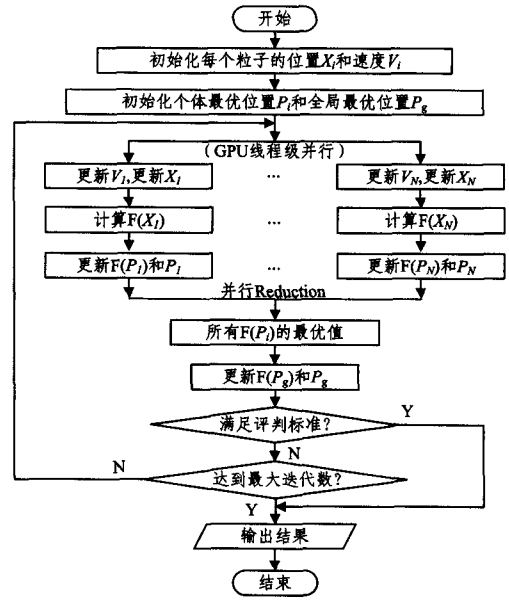


图1 基于CUDA架构的并行PSO算法流程图

GPU-PSO算法的步骤如下：

(1)CPU端调用malloc()函数和cudaMalloc()函数，分别在CPU端和GPU端分配变量空间。

(2)CPU端初始化粒子的位置、速度等信息。

(3)CPU端调用cudaMemcpy()函数，将CPU端粒子信息传至GPU显存。

(4)CPU端调用kernel函数，执行GPU上的并行计算任务。

(5)CPU端调用cudaMemcpy()函数，将GPU端有用信息传回至CPU端。

(6)CPU端调用free()函数和cudaFree()函数，释放CPU端和GPU端已分配的变量空间。

以上步骤中，步骤(4)是GPU-PSO算法的核心，每次迭代设计成4个依次顺序执行的kernel函数(分别对应(a)(b)(c)(d)4个并行方面)，其伪代码如下：

```
for(i=0;i< generationsNumber;i++ )
```

```
{
〈Update velocity and position of each particle〉 // kernel 1
〈Compute fitness of each particle〉 // kernel 2
〈Update pbest of each particle〉 // kernel 3
〈Update gbest of all particles〉 // kernel 4
}
```

以上伪代码中的 kernel 4 需要找出适应度值最小的粒子编号，这对单线程算法来说是非常简单的任务，在大规模并行架构上实现时却会变成一个较为复杂的问题。当粒子数大于块内最大线程数 1024(计算能力 2.0 及以上)或 512(计算能力 2.0 以下)时，在 CUDA 架构上需用 2 次并行 Reduction 实现，程序具体实现时分为 2 个 kernel。第 1 个 kernel 启动等于粒子数的线程数，找到各个线程块中的最小值；第 2 个 kernel 启动等于第 1 个 kernel 中线程块数的线程数，找到这些最小值的最小值，即当前全局最优值。当前全局最优值再与旧的全局最优值对比，决定是否需要更新。不能只使用 1 个 kernel 的原因在于：CUDA 架构能通过调用__syncthreads()使线程块内的线程同步，但不能使所有线程同步，所有线程的同步只能通过 kernel 的结束来保证。当粒子数小于等于块内最大线程数 1024 或 512 时，在 kernel 函数中启动等于粒子数的线程数做 1 次并行 Reduction 即可。

与以计算机集群为代表的其他并行计算方式相比，基于 CUDA 架构的并行 PSO 算法除了硬件成本低，最大的特点和优点是适应需要较多粒子数的复杂优化问题。在计算机集群的并行模式中，通常一台计算机计算一个粒子，并行度一般为几十；而基于 CUDA 的并行模式中，一个线程计算一个粒子，并行度可以达到上百乃至更高，本文通过在 GPU 上大幅增加粒子数能获得“有效加速比”也正是利用了这一特点(见 4.3 节)。另外，在计算机集群的并行模式中，计算机之间的通信是一个比较复杂的问题；而基于 CUDA 的并行模式中，通信时间开销要小得多。

