GPU 加速的差分进化粒子滤波算法*

曹 洁[a,b], 黄开杰[b], 王进花[b]
(兰州理工大学 a. 计算机与通信学院; b. 电气工程与信息工程学院, 兰州 730050)

摘 要: 为了解决实时系统中粒子滤波的计算复杂性问题,提出了一种零 bank 冲突并行规约的差分进化粒子滤波方法。该方法首先分析了并行差分进化粒子滤波算法在 GPU 中的内存访问模式,根据粒子滤波器的均方根误差与内存访问 bank(存储体)冲突度成正比的关系,提出了一种去除 bank 冲突的有填充寻址的差分进化粒子滤波算法,降低了计算复杂度。将该算法在 NVIDIA GTX960 GPU 中实现,与串行差分进化粒子滤波算法进行比较。实验表明,随着粒子数增加,计算量以指数增加,采用 GPU 加速的跟踪算法的执行时间明显减少,有效提高了跟踪精度,降低了计算时间。

关键词: GPU; 粒子滤波; 差分进化; 并行规约; 零内存访问冲突
中图分类号: TP301.6 **文献标志码:** A **文章编号:** 1001-3695(2018)07-1965-05
doi:10.3969/j.issn.1001-3695.2018.07.009

Speeding up differential evolution particle filter algorithm by GPU

Cao Jie[a,b], Huang Kaijie[b], Wang Jinhua[b]
(*a. College of Computer & Communication, b. College of Electrical & Information Engineering, Lanzhou University of Technology, Lanzhou* 730050, *China*)

Abstract: This paper proposed a differential evolution particle filter method based on parallel protocol of zero bank conflict, in order to solve the computational complexity of particle filtering in real-time systems. Firstly it analysed the memory access mode of parallel differential evolution particle filter algorithm in GPU. Then according to the relationship that the root mean square error was proportional to the degree of bank conflicts in memory access, it proposed a differential evolution particle filter with filled addressing mode to remove bank conflict, which reduced the computational complexity. It implemented the algorithm in NVIDIA GTX960 GPU, and compared with the serial differential evolution particle filter algorithm. As the number of particles increases, the amount of calculation increases exponentially. Theoretical analysis and simulation results show that the tracking algorithm using GPU acceleration is significantly reduced the execution time. It improves the tracking accuracy and reduces the computation effectively.

Key words: GPU; particle filter; differential evolution(DE); parallel protocol; bank conflicts free

0 引言

粒子滤波是一种采用粒子来近似后验概率密度分布的序贯蒙特卡洛方法。作为一种可处理非线性、非高斯状态估计问题的有效方法,粒子滤波被广泛应用于信号处理[1]、目标跟踪、机器人定位和图像处理等领域。目前粒子滤波算法研究中主要存在粒子退化和实时性问题。针对粒子退化和滤波精度下降问题,研究人员提出了粒子滤波的多种改进方法,其中将多种寻优方法引入重采样过程,以快速地提取到反映系统概率特征的典型粒子。文献[2,3]将遗传算法与粒子滤波相结合,有效抑制了粒子贫化的现象。文献[4]将多智能体协同进化机制引入粒子滤波,通过粒子间的竞争、交叉、变异以及自学习等进化行为来实现重采样过程,有效解决了粒子退化和粒子匮乏问题。文献[5]则对比了粒子滤波在不同搜索策略下的滤波精度,差分进化粒子滤波算法的精度提高了,但增加了计算复杂度。针对计算复杂度问题,文献[6~8]提出基于 GPU 的粒子滤波并行算法,将传统粒子滤波算法与 GPU 有效结合起来,充分利用 GPU 并行运算的性能,加快粒子滤波算法的计算速度。文献[9,10]提出对粒子滤波进行基于 GPU 的并行优化设计和实现,从而提升跟踪算法的计算速度。文献[11~13]基于 CUDA 的并行粒子群优化算法的设计与实现,用大量的 GPU 线程来加速整个粒子群的收敛速度。上述文献的并行粒子滤波算法中均用并行规约算法来简化线程操作。并行规约算法是一种目前常用的并行算法。作为一种最佳的顺序算法,相对串行算法,它能执行较多的操作,提高执行效率。这种算法工作效率较高,但由于规约算法[14,15]内存访问模式,易产生共享内存访问冲突现象,不能有效利用 NVIDIA GPU 的硬件资源。规约求和中包含大量重复的操作,简单但低效。分块规约求和[14]把问题分解成多个子问题,减少线程操作复杂度,一定程度上解决了重复操作问题,提高了执行效率,但仍存在重复的操作。并行前缀求和避免了线程重复,但存在严重的内存访问 bank conflicts 问题,使得 GPU 硬件资源的利用率较低。

针对共享内存访问时的 bank conflicts,本文提出了 free bank conflicts 并行规约算法,采用带填充寻址的方式,在每个共享内存数组索引中添加一个可变数量的填充。线程执行重采样步骤中,对 N 个粒子进行重采样时,不可避免地会产生粒子间的数据关联,使得并行难以实现。本文采用改进前缀和的方式来减少粒子滤波器中粒子间的依赖性,解决难以并行的问题。

