基于 GPU 的不确定数据流窗口连接运算*

江 虹, 钱江波, 陈叶芳†
(宁波大学 信息科学与工程学院, 浙江 宁波 315211)

摘 要: 在很多新兴应用领域、如传感器网络,实时监控系统等,产生的数据流是不断变化的、连续到达的、数据值可能不确定、且必须被快速处理。其中有些操作,如数据流的实时窗口连接运算,非常消耗时间,这对数据流处理系统的性能提出了严峻的挑战。目前,大多数算法采用软件优化来提高处理速度,但其性能提高有限。利用 GPU(图形处理器)的高并行度、多线程、高带宽的并行处理能力,设计了一种软硬件结合的方法来加速处理数据流的窗口连接操作。在 CUDA(统一计算架构)下,由 CPU 控制将内存中的数据传输至 GPU 存储器中,然后利用多线程进行并行处理。实验验证了提出的方法可以大幅度提高多数据流窗口连接的处理速度,可达到纯软件处理的 50 倍左右。

关键词: 图形处理器; 统一计算架构; 不确定数据流; 窗口连接操作
中图分类号: TP331;TP311 **文献标志码**: A **文章编号**: 1001-3695(2014)05-1428-05
doi:10.3969/j.issn.1001-3695.2014.05.034

GPU based window joins over uncertain data streams

JIANG Hong, QIAN Jiang-bo, CHEN Ye-fang†
(*College of Information Science & Engineering*, *Ningbo University*, *Ningbo Zhejiang* 315211, *China*)

Abstract: Data in some emerging applications, such as sensor networks, real-time monitoring systems, etc., are always time-varying, uncertain, and continuously arriving. These data need to be quickly processed. However, some important operations are costly, such as the real-time window joins over data streams. The requirement is a high challenge for a data streams processing system. Currently, most of the algorithms use software optimization to accelerate the processing speed, yet the results are unsatisfactory. GPU (graphic processing unit) has a high processing performance as it uses multi-thread and high-bandwidth to process data in parallel. This paper presented a co-processing method with hardware and software to accelerate the window join operation over data streams. In CUDA (compute unified device architecture) architecture, CPU (central processing unit) transfers data to memory of GPU to be processed in parallel. Experiment results show that this method can greatly improve the processing speed with about 50 times faster than software implementation.

Key words: graphic processing unit (GPU); compute unified device architecture (CUDA); uncertain data streams; window joins processing

1 概述

1.1 数据流

在传统的数据处理中,数据具有持久性和固定性查询具有短暂性。传统的数据库系统一般是客户主动提出查询,数据库将查询后的结果反馈回去。在很多新兴应用领域,如传感器网络、实时监控系统等,产生的数据是不断变化的、连续到达的、数据值可能不确定、且必须被快速处理,这样类型的数据叫做不确定数据流[1]。其处理系统,也就是不确定数据流处理系统(UDSMS)一般要求具有以下特征:a)可以及时快速处理来到的数据;b)查询处理操作是无阻塞的;c)可以长时间进行查询处理操作;d)可以处理不确定的数据流。

然而在 UDSMS 的查询处理中,当一个数据元组进入一个无限大的数据流中,那么这个元组就必须与数据流中的每个元组进行一次比较,看是否符合连接操作的条件。很显然,这样大量数据的操作是不现实的。其中的一个解决方法就是利用规定大小的滑动窗口进行大量数据上的连接操作,新来的元组插入窗口的最末端,窗口最前端元组删除。图 1 演示了当有一个新的带有概率属性“p1”的元组“b”来到在三个流上的连接操作,该连接操作是串行的。当有一个新的元组到来的时候重复相同的操作。

1.2 GPU 以及 CUDA

并行性是未来计算的发展趋势,未来微处理器的重点也将集中在增加处理器核心的数量上而不是提高单线程的处理能力。高度并行性的 GPU[2],如 NVIDIA 公司的 GeForce 8 系列被设计成为可以利用大量处理器核心进行并行数据流处理的可编程处理器。GPU 所表现出来的性能使人们相信在不久的将来,它将为未来计算系统提供无限的潜力。GPU 的架构和编程模型与普遍单芯片的 CPU 编程模型在数据的组织上是有差异的。

