基于分层存储理论模型的近似字符串匹配并行算法研究

满都呼 宋 展

（中国科学院深圳先进技术研究院 深圳 518055）

摘 要 CUDA（Compute Unified Device Architecture）是一种重要的并行处理架构，但其具有相对复杂的线程管理机制和多重存储模块，从而使得基于 CUDA 的算法时间复杂度很难量化。针对这一问题，提出了一种分层存储理论模型—HMM（Hierarchical Memory Machine）模型，该模型所具有的分层存储结构可以有效地描述图形处理单元设备不同存储模块的物理特性，因此非常适用于对 CUDA 算法时间复杂度的量化评估。作为 HMM 模型的应用实例，文章提出了一种基于 HMM 模型的并行近似字符串匹配算法，并给出了相应算法时间复杂度的计算过程。与串行算法相比，该算法可以获得 60 倍以上的加速比。

关键词 近似字符串匹配；分层存储理论模型；并行算法；CUDA

中图分类号 TP 391.3 **文献标志码** A

A Parallel Algorithm for Approximate String Matching Based on Hierarchical Memory Machine

MAN Duhu SONG Zhan

(*Shenzhen Institutes of Advanced Technology*, *Chinese Academy of Sciences*, *Shenzhen* 518055, *China*)

Abstract CUDA(Compute Unified Device Architecture) has a complex thread organization and multi-level memory modules, which makes it difficult to quantitatively evaluate time complexity of CUDA-based algorithms. In this paper, a Hierarchical Memory Machine (HMM) Model was investigated to solve this problem. HMM is a theoretical parallel computing model, which is capable of representing the essence of computing and memory structures on the GPU(Graphics Processing Units) devices. Based on the proposed HMM model, a parallel algorithm was presented for the approximate string matching problem. The proposed algorithm is evaluated and compared with existing approaches, to show a speedup ratio of more than 60.

Keywords approximate string matching; Hierarchical Memory Machine; parallel computing; CUDA

收稿日期：2015-08-23 **修回日期**：2015-12-08

作者简介：满都呼（通讯作者），博士，副研究员，研究方向为并行计算和图像处理，E-mail：dh.man@siat.ac.cn；宋展，博士，研究员，博士生导师，研究方向为计算机视觉与 3D 重建、模式识别。

1 引 言

众所周知，最初 GPU（Graphics Processing Units）[1]是为了加速图形图像处理应用而设计的特殊电路。但现在它的应用不仅仅局限于图形图像处理，已经延伸到了一般的科学计算领域[2-6]。在实践中，我们可以利用 NVIDIA 公司所提供的并行处理架构——CUDA（Compute Unified Device Architecture）作为引擎对 NIDIA GPU 进行编程。CUDA 提供了一组虚拟命令集和多重存储模块，以便开发者对 GPU 进行操作。实践证明，因为 GPU 有着数以百计的处理单元（Processing Unit）和非常高的存储带宽，所以大多数应用 GPU 的处理速度远远高于一般的多核（Multi-Core）处理器[7]。但 CUDA 对数以万计的线程管理却是通过使用相对复杂的组织架构（包括 Grid、Block、Thread）[1]来实现的。另外 CUDA 还向用户提供了包括寄存器（Register）、共享存储器（Shared Memory）和全局存储器（Global Memory）等不同的存储模块。这些使得基于 CUDA 的算法时间复杂度很难量化。文章提出了针对 CUDA GPU 的理论模型——HMM（Hierarchical Memory Machine）模型，基于 HMM 模型，我们可以量化基于 CUDA 的算法时间复杂度：给定两个字符串 X 和 Y，长度分别为 m 和 n，$m>n$；近似字符串匹配算法（Approximate String Matching）可以找出字符串 Y 中与字符串 X 最为相近的子串，其中近似字符串匹配算法有着非常广泛的应用，包括信号处理、自然语言处理和生物信息处理等；实践中我们可以使用动态编程方法实现近似字符串匹配算法，其时间复杂度为 $O(mn)$[7]。为了加速近似字符串匹配算法，研究者们提出了不同的算法[8-14]。在本文中，我们提出了基于 HMM 模型的并行近似字符串匹配算法，并给出了算法时间复杂度，且在 NVIDIA 的 GeForce GTX 580 GPU 中实现了该算法。