1 差分进化粒子滤波算法

1.1 粒子滤波

假设状态方程和观测方程如下:

$$\begin{aligned} x_k &= f(x_{k-1},k) + v_{k-1} \\ y_k &= g(x_k,k) + u_k \end{aligned} \tag{1}$$

收稿日期: 2017-03-03; **修回日期:** 2017-04-10 **基金项目:** 国家自然科学基金资助项目(61633031);甘肃省自然科学基金资助项目(1506RJZA105)

作者简介: 曹洁(1966-),女,安徽灵璧人,教授,博导,主要研究方向为信息融合理论与应用、智能交通、信号检测与估计;黄开杰(1989-),男,湖北大冶人,主要研究方向为智能信息处理(18893471259@163.com);王进花(1978-),女,甘肃天水人,副教授,硕士,主要研究方向为检测技术与自动化装置.

其中：x_k 为 k 时刻的状态变量；y_k 为 k 时刻的观测值；f 和 g 为非线性函数；v_k 和 u_k 分别为系统噪声和观测噪声，具有方差 Q_k 和 R_k。在 k 时刻，粒子滤波首先通过预测采样获得新的粒子集，即

$$P_k = \{(x_k^i, \omega_k^i) \mid i = 1, 2, \cdots, N\} \tag{2}$$

并利用式(3)近似该时刻的后验概率密度。

$$p(x_k \mid y_{1:k}) = \sum_{i=1}^{N} \omega_k^i \delta(x_k - x_k^i) \tag{3}$$

其中：N 表示 k 时刻的粒子数目；$\delta(\cdot)$ 为狄利克雷函数。依据序贯重要性采样原理，重要性权值的更新公式可表示为

$$w_k^i \propto w_{k-1}^i \frac{p(y_k \mid x_k^i) p(x_k^i \mid x_{k-1}^i)}{q(x_k^i \mid x_{k-1}^i, y_k)} \tag{4}$$

其中：$q(x_k^i \mid x_{k-1}^i, y_k)$ 为建议分布函数。当采样的粒子数目足够多时，由重要性采样定理即可保证粒子集 $\{x_k^i, w_k^i\}_{i=1}^N$ 所描述的分布逼近真实的后验概率密度。

1.2 差分进化算法

差分进化(differential evolution，DE)算法是由 Storn 等人提出的一种用于连续空间全局优化的启发式算法。其基本思想是从原始种群开始，通过变异、杂交、选择操作来衍生出新的种群，不断地进行迭代，并根据每个个体的适应度值，保留优良个体，淘汰劣质个体，引导搜索过程向全局最优解逼近，以实现全局最优解的搜索。算法具有结构简单、易于实现、无须梯度信息、参数较少等优点。差分进化粒子滤波(differential evolution particle filters，DE-PF)的计算过程如下：

a)初始化。产生 N 个初始粒子 $x_0^1, x_0^2, \cdots, x_0^N$ 从预先定义的初始状态分布 $p(x_0)$。所有的粒子具有相同的初始权重，$w_0^i = 1/N$。重复迭代 for $T = 1, 2, 3, \cdots, N$。

b)预测。通过状态转移模型绘制预测粒子 x_T^i($i = 1, 2, 3, \cdots, N$)。x_T^i是相互独立的。这些预测的粒子可以用来近似地先验预测分布 $p(x \mid y_{1:T-1})$。

c)权重更新及归一化。接收测量后，每个粒子需要根据似然函数 $p(y_T \mid x_T^i)$ 更新权重。

$$w_t^i = w_{t-1}^i \times p(y_t \mid x_t^i) \tag{5}$$

归一化过程使得粒子权重的总和等于 1。归一化过程表示为

$$w_t^i = w_t^i / \sum_N w_t^i \tag{6}$$

d)差分进化重采样。

(a)设置迭代次数 $g = 1$，初始化粒子 $\{x_k^{g,i}\}_{i=1}^N = \{x_k^i\}_{i=1}^N$。

(b)对粒子集 $\{x_k^{g,i}\}_{i=1}^N$ 进行变异操作，然后进行交叉操作，得到候选粒子集 $\{\tilde{x}_k^{g,i}\}_{i=1}^N$。

(c)计算候选粒子集 $\{\tilde{x}_k^{g,i}\}_{i=1}^N$ 的适应度值，并进行选择操作，所得粒子集为 $\{x_k^{g+1,i}\}_{i=1}^N$。