GPU 的架构和编程模型与普遍单芯片的 CPU 编程模型在数据的组织上是有差异的。GPU 是一种提供特殊应用的处理

收稿日期: 2013-06-26; **修回日期**: 2013-08-07 **基金项目**: 浙江省自然科学基金资助项目(LY13F020040);浙江省“信息与通信工程”重中之重学科开发基金;宁波市自然科学基金资助项目(2012A610065)

作者简介:江虹(1989-),女,江西抚州人,硕士,主要研究方向为数据库技术;钱江波(1974-),男,浙江宁波人,教授,博士,主要研究方向为移动数据库、数据挖掘、人工智能;陈叶芳(1973-),女(通信作者),讲师,硕士,主要研究方向为数据库管理、数据挖掘、人工智能(chenyefang@nbu.edu.cn).

器,具备了流处理器模型的特点,并且集成了流加速部件。图 2 是 CPU 与 GPU 结构的对比。图中 CPU 的核心是一个 control(控制单元)和四个 ALU(arithmetic logical unit,算术逻辑单元),并带有大量的数据缓存单元(cache 以及 DRAM),而 GPU 是由多个 control 且每个 control 带有四个以上的 ALU,以及很小的数据缓存单元(DRAM)构成。由于需要进行大量的高密度计算和高并行性计算,因此 GPU 将更多的晶体管用于数据处理而不是数据缓存和流控。GPU 最初只是应用于一些三维图像渲染处理[3]的,现在已经应用于普通信号处理以及计算金融和计算生物学[4],因为这些应用都需要运用数据的并行处理。

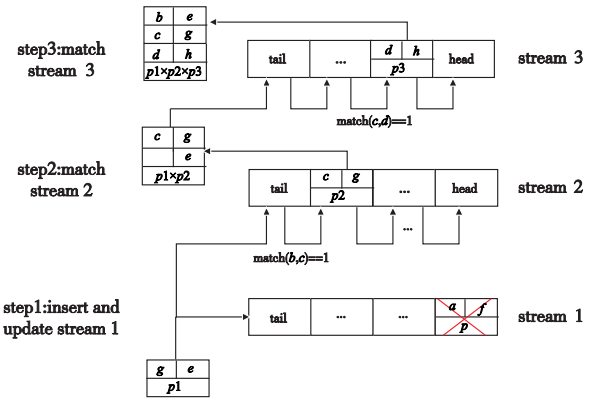

图1　基于三条数据流的窗口连接操作

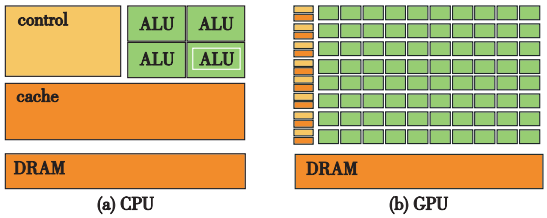

图2　CPU与GPU结构的对比

CUDA[5]是由 NVIDIA 公司推出的一种通用并行计算架构,包括了以 C 语言作为最小扩展级的并行编程模型。CUDA 利用 GPU 的多线程并行处理,可以比 CPU 更加高效地解决更多复杂计算任务。CUDA 程序编写前需要了解清楚 GPU 中的物理处理器数量,以便针对数据规模的大小进行线程结构的设计。GPU 与 CPU 协调进行并行处理的具体流程如图 3 所示。图中 1)表示内存将需要处理的数据(data)传送给到 GPU 的内存当中;2)表示内存也需要将处理数据的指令(command)传送到 GPU 的控制器中;3)表示 GPU 利用其 ALU 并行处理能力对数据进行处理;4)表示 GPU 将结果(result)返回到内存中。