2 DMM 模型、UMM 模型和 HMM 模型

在 CUDA 并行处理架构中，每个 GPU 由多个独立流处理器（Streaming Multiprocessors）组成，且每个流处理器可并行执行多个线程。另外 CUDA 还提供了两种非常重要的存储器供开发者使用——共享存储器和全局存储器[6]。每个流处理器有着局部共享存储器。在实际物理硬件中，每个流处理器的处理单元和共享内存集成在同一电路中（on-chip），所以对共享存储器的访问延迟是非常短的。然而共享存储器的容量非常有限，一般在 16～48 KBytes。不同于共享存储器，全局存储器可被每个流处理器访问。因为共享存储器的电路和流处理器的电路相对独立（off-chip），所以共享存储器的存取延迟是非常高的。然而，共享存储器有着非常大的容量，一般在 1.5～6 GBytes。

在实际编程中，还需考虑每一种存储器的访问特性。具体地说，就是共享存储器的存储体访问冲突（Bank Conflict）和全局存储器的对齐访问（Coalescing Access）特性[14-16]。在 CUDA 并行处理架构中，共享存储器的存储地址被映射到了不同的存储体中，如果不同的线程同时访问同一存储体就会导致存储体访问冲突。所以在实际编程中我们需要考虑如何避免共享存储器的访问冲突。为了最大程度利用 GPU 和 DRAM（Dynamic Random Access Memory）之间的存储带宽并减少全局存储器的访问延迟，下标连续的不同线程需访问全局存储器的连续地址。这就是全局存储器的对齐访问问题。

下面我们简单介绍三种并行计算模型——DMM（Discrete Memory Machine）模型、UMM（Unified Memory Machine）模型和 HMM 模型，它们反映了 CUDA 共享存储器和全局存储器的基本特征。如图 1 所示，在 DMM 模型和 UMM 模型中，一组线程通过存储管理单元

(Memory Management Unit，MMU)连接到了存储体。每个线程可被视为独立的随机存取机(Random Access Machine)[17]。在一个单元时间内，它可以独立执行一个基本操作。所有的存储体组成了连续的存储空间。如图 1 所示，DMM 和 UMM 的主要区别在于连接到存储管理单元的地址线(Address Line)的连接方式不同。在 DMM 中，每个存储体都有地址线连接到存储管理单元，而在 UMM 中，只有一个地址线连接到存储管理单元。这就是说在 UMM 中相同的地址会以广播的形式发送到每个存储体，因而所有的存储体中偏移量相同的地址可被同时访问。而在 DMM 中不同的存储体可被同时访问。HMM 模型是 DMM 模型和 UMM 模型的组合。HMM 模型反映了 CUDA GPU 的多重存储架构特性。如图 2 所示，HMM 由 d 个 DMM 和 1 个 UMM 组成。UMM 及每个 DMM 都由 w 个存储体组成。我们命名每个 DMM 的存储体为共享存储器，UMM 的存储体为全局存储器。这也对应了 CUDA 构架中的共享存储器和全局存储器。通过使用自身的共享存储器，每个 DMM 可独立执行不同操作。所有 DMM 的全部线程可以访问同一个 UMM 的全局存储器。在 NVIDIA GPU 中，对共享内存的访问一般只需几个时钟周期，而对全局存储器的访问需要数以百计的时钟周期[6]。所以我们用 l 和 L 分别代表共享存储器的访问延迟和全局存储器的访问延迟，假设 $l<<L$。

首先，我们给出 DMM 模型定义。假设 DMM 的存储带宽为 w，存储延迟为 l，$m[i]$ ($i\geqslant 0$)代表地址为 i 的存储单元，$B[j]=\{m[j], m[j+w], m[j+2w], m[j+3w], \cdots\}$ ($0\leqslant j\leqslant w$)代表 DMM 的第 j 个存储体，则存储单元 $m[i]$必然位于第(i mod w)个存储体内。显然位于不同存储体内的存储单元可被同时访问，而位于同一存储体内的不同存储单元不可以被同时访问。同一存储体内不同存储单元的并行访问是流水线模式(Pipeline Fashion)处理的，所以对同一存储体内 k 个不同存储单元的访问需要 $k+l-1$ 个单位时