## 3 GPU-PSO 算法性能优化

### 3.1 粒子(线程)数目和线程块大小的设计

一个线程束(Warp)包含索引相邻的 32 个线程，流多处理器(Stream Multiprocessor，SM)以 Warp 为单位调度和执行线程，因此将粒子数目和线程块大小都设计成 32 的倍数值。具体实现时，粒子数目取 128 的倍数值，每个线程块中的线程数目在 kernel 1、kernel 2、kernel 3 中可取 128、192、256 这样的典型值，在 kernel 4 中为了充分利用共享内存，第一个 Reduction 时线程数尽量取大(计算能力 2.0 及以上的块内最大线程数为 1024，计算能力 2.0 以下的块内最大线程数为 512)，第二个 Reduction 时取第一个 Reduction 的线程块数。

### 3.2 最小化线程分支

SIMT 执行模式会导致线程分支(Thread Divergence)特别耗时，应尽量减少 Warp 内的分支数目。以 kernel 4 中的并行规约为例(实验中至少有 128 个线程，这里简单起见只列出 8 个线程)，图 2 的方案具有明显的线程分支，在第一次求 min 过程中，只有那些索引为偶数的线程才执行求 min，相邻线程行为不同。图 3 的方案分支就较少，表现在相邻线程行为相同，都求 min 或者都不求 min。

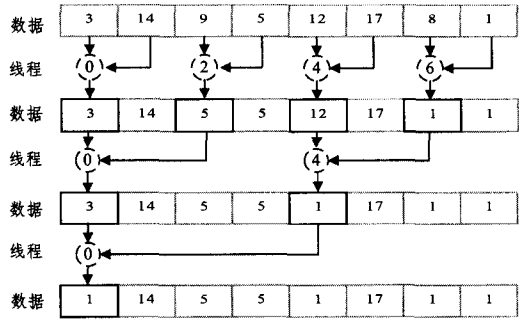


图 2 大量线程分支的并行规约求极值方案

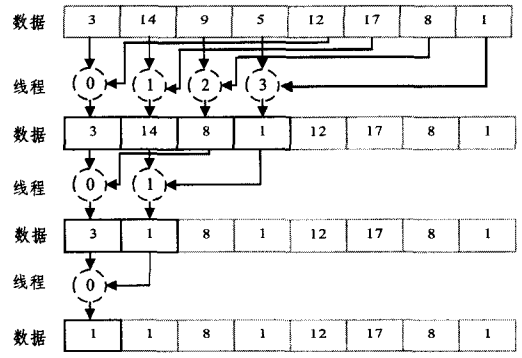


图 3 最小化线程分支的并行规约求极值方案

### 3.3 合并访问全局存储器

粒子位置、速度等信息在 CPU 内存中以二维形式存储，见图 4(图中 $d=D-1$，$n=N-1$)，而 GPU 全局内存是一维形式，将粒子位置、速度等信息在 GPU 全局内存中布局涉及到合并访问(Coalesced Access)的问题。合并访问的原理这里不具体展开，简单地说，相邻的线程访问相邻的数据，即可满足合并访问的要求。合并访问能使传输数据时的速度接近全局存储器带宽的峰值。粒子位置信息在 GPU 全局内存中按粒子顺序存储(文献[13,16]等采用的就是这种方式)，如图 5 所示，虽然简单直观但不符合合并访问的要求，会造成访存效率大幅下降。这里采用文献[17]所述的存储布局方法，如图 6 所示，访存时同时访问各个粒子的同一维，满足合并访问的条件，提高了访存效率。粒子速度信息的存储布局与粒子位置信息类似。

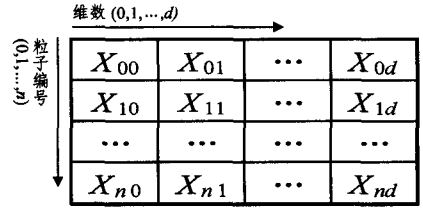


图 4 粒子位置信息在 CPU 内存中的存储

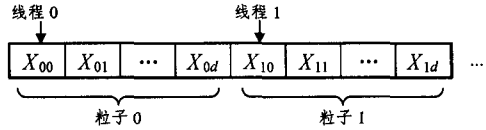


图 5 粒子位置信息在 GPU 全局内存中的存储(没有合并访问)

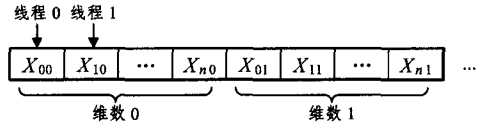


图 6 粒子位置信息在 GPU 全局内存中的存储(合并访问)