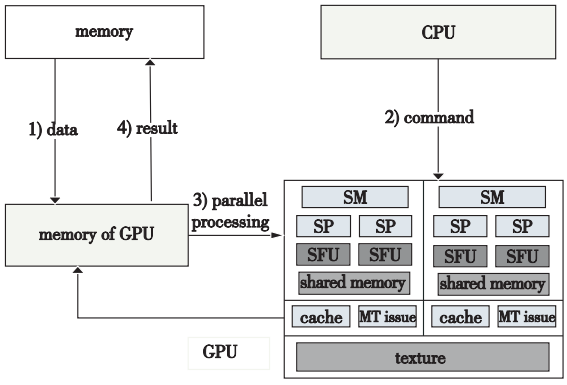

图3　GPU与CPU协调处理流程

本文的主要贡献有:

a)首先,本文尝试使用硬件与软件结合(GPU 与 CPU 结合)的方法来处理基于多条流的窗口连接操作以提供高速处理性能。GPU 与 CPU 协调处理数据具有更好的灵活性,窗口的大小可以改变,也可以执行多种类型的查询操作。

b)实验验证了本文提出的基于 CUDA 的并行数据流处理可以很好地解决多数据流软件处理速度较慢的问题,使得数据流的处理速度较使用纯软件处理更快。

在确定的数据流上的窗口连接操作是一个重要的问题,已经吸引了大批研究者的关注。Golab 等人[6]提出了一种增量式的嵌套循环连接算法(NLJs)和多方式增量哈希连接算法,并比较了它们的性能,发现 NLJs 更适用于等值连接而算法的效率主要是依赖于从滑动窗口中移出已过期元组的策略。Das 等人[7]针对数据流处理系统上的近似滑动窗口连接由于资源有限所造成的问题提出了一种模型,在该交替模型中主要是通过丢弃元组来减少资源的消耗。

随着各种领域应用的开发和应用,仅仅是确定数据流上的操作是不足以满足人们的需求的。最近在不确定数据流处理上的研究工作越来越多。不确定数据流模型最初是由 Jayram 等人[8]提出来的。它是数据流模型的一个变形,适应于处理不确定数据。在不确定数据流中,数据流中的每个元组代表了一系列可能事件的概率分布。这些文章提出了很多在不确定数据流上的聚类算法,然而很少文章有聚焦于不确定数据流上的窗口连接。PROUD[9]是用来处理不确定数据流上的相似连接查询的方法,提供了一种灵活平衡假阴性和假阳性的算法。UWJSP[10]利用 FPGA(field programmable gate array)来加速处理不确定数据流上的连接操作。而周茂春等人[11]采用了可重构的数据流 SPJ 查询处理器,可以根据输入查询的查询树调用相应的模块自适应地对 FPGA 编程,实现数据流的处理。

在数据库领域,也有一些研究者利用 GPU 来加速数据的操作处理。例如,He 等人[12]提出了一种利用 GPU 来执行关系连接操作的算法。Ligowski 等人[13]提出了针对 Smith Waterman 算法在 GPU 上的一种高效率的执行策略,比以往的基于 GPU 的执行策略性能得到了很大的提高。许多研究者[14~16]则提出了利用高效的 GPU 执行 SQL 命令处理器的选择查询操作,使得选择查询的速度得到了很大的提升。还有的研究者[17,18]直接利用 GPU 加速关系数据库中的原语操作,使得整个数据库的运行效率得到了提高。而 JO 等人[19]则在 CUDA 架构下执行数据库中的加/解密算法,速度达到了在 CPU 中执行的 8 倍。另外有些研究者注重于多数据流的研究领域,如周勇等人[20,21]采用精确方法并行计算多数据流间任意两条的相关系数,以及利用 GPU 的强大计算能力和高内存带宽的特性计算数据流分位数信息。

本文主要是针对不确定数据流窗口连接处理时间消耗大的问题,利用 GPU 加速窗口连接处理。以往的研究主要是集中在确定数据流上的连接操作,通过对算法进行改进以提高处理速度。而对于利用 GPU 来提高算法效率的连接操作也主要是集中在确定数据流上,没有考虑到不确定数据流上的窗口连接操作。

2　窗口连接算法

2.1　相关概念定义

本节主要介绍一下不确定数据流的相关概念。

定义 1　不确定元组。一个不确定元组由元组的属性和属性的概率[22]组成。

定义 2　不确定数据流。一个不确定数据流 $U\langle t,\tau\rangle$ 是无

限制的、实时的、连续的数据对$\langle t,\tau\rangle$，其中 t 是一个不确定元组，而 τ 是时间戳。因为流是有序的，这个时间戳有时候是可以省略的。