(d)若 $g < G_{\max}$ 且 $\sigma > \sigma_{\min}$，σ 为融合系数，则设定 $g = g + 1$ 转入步骤 b)；否则转入下一步。

e)状态输出，计算状态估计 $x_k = \sum_{i=1}^{N} w_k^i x_k^i$。

2 基于 GPU 改进的并行规约

2.1 GPU 架构

计算统一设备架构(CUDA)是由 NVIDIA 开发的并行计算架构。图 1 是 GPU 计算单元图，表示 GPU 计算单元中的一个流处理器。其中 SM 代表流多处理器(stream multiprocessor)。每个 SM 中包含多个标量流处理器 SP(stream processor)。一个 SM 存储器层次结构分为四种：a)每个 SP 有一组寄存器；b)每个 SP 中都有 16 个存储体(bank)的共享内存(shared memory)；c)纹理内存(texture memory)和常量内存(constant memory)，都是只读内存；d)可被所有流处理器写和读的设备端全局内存(global memory)。由于 GPU 硬件的性质，四种类型存储器的访问时间有很大差异，全局内存比其他类型的内存慢了大约 100 倍，因此是计算性能提升的瓶颈。所以可以通过优化数据的局部性或减少全局存储器访问来解决实时性差的问题。在 CUDA 并行程序中，线程是执行的基本单元。一定数量执行相同指令的线程组成一个 block。在这些 block 中，线程之间可以通过共享内存进行数据通信和共享，并实现同步。所有完成相同功能的 block 组成一个 grid，grid 间的执行是串行的。当程序加载时，grid 加载到整个设备上，所有 block 被串行地分配到每一个流多处理器上，而线程则对应着每一个流单处理器核心。因此，在 CUDA 程序中，首先必须合理地对 GPU 程序进行执行配置，即确定 block 数和每个 block 中的线程数以及共享内存的大小，这样才能更好地提高并行度，充分利用系统资源。

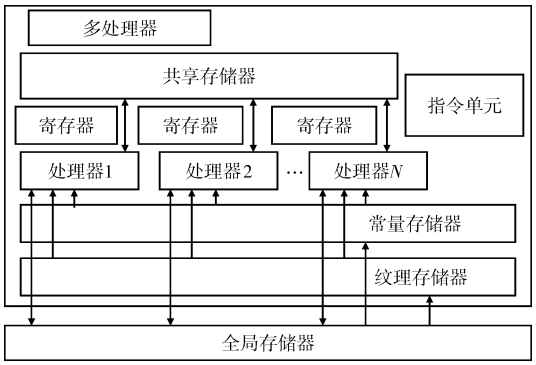


图 1 计算单元简略图

2.2 GPU 上的并行规约

重采样需要计算粒子的累积分布函数(cumulative distribution function，CDF)，是一个简单连续前缀和操作，描述如下：

$$y[n] = y[n-1] + x[n] \tag{7}$$

其中：$n = 0, 1, \cdots, N-1$；$y[-1] = 0$；N 是数据的大小。$y[n]$ 的顺序计算非常直接。由于输出数据之间的依赖性，使得并行化困难。对于小型的前缀和问题，只使用一个线程块，采用递归倍增的方法来解决。然而并行粒子滤波需要较长前缀和问题的计算。

基本规约：规约过程可以转换为两个矩阵的乘积，该点积运算通过调用 CUDA 库函数获得，然而包含大量重复操作。该方法简单，但不高效。

分块规约：把该规约求和问题分解成很多子问题，这些子问题并行执行。效率有提高，但仍然存在重复操作。

为了解决基本规约和分块规约中存在重复线程操作的问题，使用并行前缀和。

并行前缀求和：长前缀和问题需要使用多个线程块计算。当输入数据 1 ~ 16 时，每一步骤的计算操作如图 2 所示。

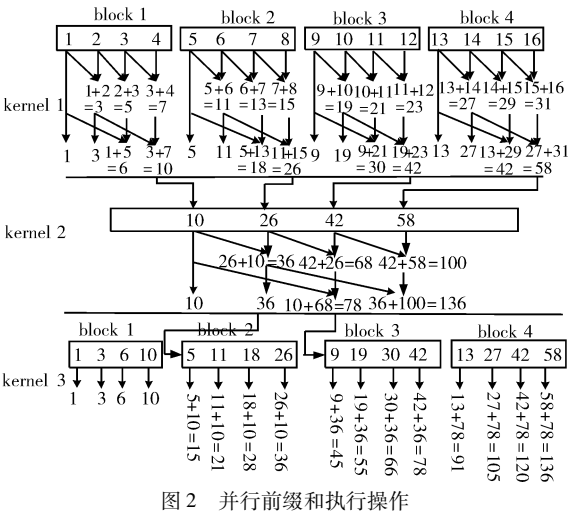


图 2 并行前缀和执行操作

输入数据平均划分并分发给每个线程块。步骤如下：a)每个线程块进行局部递归倍增，并产生局部前缀和值；b)所有线程块的最后一个输出值，作为下一个线程块的初始条件；c)通过组合初始条件和输入数据，得到累积和。每步对应一个 GPU 内

核,通过全局内存访问实现各步骤间的全局通信。并行执行粒子滤波中,步骤 a) ~c)被分发到粒子滤波的更新粒子、计算权重和差分进化重采样三个阶段中。并行前缀和操作表示为

$$\sum_{i=1}^{b}\left\lceil\frac{\frac{N}{B^{i-1}}}{P_{\max}}\right\rceil\lg B+\lg C+\left\lceil\frac{N-B}{P_{\max}}\right\rceil \tag{8}$$

令粒子数为 $N=B^bC$,其中 $C<B$;其中:⌈ ⌉表示单元操作;$P_{\max}$表示由硬件提供的并行线程的最大数目。$N=16$,$B=4$,$C=1$,$P_{\max}\ll N$。