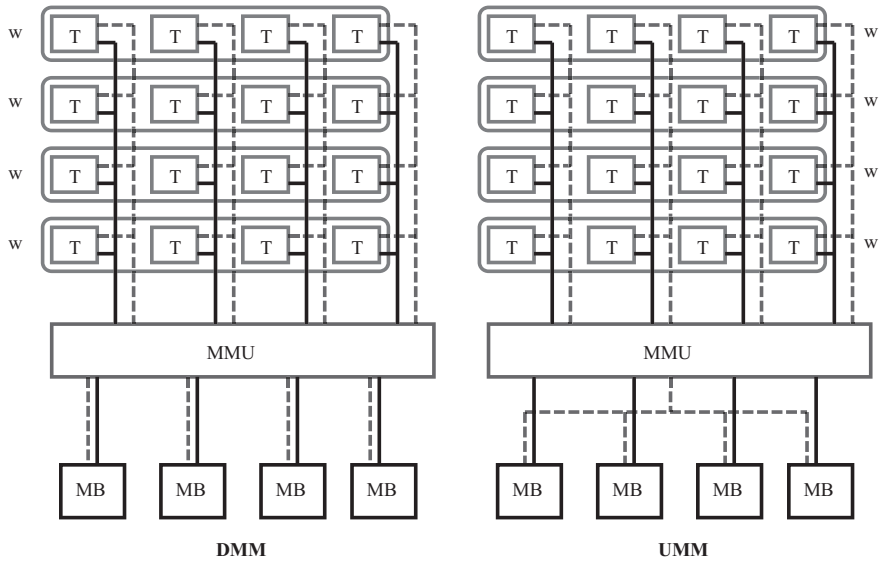

图 1　DMM 和 UMM 模型（$W=4$）

Fig. 1　DMM and UMM model with $W=4$

间。假设每个 DMM 每次可执行 w 个线程，我们把 p 个线程分成 p/w 个组，用 $W(i)$ $(0\leqslant i\leqslant p/w-1)$ 代表一个组，则所有组 $W(0), W(1), \cdots, W(p/w-1)$ 的执行是根据 Round Robin 规则进行的。而且每个线程的存储访问请求只能等到前一个访问请求完成之后才能执行。所以每个线程至少等待 l 个单位时间后才能发新的存储访问请求。

其次，我们给出 UMM 模型定义。假设 UMM 的存储带宽为 w，存储延迟为 L，$A[j]=\{m[j\cdot w], m[j\cdot w+1], \cdots, m[(j+1)\cdot w-1]\}$代表第 j 组地址。同一组地址内的存储单元可被同时访问。而不同组的访问是按 Round Robin 方式处理的。

最后，我们给出 HMM 模型定义。如图 2 所示，HMM 由多个 DMM 和 1 个 UMM 组成，且每个 DMM 独立工作。UMM 及每个 DMM 都由 w 个存储体组成。我们命名每个 DMM 的存储体为共享存储器，UMM 的存储体为全局存储器。所有线程被分成多个组，每组包含 w 个线程。每组线程可访问当前执行这组线程的 DMM 的共享存储器。不同于 DMM 共享存储器的局部访问特性，所有 DMM 的全部线程都可以访问全局存储器。

3 对齐访问和无冲突访问

因为对齐访问和无冲突访问对我们所提出的近似字符串匹配并行算法性能有着重要的影响，所以接下来简述对齐访问对全局存储器性能的影响及无冲突访问对共享存储器性能的影响。

当一个线程组内所有线程完成一次对共享存储器或全局存储器的访问，则一轮存储访问结束。如果一个线程组内的所有线程访问了全局存储器内的同一组地址，则称这一轮存储访问为对齐访问。与此类似，如果一个线程组内的所有线程访问了共享存储器的不同存储体，则称这一轮存储访问为无冲突访问。具体地说，对于线程组的线程 $T(i)$ $(0\leqslant i\leqslant w-1)$，如果 $T(i)$ 所访问