定义 3 基于数量的滑动窗口。在任何时间定义了在一个流上部分子集元组数量的整形参数 N，指的是最后的 N 个元组。一个新的元组插入窗口将会删除生存时间最长的一个元组。

定义 4 连续查询。基于流的查询在一段时间内一直在进行，当有新的数据到达的时候就会有新的查询结果产生。这是一个长时间运行的、连续的、持久的查询[23]。

定义 5 基于 M 条流的窗口连接。当新元组输入时，在 M 条流上的窗口连接是一个对称操作。对于每个属于任何一条流上的新来的元组，都要在其他的流上进行连接操作，然后输出符合条件的连接结果。

如图 1 所示，来自流 1 的元组首先在窗口 1 中进行插入并删除最老元组之后；然后输入窗口 2 进行探查操作，操作之后的结果输出；探查结果输入窗口 3 中进行探查操作，最终输出所得到的操作结果。假如最初输入的是流 2 上的元组，那么类似的操作将会重复进行一次，因此说这种操作是对称的。

定义 6 连接条件。流 R 和流 S 的连接声明可以描述成“$R.a$ ope $S.d$”，其中：“$R.a$”和“$S.d$”分别代表了流的属性，而 ope $\in\{<,>,=,\neq,\leqslant,\geqslant\}$。

2.2 具体操作过程

本节主要介绍具体的操作过程。

例 1 假设现有三条流：$R(a,b,c,p)$、$S(d,e,p)$ 和 $T(f,g,p)$。其中 a、b、c、d、e、f、g 是普通属性，而 p 是概率属性，两个元组的连接结果 p 的值等于两个元组概率之积。三条流总共有八条连接查询，这些查询有一样的或者不同的连接条件，并且有不同的窗口大小。

查询 1：select * from R, S, where $R.a=S.d$ window 500.

查询 2：select * from R, S, T, where $R.a=S.d$ and $S.e=T.f$ window 1000.

查询 3：select * from S, T, where $S.e=T.f$ window 500.

查询 4：select * from R, S, where $R.b=S.e$ window 1000.

查询 5：select * from R, S, where $R.a=S.d$ or $R.c=S.e$ window 500.

查询 6：select * from R, S, where $R.a=S.d$ and $R.c=S.e$ window 1000.

查询 7：select * from R, S, T, where $R.a=S.d$ or $S.e=T.f$ window 500.

查询 8：select * from R, S, T, where ($R.c=S.e$ or $S.e=T.f$) and $R.a=S.d$ window 2000.

为了方便操作，本文定义了四个标记量。其中 flag 是一个整型变量，作为一个标记量。当 flag 的值为 0 时，表示新输入的元组是属于流 R 的；当 flag 的值为 1 时，表示新输入的元组是属于流 S 的。当 flag 的值为 2 时，表示新输入的元组是属于流 T 的。三条流 R、S、T 被定义为二维数组，每一行代表流的一个元组，每一列代表流的一个属性。tag1、tag2、tag3 分别是标记流 R、S、T 的整型标记量，主要用来标记连接匹配是否成功。当 tag1 = 1 时，表示其他流的元组与流 R 的元组连接匹配成功；当 tag1 = 0 时，表示其他流的元组与流 R 的元组连接匹配不成功。这里讲解一个具体的例子（查询 2）。这里假设每条流的窗口包括三个元组，如图 4（a）所示。当 flag = 0 时，且流 R 的新元组（4,5,6,0.9）到达的时候，具体步骤如下：

a）新元组（4,5,6,0.9）插入到流 R 的窗口中，并且是插入窗口的最后一个位置，这代表该元组最新。由于整个窗口都是满的，插入一个新的元组以后，窗口需要删除一个最老的元组，因此窗口把第一个元组删除以防止窗口溢出。