### 3.4 最大化使用共享存储器

每个 SM 提供最多 48kB 的共享存储器，比全局存储器的访问速度快得多，但只对块内线程可见。因此应尽量使用共享存储器来保存全局存储器中在 kernel 函数的执行阶段需要频繁使用的那部分数据。以 kernel 4 中的并行规约为例，每次规约时先将全局内存中的数据保存至共享内存，规约时反复使用共享内存上的数据，规约完成后再将共享内存上的结果保存至全局内存。

### 3.5 最小化 CPU 和 GPU 之间的数据传输

GPU 上执行 PSO 的粒子速度更新需要大量的随机数。早期的 GPU 上没有自带的随机数生成库，需要将 CPU 产生的随机数传至 GPU（传输时间降低计算性能）[10]，或编写 GPU 随机数生成函数（使用不方便）[18]。目前，可以使用 CURAND 库中的 curand_uniform()函数在 GPU 上产生随机数，这样使整个迭代过程都在 GPU 上完成，避免在 CPU 和 GPU 之间频繁传输数据带来的时间损耗。

## 4 实验结果与分析

### 4.1 数值实验所用基准函数及试验平台

本文采用 Sphere、Rosenbrock、Rastrigrin、Griewangk 这 4 个基准测试函数对 GPU-PSO 算法和 CPU-PSO 算法进行性能测试。测试函数的表达式如式（3）－式（6）所示。Sphere、Rosenbrock、Rastrigrin、Griewangk 这 4 个函数的目标精度均为 $10^{-6}$，其维数 $D$ 分别为 120、6、8、32。惯性权重 $w$ 在 Sphere 函数中取值 0.32，在其余函数均取 0.9 至 0.4，线性递减。学习因子 $c_1$ 和 $c_2$ 均取 2.05。$k$ 取 0.5。

$$f_{Sphere}(\boldsymbol{x})=\sum_{d=1}^{D}x_d^2,\ -100\leqslant x_d\leqslant 100 \tag{3}$$

$$f_{Rosenbrock}(\boldsymbol{x})=\sum_{d=1}^{D-1}[100(x_{d+1}-x_d^2)^2+(x_d-1)^2],\ -10\leqslant x_d\leqslant 10 \tag{4}$$

$$f_{Rastrigrin}(\boldsymbol{x})=\sum_{d=1}^{D}(x_d^2-10\cos(2\pi x_d)+10),\ -5.12\leqslant x_d\leqslant 5.12 \tag{5}$$

$$f_{Griewangk}(\boldsymbol{x})=\frac{1}{4000}\sum_{d=1}^{D}x_d^2-\prod_{d=1}^{D}\cos\left(\frac{x_d}{\sqrt{d}}\right)+1,\ -600\leqslant x_d\leqslant 600 \tag{6}$$

在实验过程中，为了避免 CPU 端过长的执行时间，方便实验测试，这里的维数取值偏小，但默认这些测试函数是比较复杂的优化问题，在 CPU 端适合 100 个以上的粒子数目来求解。实验所采用的计算平台如表 1 所列。CPU 和 GPU 中的小数都使用单精度浮点型（float）表示。GPU 中三角函数使用的是软件库中的 cosf()函数，而不是更快速但降低精度的硬件函数__cosf()。为保证结果的可靠性，实验数据为 20 次实验去掉最大值和最小值之后 18 次的平均值。

表 1 计算平台

| 名称 | 型号 |
|---|---|
| CPU | Intel Core i3-2100，3.1GHz |
| GPU | NVIDIA Tesla K20c，706MHz，2496 CUDA Cores，计算能力 3.5 |
| 操作系统 | Windows 7 中文专业版 |
| 开发环境 | Microsoft Visual C++ 2010，CUDA 5.0 |

### 4.2 等粒子数等迭代次数加速比

“等粒子数等迭代次数加速比”$S_{iteration}$ 是最常用的加速性能指标，定义为 PSO 算法在相同的粒子数和相同的迭代次数下 CPU 程序运行时间 $T_{cpu\text{-}iteration}$ 和 GPU 程序运行时间 $T_{gpu\text{-}iteration}$ 的比值：

$$S_{iteration}=\frac{T_{cpu\text{-}iteration}}{T_{gpu\text{-}iteration}} \tag{7}$$