采用并行前缀和方式,消除了线程重复操作,执行更少的操作,提高了计算效率。但在内存访问时容易产生 bank conflicts,不能有效利用 NVIDIA GPU 的硬件资源,计算性能差。为了解决共享内存访问冲突这一问题,本文提出内存访问零冲突的并行前缀求和。在 CUDA 编程指南中,多个存储体(banks)组成共享内存。当 Warp 中多个线程访问同一个 bank 时,这就发生 bank conflicts。访问同一个 bank 的线程数称为 bank 冲突度。Bank conflicts 导致序列化的多次访问内存,存在 n 度 bank conflicts 共享内存访问需要 n 次周期处理才没有访问冲突。在半个 Warp 上,执行 16 个线程,最多产生 16 度 bank conflicts。在大多数 CUDA 中,如果改进共享内存数组的访问,就能避免 bank conflicts。在每个共享内存数组索引中添加一个可变数量的填充。即给索引添加一个值,该值由索引值除以共享内存 bank 数量所得到。通过修改规约算法,简化线程操作,定义无冲突偏移宏 conflict_free_offset。

原代码和改进后代码如下所示:

```
#define NUM_BANKS  16
#define LOG_NUM_BANKS  4
#if define ZERO_BANK_CONFLICTS
#define CONFLICT_FREE_OFFSET(n) \ ((n) > > NUM_BANKS + (n) > > (2 * LOG_NUM_BANKS))
#else
#define CONFLICT_FREE_OFFSET(n) ((n) > > LOG_NUM_BANKS)
#end if
```

线程块代码

```
block A: temp[2 × thid] = g_idata[2 × thid]; //load input into shared memory
         temp[2 × thid + 1] = g_idata[2 × thid + 1];
block B: int ai = offset × (2 × thid + 1) - 1; int bi = offset × (2 × thid + 2) - 1;
block C: if (thid = = 0) {temp[n - 1] = 0;} //clear the last element
block D: int ai = offset × (2 × thid + 1) - 1; int bi = offset × (2 × thid + 2) - 1;
block E: g_odata[2 × thid] = temp[2 × thid]; //write results to device memory
         g_odata[2 × thid + 1] = temp[2 × thid + 1];
```

改进后线程块代码

```
block A: int ai = thid; int;int bi = thid + (n/2); int bankOffsetA = CONFLICT_FREE_OFFSET(ai); int bankOffsetB = CONFLICT_FREE_OFFSET(ai);
temp[ai + bankOffsetA] = g_idata[ai]; temp[bi + bankOffsetB] = g_idata[bi];
block B and D:int ai = offset × (2 × thid + 1) - 1; int bi = offset × (2 × thid + 2) - 1; ai + = CONFLICT_FREE_OFFSET(ai); bi + = CONFLICT_FREE_OFFSET(bi);
block C: if (thid = =0) {temp[n - 1 + CONFLICT_FREE_OFFSET(n - 1)] = 0;}
block E: g_odata[ai] = temp[ai + bankOffsetA]; g_odata[bi] = temp[bi + bankOffsetB];
```

图 3 为无填充寻址。int ai = offset × (2 × thid + 1) - 1;int bi = offset × (2 × thid + 1) - 1;temp[bi] + = temp[ai]。其中:ai 表示访问的线程;bi 表示被 ai 线程访问的 bank 所对应的字地址;thid 表示线程在内存数组中的索引,即线程索引。图 3(a)中,偏移量 =1,寻址(ai),步长 =2,由图可知,两个线程访问同一个 bank,产生 2 度 bank 访问冲突;图 3(b)中,偏移量 =2,寻址(ai),步长 =4,由图可知,四个线程访问同一个 bank,产生 4 度 bank 访问冲突。

图 4 为有填充寻址。int ai = offset × (2 × thid + 1) - 1;int bi = offset × (2 × thid + 1) - 1;ai + = ai/NUM_BANKS;bi + = bi/NUM_BANKS;temp[bi] + = temp[ai]。图 4(a)中,偏移量 =1,每隔 16 个元素填充 1,由图可知,一对一访问,bank 访问冲突为零;图 4(b)中,偏移量 =2,按每隔 16 位元素填充,填充量为由索引值除以共享内存 bank 数量所得到,由图可知,一对一访问,访问冲突为零。有填充寻址的规约方式有效解决了内存访问 bank 冲突这一问题。