b）新元组（4,5,6）将进入流 S 的窗口与 S 的元组进行探查，如图 4（a）所示，此时：

（a）$R.a=4$，$S.d=3$，将 tag2[0] 置为 0；

（b）$R.a=4$，$S.d=4$，将 tag2[1] 置为 1；

（c）$R.a=4$，$S.d=7$，将 tag2[2] 置为 0；

得到中间连接结果（4,5,6,8,0.9×0.8）。

c）探查流 S 窗口以后的结果为 tag2[1] = 1（此时 $S.e=8$）的结果输入流 T 的窗口中继续进行探查，如图 4（a）所示，此时：

（a）$S.e=8$，$T.f=8$，将 tag3[0] 置为 1；

（b）$S.e=8$，$T.f=9$，将 tag3[1] 置为 0；

（c）$S.e=8$，$T.f=12$，将 tag3[2] 置为 0；

这时将产生的最后结果（4,5,6,8,1,0.9×0.8×0.9）输出，如图 4（b）所示，其过程与图 4（a）是类似的，最终结果为（7,11,13,0.21）。

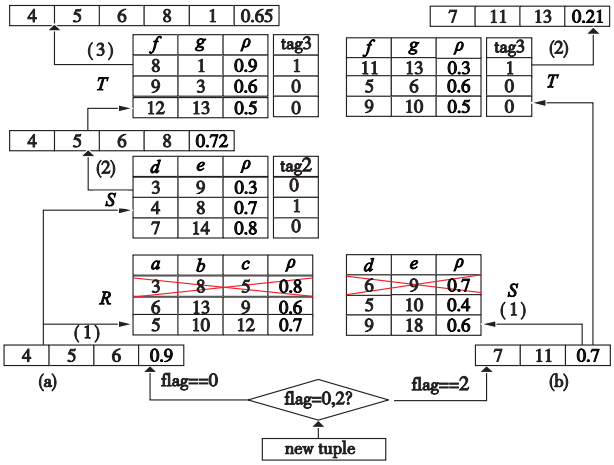

图4 具体处理步骤

3 GPU 实现

3.1 GPU 线程设计

本节先解释一下 GPU 中线程（thread）的具体设计。如图 3 所示，在数据处理之前，首先要将 CPU 中的需要进行处理的数据传送到 GPU 的共享存储器中（传送速度达到几千兆每秒）。如图 5 所示，这里查询窗口的大小是 1 000，为了方便执行，将 GPU 存储器的第一个 block（这里只需要用到一个 block，在其他的查询中需要用到两个 block）中的线程设计成 25×40，因此将流中的第一个元组放入线程（0,0）中处理，然后将第二个元组放入线程（0,1）、将最后一个元组放入线程（39,24）中进行处理，其他元组分别放入对应的线程中进行处理。所有的这些线程都是并行执行的，相互之间是无干扰的。

3.2 具体实现

在内核（kernel）程序处理元组数据之前，首先需要 CPU 程序先进行各种参数的定义和随机产生以及参数的传递。如图 6 所示，定义和随机初始化流 R、S、T，新到来的元组随机产生。查看标志 flag 的数值，如果 flag = 0，则新来的元组是属于 R，并将新元组以及 S、T 传递到 GPU 的内存中进行 2.2 节中八条查

询中 R 所涉及到的七条查询(其中第三条查询与 R 无关);如果 flag = 1,则新来的元组是属于 S,并将新元组以及 R、T 传递到 GPU 的内存中进行八条查询(所有的查询都涉及到 S);如果 flag = 2,则新来的元组是属于 T,并将新元组以及 R、S 传递到 GPU 的内存中进行四条查询(其中有四条查询没有涉及到 S)。这里对于产生的每一个新元组,八条查询中涉及该新元组的流都要依次进行操作且操作是串行的。在每条查询中,新元组与流进行比较是并行操作的。