具体测试时，不同粒子数对应的迭代次数动态改变以减少 CPU 端执行时间。实验结果如表 2－表 5 所列。

表 2 Sphere 函数等粒子数等迭代次数加速比

| 粒子数 | 迭代次数 | 运行时间/s | | 加速比 |
|---|---|---|---|---|
| | | CPU | GPU | |
| 128 | 5000 | 7.156 | 3.618 | 1.987 |
| 256 | 4000 | 11.368 | 2.911 | 3.905 |
| 512 | 3000 | 17.270 | 2.377 | 7.264 |
| 1024 | 3000 | 36.779 | 2.519 | 14.594 |
| 2048 | 2000 | 49.965 | 1.751 | 28.521 |
| 4096 | 2000 | 105.898 | 1.807 | 58.590 |
| 8192 | 1500 | 161.342 | 1.472 | 109.561 |
| 16384 | 1000 | 213.408 | 1.144 | 186.579 |
| 32768 | 1000 | 441.650 | 2.134 | 206.887 |
| 65536 | 1000 | 883.412 | 3.715 | 237.764 |
| 131072 | 800 | 1408.179 | 5.408 | 260.357 |

表 3 Rosenbrock 函数等粒子数等迭代次数加速比

| 粒子数 | 迭代次数 | 运行时间/s | | 加速比 |
|---|---|---|---|---|
| | | CPU | GPU | |
| 128 | 100000 | 7.364 | 10.276 | 0.717 |
| 256 | 80000 | 11.488 | 8.369 | 1.372 |
| 512 | 60000 | 17.044 | 6.340 | 2.688 |
| 1024 | 40000 | 22.499 | 4.249 | 5.295 |
| 2048 | 30000 | 33.434 | 3.245 | 10.302 |
| 4096 | 20000 | 45.018 | 2.210 | 20.362 |
| 8192 | 10000 | 46.714 | 1.174 | 39.772 |
| 16384 | 8000 | 78.129 | 1.061 | 73.602 |
| 32768 | 6000 | 137.373 | 1.112 | 123.465 |
| 65536 | 4000 | 189.028 | 1.035 | 182.521 |
| 131072 | 2000 | 193.016 | 0.840 | 229.605 |

表 4 Rastrigrin 函数等粒子数等迭代次数加速比

| 粒子数 | 迭代次数 | 运行时间/s | | 加速比 |
|---|---|---|---|---|
| | | CPU | GPU | |
| 128 | 40000 | 4.910 | 6.163 | 0.797 |
| 256 | 20000 | 4.844 | 3.146 | 1.540 |
| 512 | 10000 | 4.808 | 1.605 | 2.995 |
| 1024 | 5000 | 4.837 | 0.818 | 5.910 |
| 2048 | 4000 | 7.791 | 0.694 | 11.226 |
| 4096 | 3000 | 11.605 | 0.552 | 20.994 |
| 8192 | 2000 | 15.700 | 0.427 | 36.695 |
| 16384 | 1000 | 16.115 | 0.237 | 67.882 |
| 32768 | 1000 | 32.433 | 0.387 | 83.772 |
| 65536 | 1000 | 64.444 | 0.578 | 111.401 |
| 131072 | 800 | 107.656 | 0.793 | 135.620 |

表 5 Griewangk 函数等粒子数等迭代次数加速比

| 粒子数 | 迭代次数 | 运行时间/s | | 加速比 |
|---|---|---|---|---|
| | | CPU | GPU | |
| 128 | 20000 | 20.694 | 8.478 | 2.441 |
| 256 | 10000 | 20.664 | 4.310 | 4.794 |
| 512 | 8000 | 33.125 | 3.472 | 9.540 |
| 1024 | 6000 | 49.529 | 2.627 | 18.851 |
| 2048 | 5000 | 83.032 | 2.486 | 33.399 |
| 4096 | 4000 | 136.096 | 2.211 | 61.528 |
| 8192 | 3000 | 204.814 | 1.895 | 108.050 |
| 16384 | 2000 | 277.687 | 1.336 | 207.755 |
| 32768 | 2000 | 560.219 | 2.519 | 222.360 |
| 65536 | 1500 | 822.146 | 2.947 | 278.907 |
| 131072 | 1000 | 1108.854 | 3.451 | 321.222 |

分析表 2－表 5 数据可知，Sphere、Rosenbrock、Rastrigrin、Griewangk 这 4 个基准测试函数最大分别取得了 260.4、229.6、