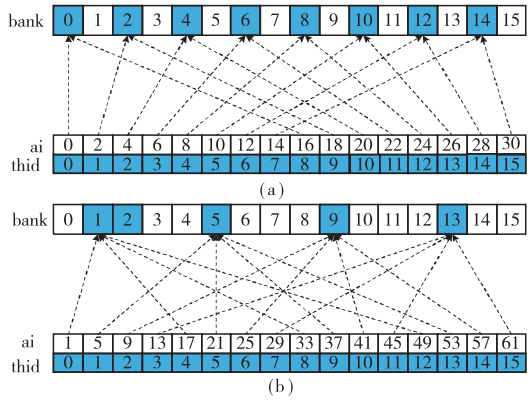


图 3　无填充寻址

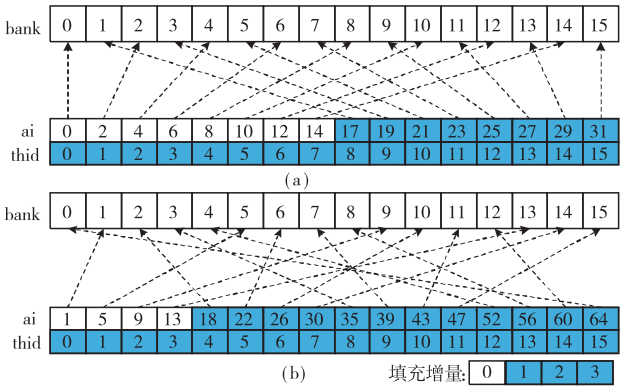


图 4　有填充寻址

3　改进规约的差分进化粒子滤波算法在 GPU 上的并行实现

3.1　改进的粒子滤波算法在 GPU 上的实现

基于 GPU 的粒子滤波器根据每个时刻的观测值估计当前状态的状态值。如图 5 所示,粒子滤波器根据每个时刻的观测值估计当前状态的状态值。并行实现过程分为以下三个阶段:

a)第一阶段。首先粒子选择,通过上一时刻产生的复制索引数组,每个线程在全局存储器中读取在上一个时刻的粒子;接着采样和权重更新,再进行差分进化重采样,即对上面获得的粒子进行变异、交叉、选择操作;然后进行局部加权值求和,最后进行局部估计,其中每个线程块独立运行。

b)第二阶段。以并行前缀和计算为开始,进行全局加权求和;再进行全局估计。

c)第三阶段。首先计算累计分布函数;再进行复制索引的生成,将产生在下一个时刻中需要的复制索引数组。详细过程在 3.2 节中介绍。

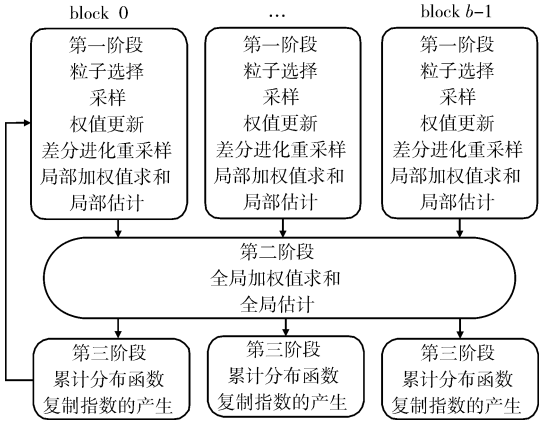


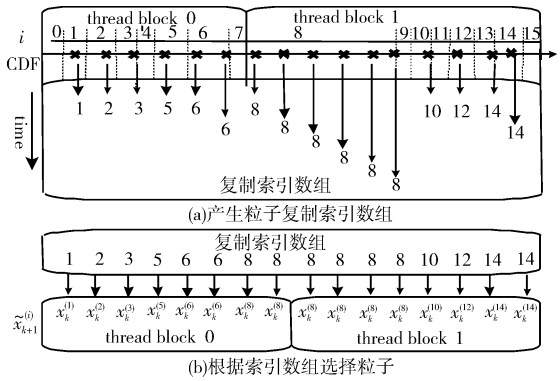
图 5　基于 GPU 的并行粒子滤波线程块

线程块由多个线程组成,给每个线程分配一个粒子,粒子总数 M 等于总线程数 LB。本文中第一、三阶段采用 B 个线程

块,每个块有 L 个线程组成;第二阶段使用由 LB 个线程组成的一个线程块。

3.2 粒子复制

图 6(a)表示粒子复制过程。图中有两个线程块被绘制,每个线程块有八个线程。



(a)产生粒子复制索引数组

(b)根据索引数组选择粒子

图 6 复制索引数组示意图

在复制索引生成步骤中,线程同时更新存储在全局内存中的复制索引数组。在粒子选择步骤中使用复制索引数组,从上一时刻粒子集中选择新的粒子。即在 K 时刻结束时,每一个线程需要确定它的目标索引;在 $K+1$ 时刻,它的粒子被复制,即粒子 $x_k^{(i)}$ 复制到 $\{\tilde{x}_{k+1}^{j}\}_{j\in\Omega_k^{(i)}}$,$\Omega_k^{(i)}$ 是是一组粒子 $x_k^{(i)}$ 被复制到线程的索引集合。然后带有全局索引 j 的线程从 $x_k^{(\rho_k[j])}$ 中选择粒子 $\tilde{x}_{k+1}^{(j)}$。粒子复制指数数组 $\rho_k[j]$ 定义为