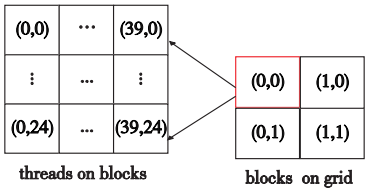

图5　线程结构设计

Algorithm:executed at CPU transmit to GPU
Input: R the stream need to be matched; S the stream need to be matched; T the stream need to be matched; flag the tag value for new tuple;
Process: Initialize R,S,T randomly, 1.New tuple arrives randomly. 2.if flag equal 0,the new tuple belong to R,aupdate R and transmit S,T,new tuple to memory of GPU. 3.if flag equal 1,the new tuple belong to S,update S and transmit R,T,new tuple to the memory of GPU. 4.if flag equal 2,the new tuple belong to T,updateT and transmit R,S,new tuple to the memory of GPU.
Output: R the stream after operate; S the stream after operate; T the stream after operate; fresult the result for a new tuple;

图6　在CPU执行中的伪代码

内核(kernel)程序主要是有两大步骤(以查询 2 为例):a)将新元组(流 R 的元组数据)与第一个流(流 S)中所有的元组并行进行连接查询,且进行概率更新(两条元组概率相乘);b)将步骤 a)产生的结果元组与第二个流(流 T)中所有的元组并行进行连接查询,且进行概率更新(两条元组概率相乘)。所给出的伪代码如图 7 所示。

Algorithm:executed at GPU
Input: New the New Tuple of stream R; S the stream need to be matched; T the stream need to be matched; tag2 the flag value for S; tag 3 the flag value for T;
Process: for each tuple of S in each Thread, 1.Match New.a with S.d to set tag2. 2.if tag 2 equal true,calculate ρ1 × ρ2 and join the attributes. 3.Match S.e of the matched tuple of S with T.f to set tag 3. 4.if tag 3 equal true,calculate ρ1 × ρ2 × ρ3 and join the attributes. NOTE:All threads are executed in parallel.
Output: result the result after joining; tag2 the flag value for S after joining; tag3 the flag value for T after joining;

图7　在GPU中的kernel函数(查询2)

4　实验性能分析

实验所采用的 GPU 是 NVIDIA GeForce GTX460,总共有七个处理核心和 1 024 MB 的设备内存。实验所采用的 CPU 是 Intel Core 2 Quad @ 2.66 GHz,内存为 2 GB,操作系统是 Windows XP。CUDA 编译环境为版本 3.0。

4.1　与其他软硬件结合算法的比较

本文将基于 GPU 的算法与另一种软硬件结合算法进行(UWJSP[10])比较。UWJSP 也是一种处理不确定数据流窗口连接操作的协处理器,它携带有特殊的指令集专门处理各种查询操作。在 UWJSP 中,每一个 JUH 是一个独立的处理单元,并且是并行处理连接操作。这里所进行比较的是由于 JUH 数量的不同而导致的几种不同类型的 UWJSP 以及基于 GPU 的算法。如图 8 所示,可以看到当窗口较小时,UWJSP 的性能优于基于 GPU 的算法(每秒处理的新来元组数更多),并且 JUH 数量越多,性能越优。这主要是由于窗口越小,GPU 上的并行资源并没有充分得到利用,并且还需要承担一定的 GPU 存储器与内存之间数据传输的资源消耗;当窗口数量增加到 2 000 时,可以看到基于 GPU 算法的性能已经与 UWJSP 的性能持平;而在窗口大小为 4 000 时,基于 GPU 算法性能已经优于 UWJSP,这时 GPU 上的线程资源得到充分的利用。

4.2　与软件算法的比较

SA 是基于 Visual C ++ 6.0 的算法。对于每条新来的元组,不仅仅八条查询是串行的,而且在流的窗口中进行连接操作也是串行执行的。通过确定窗口大小的情况下,不断增加不确定数据流的到达数量来分析两种算法的性能,以及确定数量的不确定数据流的情况下,改变窗口的大小来比较两种算法的性能。在窗口的大小变大的同时,相应的线程数目也在增加(线程数目与窗口大小相同)。在实验过程中,所有的数据流元组(流 R、S、T 以及新产生的元组)的属性都是随机程序产生的(包括数据流的概率属性)。