135.6、321.2 的“等粒子数等迭代次数加速比”。当粒子数小于等于 16384(实验所用 GPU 的最大驻留线程数为 26624)时,随着粒子数(线程数)的翻倍,加速比大致呈翻倍趋势;当粒子数大于等于 32768 时,加速比呈放缓增加趋势。但是,这种加速比的不断增长在相当程度上是依靠“恶化 CPU 程序性能”而获得。以 Sphere 函数为例,128 个粒子和 131072 个粒子在迭代次数内都至少达到了 $10^{-6}$ 的目标精度,但前者只用了 7.2 秒,后者却用了 1408.2 秒。这个问题产生的本质在于:一般情况下 100～200 个粒子数目已足够应付较复杂的优化问题,过多增加粒子数会显著恶化 CPU 算法性能,造成加速比虚假现象。文献[16]中虽然获得了 90 倍的加速比,但是使用的粒子数目为 400、1200、2000、2800、5000,就算是最小的粒子数目 400 也足够恶化对应的 CPU 程序。要真正反映加速性能,就要将最优 CPU 串行 PSO 算法的性能与 GPU 并行 PSO 算法的作比较,以能否加速收敛到目标精度而不是能否加速执行到目标迭代次数为判断依据。

### 4.3 等粒子数等精度加速比和有效加速比

首先将“等粒子数等精度加速比”$S_{precisom}$ 定义为相同粒子数时在达到精度要求(实验中取 $10^{-6}$)时 CPU 程序运行时间 $T_{cpu\text{-}precision}$ 和 GPU 程序运行时间 $T_{gpu\text{-}precision}$ 的比值:

$$S_{precision}=\frac{T_{cpu\text{-}precision}}{T_{gpu\text{-}precision}} \tag{8}$$

将所有 CPU 程序运行时间的最小值 $\min(T_{cpu\text{-}precision})$ 认为是最优 CPU 程序的执行时间(实验中为 128 个粒子对应的 CPU 程序执行时间),并定义“有效加速比”$S_{effectiveness}$ 如下:

$$S_{effectiveness}=\frac{\min(T_{cpu\text{-}precision})}{T_{gpu\text{-}precision}} \tag{9}$$

实验结果如表 6－表 9 所列。

表 6 Sphere 函数等粒子数等精度加速比和有效加速比

| 粒子数 | 达到精度要求 | | | | 加速比 | 有效加速比 |
|---|---|---|---|---|---|---|
| | CPU | | GPU | | | |
| | 迭代次数 | 运行时间/s | 迭代次数 | 运行时间/s | | |
| 128 | 4036 | 5.787 | 2721 | 1.969 | 2.938 | 2.938 |
| 256 | 2958 | 8.409 | 2213 | 1.611 | 2.219 | 3.592 |
| 512 | 1790 | 10.322 | 1878 | 1.488 | 6.933 | 3.887 |
| 1024 | 1257 | 15.439 | 1566 | 1.315 | 11.734 | 4.398 |
| 2048 | 968 | 24.324 | 1294 | 1.133 | 21.460 | 5.106 |
| 4096 | 810 | 43.180 | 1091 | 0.986 | 43.768 | 5.866 |
| 8192 | 660 | 71.503 | 943 | 0.926 | 77.216 | 6.249 |
| 16384 | 472 | 101.204 | 753 | 0.862 | 117.326 | **6.709** |
| 32768 | 432 | 192.383 | 621 | 1.326 | 145.082 | 4.364 |
| 65536 | 390 | 346.949 | 563 | 2.094 | 165.663 | 2.763 |
| 131072 | 334 | 599.622 | 493 | 3.337 | 179.638 | 1.734 |

表 7 Rosenbrock 函数等粒子数等精度加速比和有效加速比

| 粒子数 | 达到精度要求 | | | | 加速比 | 有效加速比 |
|---|---|---|---|---|---|---|
| | CPU | | GPU | | | |
| | 迭代次数 | 运行时间/s | 迭代次数 | 运行时间/s | | |
| 128 | 65164 | 4.728 | 66269 | 6.810 | 0.694 | 0.694 |
| 256 | 51299 | 7.307 | 51540 | 5.391 | 1.355 | 0.877 |
| 512 | 38963 | 11.131 | 39373 | 4.160 | 2.675 | 1.136 |
| 1024 | 25898 | 14.611 | 25675 | 2.727 | 5.357 | 1.736 |
| 2048 | 18596 | 20.828 | 19104 | 2.066 | 10.078 | 2.288 |
| 4096 | 11549 | 26.310 | 12196 | 1.348 | 19.514 | 3.507 |
| 8192 | 2014 | 9.621 | 6512 | 0.764 | 12.578 | 6.181 |
| 16384 | 771 | 7.830 | 3906 | 0.518 | 15.105 | 9.121 |
| 32768 | 629 | 14.502 | 3010 | 0.558 | 25.980 | 8.470 |
| 65536 | 588 | 27.921 | 1922 | 0.497 | 56.093 | 9.498 |
| 131072 | 507 | 48.995 | 945 | 0.397 | 123.234 | **11.892** |