$$j\in\Omega_k^{(i)},\rho_k[j]=i \tag{9}$$

$\{\Omega_k^{(i)}\mid\Omega_k^{(i)}\neq\varnothing\}_{i=0}^{M-1}$ 是粒子集的一部分,定义 i 是 0 ~ $M-1$ 的整数,$\rho_k[j]$ 为粒子集合的复制索引数组。每个线程从上一时刻的粒子组中选择一个粒子。这意味着多个线程可以复制相同的粒子,而一些粒子被丢弃。在第三阶段的复制索引生成步骤中,$\rho_k[j]$ 生成并存储在全局内存中。根据重采样,本文设置 $W_k[-1]=0$,其中 u_k 是在半开区间 (0,1] 中产生的一个统一的随机数,并且定义

$$s_{lo,k}[i]=\lfloor MW_k[i-1]+1-u_k\rfloor \tag{10}$$

$$s_{hi,k}[i]=\lfloor MW_k[i]-u_k\rfloor \tag{11}$$

$$\Omega_k^{(i)}=\{j\mid W_k[i-1]<\frac{j+u_k}{M}\leqslant W_k[i],j\in Z\}=$$
$$\{j\mid s_{lo,k}[i]\leqslant j\leqslant s_{hi,k}[i],j\in Z\} \tag{12}$$

在 $k+1$ 时刻,每个线程读取上一时刻储存在全局存储器中的一个粒子。例如线程块 b 中的线程 l 选择新粒子为

$$\tilde{x}_{k+1}^{(Lb+l)}=x_k^{(\rho_k[Lb+l])} \tag{13}$$

每个线程从全局内存中读取在上一时刻生成的复制索引数组;接着线程从上一个时刻的粒子组中选择粒子,这些粒子存储在全局内存中,并将 $w_{k+1}^{(i)}$ 初始化为 $1/M$。

4 实验及性能分析

为验证 IPDE-PF 算法的精度和运算时间等基本性能,利用典型一维非线性系统模型对算法的性能进行仿真,并用 PF、DE-PF、PDE-PF、IPDE-PF 进行对比。整个实验平台包括台式机和 GTX 960。详细参数如表 1 所示。

表 1 详细参数

GPU 参数		CPU 参数	
GPU	GTX960	CPU	i5-4460
核数	1 024	核数	4/8
时钟率	1.3 GHz	时钟率	3.2 GHz
共享内存	48 KB	内存	8 GB
寄存器/SM	64 KB		
全局内存	4 GB		

一维非线性系统模型为

$$\begin{cases}x_k=1+\sin(0.04\pi k)+0.5x_{k-1}+u_{k-1}\\ y_k=\begin{cases}0.2x_k^2+v_k & 1\leqslant k\leqslant 30\\ 0.5x_k-2+v_k & 30<k\leqslant T\end{cases}\end{cases} \tag{14}$$

模型的系统噪声 $u_{k-1}\sim\Gamma(3,2)$,总观测时间 $T=50$,进化算法交叉概率 CR = 0.6,最大进化迭代次数 $G_{\max}=10$。本文使用基本粒子滤波(PF)、差分进化粒子滤波(DE-PF)、基于基本规约的差分进化粒子滤波(PDE-PF)和本文 IPDE-PF 算法做实验。

实验 1 GPU 运行时间对比分析

基于多核的性能曲线如图 7 所示。多核数目设置为 4 和 8,粒子数为 1 000 ~ 10 000。采用基于 CUDA 的单指令多线程编程模型,粒子滤波的采样、权值计算、差分进化重采样被分配到各个计算核心。采样、权值计算、差分进化重采样的计算在每个线程和每个核心执行。

随着粒子数增加,粒子滤波计算时间也随之增加。如图 7 所示,当粒子数为 3 000、多核数为 4 时,用 0.1 s 来完成粒子滤波的计算;当粒子数为 10 000 时,本文算法仅用 0.2 s 来完成实时跟踪。根据图 7 的结果,在单指令多数据编程模型和 GPU 架构下,粒子数为 5 000 或更多时,本文算法可得到较好的计算性能。IPDE-PF 计算时间分析如表 2 所示。

表 2 IPDE-PF 计算时间分析 /s

算法		粒子数				
		1 k	2 k	4 k	8 k	10 k
串行		0.022	0.084	0.2	0.522	1.6
并行	核数 = 4	0.01	0.05	0.12	0.18	0.2
	核数 = 8	0.01	0.03	0.05	0.09	0.16

为了使本文算法在 GPU 中的计算性能更直观地体现,实验得出不同粒子数时的计算时间。粒子数为 1 000 ~ 10 000。首先,算法相关参数从主机存储器复制到 GPU 全局内存,再分配存储结果的设备端存储器。初始化 3 个 GPU 内核计算采样、权值计算、差分进化重采样,接着启动内核函数。本文中参数 B 为 10,L 为 100。3 个内核完成后,将所得结果从设备内存复制到主机内存。如图 8 所示,由图 8 可知,粒子数为 10 000 时,计算时间为 0.082 s。该计算性能适用于大部分实时系统应用。