首先进行比较的是在确定的窗口大小 win = 1 000 的情况下,两种算法在不同数量的数据流的运行性能。在这里将 GPU 的线程设计成 25 × 40(正好对应于窗口大小为 1 000)的形式。运行结果如图 9 所示。随着数据流数量增大,SA 算法的执行时间在数据流数量较小的情况下增长较缓慢,而当数据流增长到 10^5 的时候,执行时间快速增长。然而基于 GPU 算法的执行时间增长缓慢。当数据流数量较小的情况下(10^4 以下),基于 GPU 算法的优势并不是很明显,这主要是因为在数据量小的时候,GPU 上的并行资源并没有充分得到利用,并且还需要承担一定的 GPU 存储器与内存之间数据传输的时间消耗,随着数据量的增大,资源得到充分利用。在进一步分析 GPU 算法各部分的执行时间(表 1)时,笔者发现内核执行时间随着数据流元组数量的增大而同步上升,但是内存数据传输的执行是基本保持稳定的状况,只是很小幅度的上下浮动。原因是由于很高的 GPU 与内存的通信带宽,增加的数据流数量相对于带宽较小,因此传输的时间消耗也变化不大。

根据上一个实验发现数据流在 10^4 的情况下,两种算法的性能相当具有可比性。因此,在第二个实验中在固定数量的数据流下(10^4)比较不同窗口大小下两种算法的性能。在这里线程数目随着窗口大小的变化而变化,表 2 展示的是在不同窗口大小下线程的设计;图 10 给出了两种算法的比较结果。可以看出,当窗口较小时,两种算法的执行时间差相差不大;当窗口是 500 时,SA 算法的执行时间快速增长,而基于 GPU 算法变化不大。因此对基于 GPU 算法,窗口相对的越大越好,窗口大小接近于 GPU 可提供最大线程数时,执行效率是最高的。

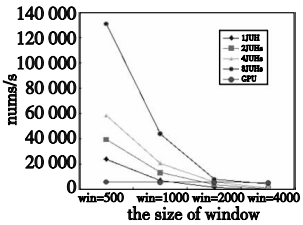

图8 不同类型的UWJSP与基于GPU算法的比较

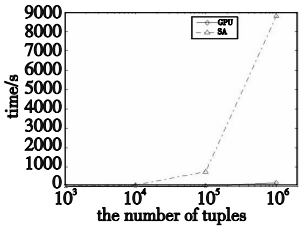

图9 不同量的不确定数据流下SA算法与基于GPU算法的比较

表1 GPU 上各部分的执行时间

Number of Tuples	10^3	10^4	10^5	10^6
Transmission	0.117	0.233	0.246	0.378
Kernel	0.667	0.936	16.668	158.626

表2 不同窗口的线程设计

Window	200	300	400	500	600	700	800	900	1000
Thread	10×20	15×20	20×20	20×25	20×30	20×35	20×40	30×30	25×40

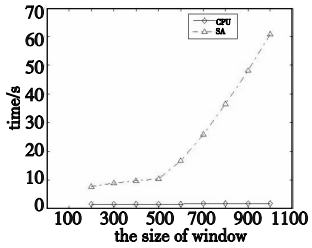

图10 不同窗口大小下SA算法与基于GPU算法的比较

5 结束语

本文提出了一种软件硬件结合的方法,在 CUDA(统一计算架构)中将数据传入 GPU 中进行并行处理。在数据流的处理过程中,连接操作是最消耗时间的,因此本文着重于数据流的窗口连接操作。实验证明了本文提出的方法可以很好地解决多数据流处理速度较慢的问题,并且数据流的数量能达到一定量时表现出来的性能较纯软件编程的性能更好。在本文中,随着要处理元组的增加,处理速度并没有得到预期的效果,这主要是由于在 GPU 的内存管理中还有待改进,这也是未来要改进的地方。

参考文献:

[1] AGGARWAL C, PHILIP S. A survey of uncertain data algorithms and applications[J]. Knowledge and Data Engineering, 2009, 21(5): 609-623.

[2] BUCK I. GPU Computing: programming a massively parallel processor [C]//Proc of International Symposium on Code Generation and Optimization. San Jose: IEEE Press, 2007: 118-135.

[3] PHARR M, FERNANDO R. Gpu gems 2: programming techniques for high-performance graphic sand general-purpose computation[M]. Boston: Addison-Wesley Press, 2005.