表 8 Rastrigrin 函数等粒子数等精度加速比和有效加速比

| 粒子数 | 达到精度要求 | | | | 加速比 | 有效加速比 |
|---|---|---|---|---|---|---|
| | CPU | | GPU | | | |
| | 迭代次数 | 运行时间/s | 迭代次数 | 运行时间/s | | |
| 128 | 11850 | 1.473 | 12134 | 1.869 | 0.788 | 0.788 |
| 256 | 6530 | 1.603 | 6591 | 1.036 | 1.545 | 1.421 |
| 512 | 3653 | 1.788 | 3661 | 0.587 | 3.042 | 2.506 |
| 1024 | 2100 | 2.077 | 2039 | 0.333 | 6.222 | 4.413 |
| 2048 | 1569 | 3.093 | 1653 | 0.286 | 10.780 | 5.134 |
| 4096 | 542 | 2.172 | 1289 | 0.237 | 9.139 | 6.198 |
| 8192 | 404 | 3.252 | 909 | 0.194 | 16.708 | 7.568 |
| 16384 | 219 | 3.619 | 523 | 0.124 | 29.127 | 11.855 |
| 32768 | 181 | 5.997 | 304 | 0.117 | 50.888 | **12.499** |
| 65536 | 164 | 10.722 | 297 | 0.172 | 62.255 | 8.553 |
| 131072 | 116 | 16.249 | 247 | 0.245 | 66.180 | 5.999 |

表 9 Griewangk 函数等粒子数等精度加速比和有效加速比

| 粒子数 | 达到精度要求 | | | | 加速比 | 有效加速比 |
|---|---|---|---|---|---|---|
| | CPU | | GPU | | | |
| | 迭代次数 | 运行时间/s | 迭代次数 | 运行时间/s | | |
| 128 | 11496 | 11.955 | 11789 | 4.997 | 2.392 | 2.392 |
| 256 | 6613 | 13.703 | 6339 | 2.732 | 3.179 | 4.375 |
| 512 | 5789 | 24.118 | 5061 | 2.196 | 10.979 | 5.442 |
| 1024 | 3749 | 31.159 | 3873 | 1.696 | 18.368 | 7.047 |
| 2048 | 1120 | 18.884 | 3194 | 1.588 | 11.889 | 7.527 |
| 4096 | 1008 | 34.728 | 2571 | 1.422 | 24.420 | 8.406 |
| 8192 | 716 | 49.423 | 1961 | 1.239 | 39.879 | 9.646 |
| 16384 | 665 | 93.170 | 1357 | 0.907 | 102.687 | **13.176** |
| 32768 | 399 | 112.599 | 1327 | 1.672 | 67.313 | 7.147 |
| 65536 | 323 | 178.141 | 1030 | 2.025 | 87.952 | 5.902 |
| 131072 | 267 | 305.934 | 734 | 2.533 | 120.767 | 4.719 |

分析表 6－表 9 中的数据可得如下结论:

(a)随着粒子数的不断增多,到达设定精度所需的平均迭代次数不断减少,但 CPU 程序和 GPU 程序各自的平均迭代次数有一定差别(总体上粒子数较少时差别较小,粒子数较多时差别较大,原因可能是 CPU 和 GPU 计算精度累计、随机数库的质量等,本文不对此进行深入讨论)。因此,相对于传统意义上的“等粒子数等迭代次数的加速比”,“等粒子数等精度的加速比”是更客观的性能指标。

(b)对于 GPU,大幅增加粒子数能加快算法达到所设精度;对于 CPU,过多增加粒子数会减慢算法达到所设精度。大幅增加粒子数是适应 GPU 计算架构的特殊方法,但不适合 CPU 计算架构。