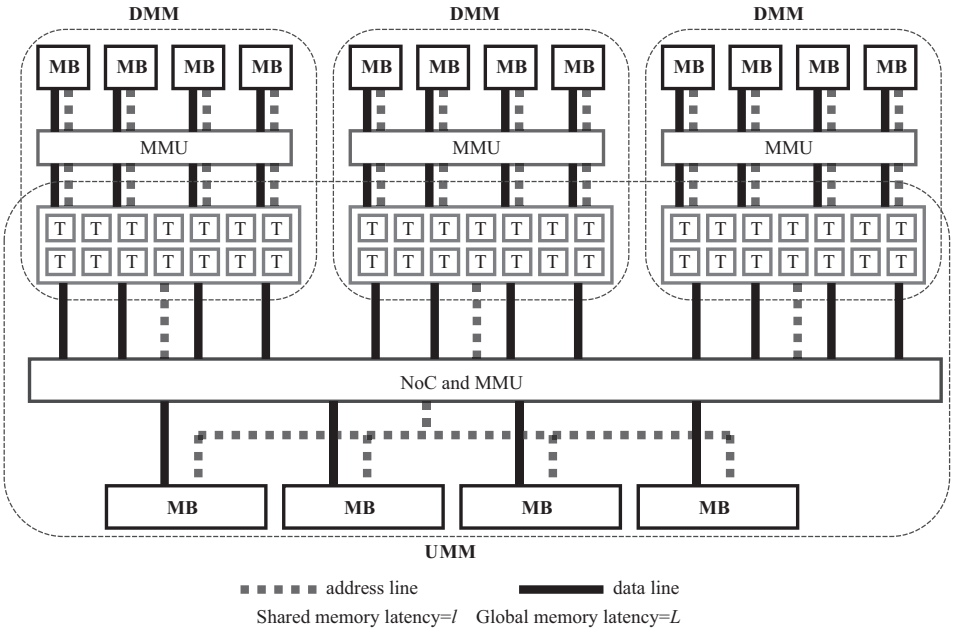

图 2 HMM 结构示意图

Fig. 2 Architecture of HMM

的全局存储器地址 $a(i)$ 满足条件 $\left\lfloor \frac{a(0)}{w} \right\rfloor = \left\lfloor \frac{a(1)}{w} \right\rfloor = \cdots = \left\lfloor \frac{a(w-1)}{w} \right\rfloor$，则称这一轮存储访问是对齐访问；如果线程组内线程 $T(i)$ 所访问的是共享存储器且访问地址 $a(i)$ 满足条件 $i \equiv j \ (0 \leqslant i, j \leqslant w-1)$ 或 $i \bmod w \neq j \bmod w$，则称这一轮存储访问为无冲突访问。如果所有线程组都可实现对齐访问或无冲突访问且所有线程组总共有 p 个线程，则称这 p 个线程也实现了一轮对齐访问或无冲突访问。

p 个线程完成一轮对齐访问或无冲突访问所需要的时间如下：

(1) 假设 p 个线程对全局存储器进行对齐访问。p 个线程可被分成 p/w 个组，每组发出 w 个存储访问请求，则 p 个线程的所有访问请求需要 p/w 个单位时间。又因全局存储器的访问延迟为 L，所以 p 个线程的一轮对齐访问需要 $p/w+L-1$ 个单位时间，即时间复杂度为 $O(p/w+L)$。

(2) 假设 p 个线程对共享存储器进行访问。每个线程组发出 w 个存储访问请求，则 p 个线程的所有访问请求需要 p/w 个单位时间发送。共享存储器的访问延迟是 l，所有 p 个线程的一轮无冲突访问需要 $p/w+l-1$ 个单位时间，即时间复杂度为 $O(p/w+l)$。