[4] ROBLER F, TEJADA E, FANGMEIER T. GPU-based multi-volume rendering for the visualization of functional brain images[C]//Proc of SimVis. Magdeburg: SCS Press, 2006: 305-318.

[5] RYOO S, RODRIGUES C I. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA[C]// Proc of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM Press, 2008: 73-82.

[6] GOLAB L, TAMER M. Processing sliding window multi-joins in continuous queries over data streams[C]//Proc of the 29th on Very Large Data Bases Conference. Berlin: VLDB Endowment, 2003: 500-511.

[7] DAS A, GEHRKE J, RIEDEWALD M. Approximate join processing over data streams[C]//Proc of SIGMOD Conference on Management of Data. San Diego: ACM Press, 2003: 40-51.

[8] JAYRAM T S, McGREGOR A, MUTHUKRISHNAN S. Estimating statistical aggregates on probabilistic data streams[C]//Proc of the 26th ACM Symposium on Principles of Database Systems. New York: ACM Press, 2007: 243-252.

[9] YEH M Y, WU K L, YU P S. PROUD: a probabilistic approach to processing similarity queries over uncertain data streams[C]//Proc of the 12th International Conference on Extending Database Technology. New York: ACM Press, 2009: 684-695.

[10] QIAN Jiang-bo, LI You-ming, WANG Yong-li, *et al*. An embedded co-processor for accelerating window joins over uncertain data streams [J]. Microprocessors and Microsystems-Embedded Hardware Design, 2012, 36(6): 489-504.

[11] 周茂春,陈叶芳,钱江波,等. 可重构数据流 SPJ 查询处理器的研究[J]. 计算机应用研究,2012,29(5):1781-1786.

[12] HE Bing-sheng, YANG Ke, FANG Rui. Relational joins on graphics processors[C]//Proc of SIGMOD Conference on Management of Data. Vancouver: ACM Press, 2008: 511-524.

[13] LIGOWSKI L, RUDNICKI W. An efficient implementation of smith-waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases[C]//Proc of IEEE International Workshop on High Performance Computational Biology. Rome: IEEE Press, 2009: 1-8.

[14] BAKKUM P, SKADRON K. Accelerating SQL database operations on a GPU with CUDA[C]//Proc of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. New York: ACM Press, 2010: 94-103.

[15] PIETRON M, I RUSSEK P, WIATR K. Accelerating select where and select join queries on a GPU[J]. Computer Science, 2013, 14(2): 243-252.

[16] KALDEWEY T, LOHAMN G, MUELLER R, *et al*. GPU join processing revisited[C]//Proc of the 3rd Workshop on Data Management on New Hardware. New York: ACM Press, 2012: 55-62.

[17] WU Hai-cheng, DIAMOS G, CADAMBI S, *et al*. Kernel weaver: automatically fusing database primitives for efficient GPU computation [C]//Proc of the 45th Annual IEEE/ACM International Symposium on Microarchitecture. Washington DC: IEEE Computer Society, 2012: 107-118.

[18] VARAKIN K, GAL A, KATZ O. System and method for the parallel execution of database queries over CPUs and multi-core processors: USA, 20130117. 305[P]. 2013-05-09.

[19] JO H, HONG S T, CHANG J W, *et al*. Data encryption on GPU for high-performance database systems[J]. Procedia Computer Science, 2013, 19: 147-154.

[20] 周勇,王皓,程春田,等. 基于 GPU 的多数据流相关系数并行计算方法研究[J]. 计算机应用研究,2010,27(4):1232-1235.

[21] 周勇,王皓,程春田. 使用 GPU 技术的数据流分位数并行计算方法[J]. 计算机应用,2010,30(2):543-546.

[22] JAYRAM T, KALE S, VEE E. Efficient aggregation algorithms for probabilistic data[C]//Proc of ACM-SIAM Symposium on Discrete Algorithms. Louisiana: ACM Press, 2007: 346-355.

[23] RAMAN V, DESHPANDE A, HELLERSTEIN J M. Using state modules for adaptive query processing[C]//Proc of the 19th International Conference on Data Engineering. Bangalore: IEEE Press, 2003: 353-364.