(c)对于 CPU 上适合 100～200 粒子数解决的具有较大计算复杂度、需要较高求解精度的函数优化问题,大幅增加粒子数会显著“恶化”CPU 程序性能,前述“等粒子数等迭代次数的加速比”和“等粒子数等精度的加速比”在相当程度上是依靠“恶化 CPU 程序性能”而获得的,是虚假的加速比;将最优 CPU 程序(本实验中指 128 个粒子对应的 CPU 程序)与 GPU 程序作比较而得到的“有效加速比”,才是真正意义上的加速比。

(d)Sphere、Rosenbrock、Rastrigrin、Griewangk 这 4 个函数最高分别获得了 6.7、11.9、12.5 和 13.2 的“有效加速比”。

(e)GPU 上大幅增加粒子数能获得“有效加速比”的本质是:当粒子数小于等于 16384(实验所用 GPU 的最大驻留线程数为 26624)时,达到同样的目标精度,减少迭代次数所节约的时间,远远大于更多线程同步(包括 kernel 函数结束对应的所有线程的同步,以及_syncthreads()调用对应的线程块内

线程的同步)所浪费的时间;当粒子数大于等于 32768 时,达到同样的目标精度,减少迭代次数所节约的时间,可能大于以下三者共同作用下浪费的时间——更多线程同步所浪费的时间、更多线程处于"不活动"状态所浪费的时间,以及线程切换能掩盖存储器访问延迟所节约的时间。

另外,上文中已经提到,GPU 中三角函数使用的是软件库中的 cosf()函数,而不是更快速但降低精度的硬件函数_cosf()。为了更好地说明该问题,将 Rastrigrin 的 GPU 函数中分别使用_cosf()和 cosf()进行了计算,对比情况如表 10 所列。

表 10 Rastrigrin 的 GPU 函数使用硬件函数和软件库函数对比

| 粒子数 | 迭代次数 | 硬件函数_cosf() | | | 软件库函数 cosf() | | |
|---|---|---|---|---|---|---|---|
| | | 达到精度要求 | | 达到迭代次数要求 | 达到精度要求 | | 达到迭代次数要求 |
| | | 迭代次数 | 运行时间/ms | 运行时间/ms | 迭代次数 | 运行时间/ms | 运行时间/ms |
| 128 | 40000 | 19696 | 2253.2 | 4575.9 | 12134 | 1869.7 | 6163.3 |
| 256 | 20000 | 9845 | 1157.8 | 2352.0 | 6591 | 1036.9 | 3146.0 |
| 512 | 10000 | 5253 | 627.8 | 1195.0 | 3661 | 587.7 | 1605.2 |
| 1024 | 5000 | 3046 | 372.1 | 610.9 | 2039 | 333.8 | 818.4 |
| 2048 | 4000 | 2476 | 308.5 | 498.3 | 1653 | 286.8 | 694.0 |
| 4096 | 3000 | 1888 | 241.9 | 384.3 | 1289 | 237.6 | 552.7 |
| 8192 | 2000 | 1370 | 195.6 | 285.4 | 909 | 194.6 | 427.8 |
| 16384 | 1000 | 759 | 124.5 | 164.1 | 523 | 124.2 | 237.3 |
| 32768 | 1000 | 762 | 181.3 | 237.7 | 304 | 117.8 | 387.1 |
| 65536 | 1000 | 717 | 239.6 | 334.0 | 297 | 172.2 | 578.4 |
| 131072 | 800 | 590 | 323.0 | 437.8 | 247 | 245.5 | 396.9 |

分析表 10 数据可知,相比使用软件库函数,使用硬件函数能更快完成指定的迭代次数,但完成指定的精度反而会变慢(至少本实验中如此)。考虑到基于精度的性能指标比基于迭代次数的性能指标更客观,GPU 函数的行为尽可能与对应的 CPU 函数相同,硬件的更新换代可能使未来的 GPU 架构更利于软件库函数的调用等因素,本实验中使用了需要消耗大量时钟周期但精度较高(与 CPU 函数的行为类似)的软件库函数。

**结束语** 本文用"有效加速比"代替传统的加速比指标来衡量 GPU 加速 PSO 算法的性能,将 GPU 并行算法的性能与最优 CPU 串行算法的作比较,以能否加速收敛到目标精度为判断依据,克服了以恶化 CPU 算法性能来提升 GPU 算法加速比的缺点。在 CUDA 架构下对多个基准测试函数的测试结果表明,大幅增加粒子数是适合 CUDA 并行架构的有效方法,能加速 PSO 算法收敛到目标精度,并获得了 10 倍以上的"有效加速比"。