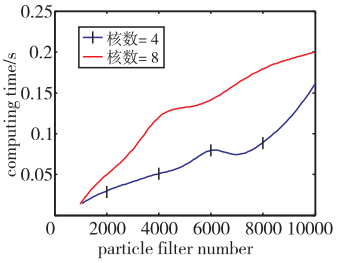


图 7 4 核和 8 核算法计算时间

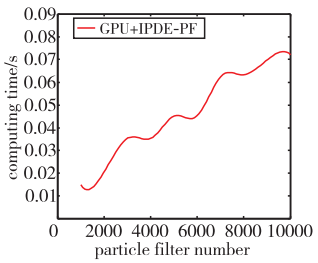


图 8 改进规约的差分进化粒子滤波算法计算时间

使用 DE-PF、PDE-PF 和本文算法在 8 核 CPU 和 GTX 960 GPU 上执行,得出加速比曲线如图 9 所示。计算时间以串行粒子滤波算法执行时间为基准线。开始曲线增势不明显,是因粒子数少、计算量小,不足以完全利用 GPU 硬件的计算性能,以至于增长率较缓慢;随着粒子数的增长、计算量的增长,充分利用了 GPU 硬件资源。另外本文带填充寻址的并行前缀和方式解决了在共享内存访问时存在的 bank 冲突问题,提高了 GPU 硬件资源利用率,有效提高了 GPU 执行率。

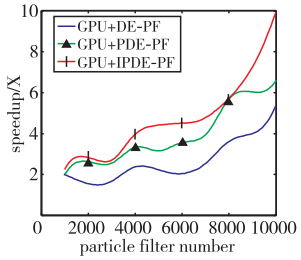


图 9 三种算法加速比

由图 9 可知,PDE-PF 算法加速比大于 DE-PF 算法的加速比,本文算法的加速比大于 PDE-PF 算法的加速比,改进规约

的差分进化粒子滤波的实时性能最佳。改进算法在 8 核下的加速比如表 3 所示。

表 3　改进算法在 8 核下的加速比

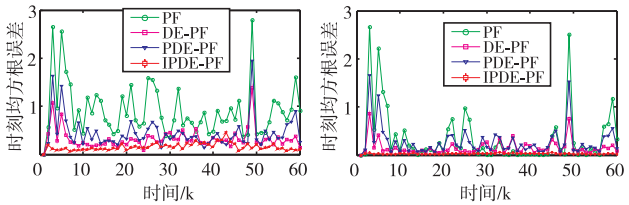
粒子数	运行时间/s		加速比
	CPU	GPU	
1 000	0.022	0.01	2.2
2 000	0.084	0.03	2.8
4 000	0.2	0.05	4
5 000	0.308	0.07	4.4
6 000	0.36	0.08	4.5
8 000	0.522	0.09	5.8
10 000	1.6	0.16	10

实验 2　均方根误差的比较

对算法进行 $R_{MC}=200$ 次独立蒙特卡洛仿真,并定义时刻 k 的均方根误差为

$$L_k^{\mathrm{RMSE}}=\sqrt{\frac{1}{R_{MC}}\sum_{j=1}^{R_{MC}}(x_{k,j}-\bar{x}_{k,j})^2} \tag{15}$$

其中:$x_{k,j}$和 $\bar{x}_{k,j}$分别表示第 j 次仿真中 k 时刻的实际状态和预测状态。量测噪声 $v_k\sim N(0,0.000\ 01)$。图 10 给出粒子数 $N=100$ 和 $N=200$ 两种设置下四种算法的时刻均方根误差对比。从图 10 可看出,IPDE-PF 的时刻均方根误差最小,说明其估计精度最高,且随着粒子数的减少估计精度仍优于另外三种算法。从图可知,采用并行前缀求和优化算法的差分进化粒子滤波误差明显小于标准粒子滤波算法,每种方法优化后的粒子滤波精度存在较大差别,IPDE-PF 的滤波精度最高;PDE-PF 的滤波精度高于 DE-PF,但算法运行时间最长,由此可见共享内存访问 bank 冲突对最终的优化效果具有显著影响;本文 IPDE-PF 算法在粒子数较少量测噪声较大的情况下,跟踪精度均好于其他三种算法。



(a)粒子数为100时均方根误差对比　(b)粒子数为200时均方根误差对比

图 10　不同粒子数的均方根误差对比

5　结束语

通过分析已有智能优化粒子滤波存在的问题,本文使用零 bank 冲突规约算法优化差分进化粒子滤波,通过采用带填充寻址方式,解决共享内存中内存访问时出现的 bank 冲突问题,提高了规约算法的执行效率,利用改进的并行前缀求和算法来优化差分进化粒子滤波,大大提高了差分进化粒子滤波算法在 GPU 上的执行效率。实验结果表明,所提出的改进规约的差分进化粒子滤波算法有效提高了实时性,能适应一般系统的应用。

下一步工作中将进一步对并行差分进化粒子滤波算法进行优化,提高可并行部分所占的比例,从而进一步提高算法整体的实时性能。

参考文献:

[1] Sileshi B G, Oliver J, Toledo R, *et al*. Particle filter SLAM on FPGA: a case study on Robot@ Factory competition[C]//Proc of the 2nd Iberian Robotics Conference. Berlin: Springer International Publishing, 2016.

[2] Pardal P C P M, Kuga H K, De Moraes R V. The particle filter sample impoverishment problem in the orbit determination application [M]//Mathematical Problems in Engineering. 2015: 1-9.