(3) 如果 p 个线程需要访问全局存储器的 n 个字 (word) 则需要 n/p 轮存储访问。如果每轮存储访问都是对齐的，则需要 $O(p/w+L)\cdot n/p = O(n/w+nL/p)$ 个单位时间。与此类似，如果 p 个线程需访问共享存储器的 n 个字，则需要 $O(n/w+nl/p)$ 个单位时间。

引理 1 全局存储器的 n 个字的对齐访问和共享存储器的 n 个字的无冲突访问分别需要 $O(n/w+nL/p)$ 和 $O(n/w+nl/p)$ 个单位时间。其中，w 代表存储带宽，L 代表全局存储器访问延迟，l 代表共享存储器访问延迟，p 代表线程个数。

4　近似字符串匹配及 ED 距离

在这一章我们简述近似字符串匹配及 ED 距离 (Edit Distance)。更详细内容可参考文献[18]。

首先给出 ED 距离的定义。假设给定两个字符串 $X=x_1, x_2, \cdots, x_m$ 和 $Y=y_1, y_2, \cdots, y_n (m<n)$，我们用下面三种操作把字符串 X 变换成字符串 Y：

(1) 插入字符串；

(2) 删除字符串；

(3) 替换字符串。

例如，假设 $X=ababa$，$Y=aaabbb$，现在我们可以用如下操作把 X 变换成 Y：$X=ababa \xrightarrow{\text{删除}} aaba \xrightarrow{\text{删除}} aaa \xrightarrow{\text{插入}} aaab \xrightarrow{\text{插入}} aaabb \xrightarrow{\text{插入}} aaabbb=Y$。或者用如下操作也可将 X 变换成 Y：$X=ababa \xrightarrow{\text{替换}} aaaba \xrightarrow{\text{替换}} aaabb \xrightarrow{\text{插入}} aaabbb=Y$。显然，第二种方法用了更少的操作。两个字符串 X 与 Y 之间的 ED 距离就是变换 X 到 Y 所用的最少操作数。在接下来的章节中我们将用 $\mathrm{ED}(X, Y)$ 代表两个字符串之间的 ED 距离。

近似字符串匹配的公式化定义就是：$\mathrm{ASM}(X, Y)=\min\{\mathrm{ED}(X, Y') \mid Y' \text{ 是字符串 } Y \text{ 的子串}\}$。可以用动态编程来求得 $\mathrm{ASM}(X, Y)$ 值。假设用一个有 $(m+1)\times(n+1)$ 元素的二维矩阵 $\boldsymbol{c}$ 计算 $\mathrm{ASM}(X, Y)$。矩阵 $\boldsymbol{c}$ 的每个元素 $c[i][j]$ $(0 \leqslant i \leqslant m,\ 0 \leqslant j \leqslant n)$ 将存储下面的值：$\min\limits_{1 \leqslant j' \leqslant j} ED(x_1 x_2 \cdots x_i, y_{j'} y_{j'+1} \cdots y_j)$。一旦得到矩阵 $\boldsymbol{c}$ 的所有元素值，就可以根据下面的公式计算 $\mathrm{ASM}(X, Y)$ 的值：$\mathrm{ASM}(X, Y) = \min\limits_{1 \leqslant j' \leqslant j} \{c[m][j]\}$。下面我们给出计算矩阵 $\boldsymbol{c}$ 每个元素的方法：

$$
\begin{aligned}
c[i][j] &= 0 \quad if \quad i=0 \\
&= i \quad if \quad j=0 \\
&= \min\{c[i][j-1]+1, c[i-1][j]+ \\
&\quad 1, c[i-1][j-1]+(x_i \neq y_j)\} \quad i, j>0
\end{aligned}
$$

如果字符 x_i 与字符 y_j 相同，则 $x_i \neq y_j$ 返回值为 0，否则返回值为 1。计算 ASM(*X*, *Y*)的伪代码如下：

[Sequential ASM Algorithm]

for $j \leftarrow 1$ to n do $c[0][j] \leftarrow 0$
for $i \leftarrow 1$ to m do $c[i][0] \leftarrow i$
for $i \leftarrow 1$ to m do
  for $j \leftarrow 1$ to n do
    $c[i][j] \leftarrow \min\{c[i][j-1]+1, c[i-1][j]+1, c[i-1][j-1]+(x_i \neq y_j)\}$
output $\min\{c[m][j] \mid 0 \leqslant j \leqslant n\}$