## 参 考 文 献


[1] Kennedy J, Eberhart R. Particle swarm optimization [C]// Proceedings of the IEEE International Conference on Neural Networks. Perth, WA, 1995, 4: 1942-1948

[2] Poli R, Kennedy J, Blackwell T. Particle swarm optimization: an overview [J]. Swarm Intelligence, 2007, 1(1): 33-57

[3] Singhal G, Jain A, Patnaik A. Parallelization of particle swarm optimization using message passing interfaces (MPIs) [C]// IEEE World Congress on Nature & Biologically Inspired Computing. Coimbatore, 2009: 67-71

[4] Deep K, Sharma S, Pant M. Modified parallel particle swarm optimization for global optimization using Message Passing Interface [C]// 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications. Changsha, 2010: 1451-1458

[5] Wang D Z, Wu C H, et al. Parallel multi-population Particle Swarm Optimization Algorithm for the Uncapacitated Facility Location problem using OpenMP [C]// IEEE Congress on Evolutionary Computation. HK, 2008: 1214-1218

[6] Venayagamoorthy G K, Gudise V G. Swarm intelligence for digital circuits implementation on field programmable gate arrays platforms [C] //Proceedings of the IEEE Conference on Evolvable Hardware. 2004: 83-86

[7] Maeda Y, Matsushita N. Simultaneous Perturbation Particle Swarm Optimization Using FPGA [C] // IEEE International Joint Conference on Neural Networks. Orlando, FL, 2007: 2695-2700

[8] Veronese L, Krohling R. Swarm's flight: Accelerating the particles using C-CUDA [C]// Proceedings of the IEEE Congress on Evolutionary Computation. Trondheim, 2009: 3264-3270

[9] Calazan R M, Nedjah N, de Macedo Mourelle L. Parallel GPU-based implementation of high dimension Particle Swarm Optimizations [C]// 2013 IEEE Fourth Latin American Symposium on Circuits and Systems. Cusco, 2013: 1-4

[10] Zhou Y, Tan Y. GPU-based parallel particle swarm optimization [C]// Proceedings of the IEEE Congress on Evolutionary Computation. Trondheim, 2009: 1493-1500

[11] Venneschi L, Codecasa D, Mauri G. An empirical comparison of parallel and distributed particle swarm optimization methods [C]// Proceedings of the 12th annual conference on Genetic and evolutionary computation. Portland, Oregon, 2010: 15-22

[12] Solomon S, Thulasiraman P, Thulasiram R. Collaborative multi-swarm PSO for task matching using graphics processing units [C]// Proceedings of the 13th annual conference on Genetic and evolutionary computation. Dublin, Ireland, 2011: 12-16

[13] Mussi L, Daolio F, Cagnoni S. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture [J]. Information Sciences, 2010, 181(20): 4642-4657

[14] Mussi L, Nashed Y S G, Cagnoni S. GPU-based asynchronous particle swarm optimization [C]// Proceedings of the 13th annual conference on Genetic and evolutionary computation. Dublin, Ireland, 2011: 1555-1562

[15] 张庆科,杨波,王琳,等. 基于 GPU 的现代并行优化算法 [J]. 计算机科学,2012,39(4):304-311

[16] 蔡勇,李光耀,王琥. 基于 CUDA 的并行粒子群优化算法的设计与实现 [J]. 计算机应用研究,2013,30(8):2415-2418

[17] Roberge V, Tarbouchi M. Efficient parallel Particle Swarm Optimizers on GPU for real-time harmonic minimization in multilevel inverters [C]// 38th Annual Conference on IEEE Industrial Electronics Society. Montreal, 2012: 2275-2282

[18] Bastos-Filho C, Oliveira M, Nascimento D, et al. Impact of the Random Number generator quality on particle swarm optimization algorithm running on graphic processor units [C]// IEEE 10th International Conference on Hybrid Intelligent Systems. Atlanta, 2010: 85-90

[19] Shi Y, Eberhart R. A Modified Particle Swarm Optimizer [C]// Proceedings of the IEEE International Conference on Evolutionary Computation. Anchorage, AK, 1998: 69-73