[3] Malarvezhi P, Kumar R. Particle filter with novel resampling algorithm: a diversity enhanced particle filter[J]. Wireless Personal Communications, 2015, 84(4): 3171-3177.

[4] 余春超, 杨智雄, 夏宗泽, 等. 采用 GPU 并行架构的基于互信息和粒子群算法的异源图像配准[J]. 红外技术, 2016, 38(11): 938-946.

[5] Li Hongwei, Wang Jun, Su Hongtao. Improved particle filter based on differential evolution[J]. Electronics Letters, 2011, 47(19): 1078-1079.

[6] 孙伟平, 向杰, 陈加忠, 等. 基于 GPU 的粒子滤波并行算法[J]. 华中科技大学学报: 自然科学版, 2011,39(5): 63-66.

[7] 刘光敏, 陈庆奎, 赵海燕, 等. 一种 GPU 加速的粒子滤波算法[J]. 计算机科学, 2014(Z11).

[8] Murray L M, Lee A, Jacob P E. Parallel resampling in the particle filter[J]. Journal of Computational & Graphical Statistics, 2013,25(3):789-805.

[9] 赵嵩, 徐彦, 曹海旺, 等. GPU 并行实现多特征融合粒子滤波目标跟踪算法[J]. 微电子学与计算机, 2015,32(9): 153-156,160.

[10] 刘伟, 孟朝晖, 薛东伟. 基于 CUDA 与粒子滤波的多特征融合视频目标跟踪算法[J]. 计算机系统应用, 2013, 22(11): 123-128.

[11] 武勇, 王俊, 曹运合, 等. 基于二次预测的粒子滤波算法[J]. 吉林大学学报: 工学版, 2015, 45(5): 1696-1701.

[12] 张硕, 何发智, 周毅, 等. 基于自适应线程束的 GPU 并行粒子群优化算法[J]. 计算机应用, 2016, 36(12): 3274-3279.

[13] 余莹, 李肯立, 郑光勇. 一种基于 GPU 集群的深度优先并行算法设计与实现[J]. 计算机科学, 2015, 42(1): 82-85.

[14] Padua D. All prefix sums [M]// Encyclopedia of Parallel Computing. 2011.

[15] Nakano K. An optimal parallel prefix-sums algorithm on the memory machine models for GPUs[C]//Proc of the 12th International Conference on Algorithms and Architectures for Parallel Processing. 2012: 99-113

(上接第 1949 页)

[4] Chen Y T, Chen Mengchang. Using Chi-square statistics to measure similarities for text categorization[J]. Expert Systems with Applications, 2011, 38(4): 3085-3090.

[5] Liu Huawen, Sun Jigui, Liu Lei, *et al*. Feature selection with dynamic mutual information[J]. Pattern Recognition, 2009, 42(7): 1330-1339.

[6] Shang Wenqian, Huang Houkuan, Zhu Haibin, *et al*. A novel feature selection algorithm for text categorization[J]. Expert Systems with Applications, 2007, 33(1): 1-5.

[7] Ogura H, Amano H, Kondo M. Feature selection with a measure of deviations from Poisson in text categorization[J]. Expert Systems with Applications, 2009, 36(3): 6826-6832.

[8] Yang Jieming, Qu Zhaoyang, Liu Zhiying. Improved feature-selection method considering the imbalance problem in text categorization[J]. The Scientific World Journal, 2014, 2014(5):articleID 625342.

[9] 闫健卓, 李鹏英, 方丽英, 等. 基于 χ^2 统计的改进文本特征选择方法[J]. 计算机工程与设计, 2016, 37(5): 1391-1394.

[10] 任永功, 杨雪, 杨荣杰, 等. 基于信息增益特征关联树的文本特征选择算法[J]. 计算机科学, 2013, 40(10): 252-256.

[11] 尤鸣宇, 陈燕, 李国正. 不均衡问题中的特征选择新算法: Im-IG[J]. 山东大学学报: 工学版, 2010, 40(5): 123-128.

[12] 徐燕, 李锦涛, 王斌, 等. 不均衡数据集上文本分类的特征选择研究[J]. 计算机研究与发展, 2007,44(S1):58-62.

[13] 张玉芳, 王勇, 熊忠阳, 等. 不平衡数据集上的文本分类特征选择新方法[J]. 计算机应用研究, 2011, 28(12):4532-4534.

[14] Uysal A K, Gunal S. A novel probabilistic feature selection method for text classification[J]. Knowledge-Based Systems, 2012, 36(12): 226-235.

[15] Ogura H, Amano H, Kondo M. Comparison of metrics for feature selection in imbalanced text classification[J]. Expert Systems with Applications, 2011, 38(5): 4978-4989.

[16] Fausett LV. Fundamentals of neural networks: architectures, algorithms, and applications[M]. New Jersey: Prentice-Hall Inc, 1994.

[17] Pietramala A, Policicchio V L, Rullo P. Automatic filtering of valuable features for text categorization[C]// Proc of International Conference on Advanced Data Mining and Applications. Berlin: Springer, 2012: 284-295.