图 3 给出了 $X=ababa$、$Y=aaabbb$ 时矩阵 ***c*** 每个元素的值。如图所示，ASM(*X*, *Y*)的值是 1。当矩阵 ***c*** 所有元素值被确定后，我们可以非常容易找到字符串 *Y* 中与字符串 *X* 近似匹配的子串位置。

		Y								
			a	*a*	*a*	*b*	*b*	*b*	*a*	*a*
		0	0	0	0	0	0	0	0	0
	a	1	0	0	0	1	1	1	0	0
	b	2	1	1	1	0	1	1	1	1
X	*a*	3	2	1	1	1	1	2	1	1
	b	4	3	2	2	1	1	1	2	2
	a	5	4	3	2	2	2	2	1	2

图 3 矩阵 *c* 元素值

Fig. 3 Elements of matrix *c*

5 基于 DMM 模型的并行近似字符串匹配算法

基于单个 DMM 的并行近似字符串匹配算法如下：给定两个字符串 *X* 和 *Y*，长度分别为 *m* 和 *n*，且假设它们被存储在 DMM 的共享存储器中。我们将在共享存储器中创建大小为 $(m+1)\times(n+1)$ 的矩阵 ***c*** 用于计算 ASM(*X*, *Y*)的值。如图 4 所示，我们将计算矩阵 ***c*** 元素值从矩阵左上角至右下角。

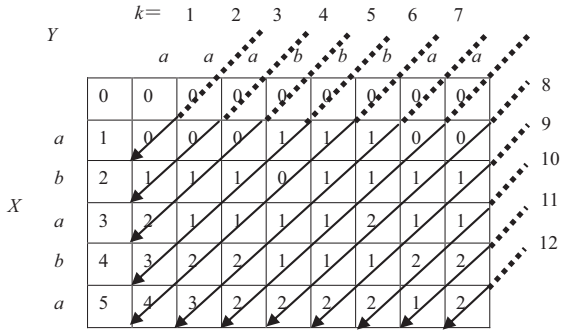

图 4 计算 *c* 元素值的并行算法，每个实斜线中的元素可被同时计算

Fig. 4 The parallel algorithm for computing elements of matrix *c*, where elements with same diagonal line can be computed in parallel

[Parallel ASM Algorithm]

for $j \leftarrow 1$ to n do in parallel $c[0][j] \leftarrow 0$
for $i \leftarrow 1$ to m do in parallel $c[i][0] \leftarrow i$
for $k \leftarrow 1$ to $n+m-1$ do
  for $i \leftarrow 1$ to m do in parallel
    begin
      $j \leftarrow k-i+1$
      if $1 \leqslant j \leqslant n$ then
      $c[i][j] \leftarrow \min\{c[i][j-1]+1, c[i-1][j]+1, c[i-1][j-1]+(x_i \neq y_j)\}$
    end
output $\min\{c[m][j] \mid 0 \leqslant j \leqslant n\}$

从上述算法容易发现，当计算与某个 *k* 值对应的矩阵 ***c*** 元素值时，只需要与 $k-1$ 及 $k-2$ 对应的矩阵 ***c*** 元素值。例如，如图 4 所示，当计算 $k=6$ 时的矩阵元素值时，只需预先知道与 $k=5$ 和 $k=4$ 对应的矩阵元素值。所以大小为 $3\times(m+1)$ 的矩阵足以满足与 *k* 对应的矩阵元素计算。这可以大大减少上述并行算法所需要的存储空

间。现命名大小为 $3\times(m+1)$ 的矩阵为 $\boldsymbol{e}$，下面为改进的并行算法：

[Improved Parallel ASM Algorithm]

$minval \leftarrow m; e[0][0] \leftarrow 0;\ e[0][1] \leftarrow 1;\ e[1][0] \leftarrow 0$
for $k \leftarrow 1$ to $n+m-1$ do
  begin
    for $i \leftarrow 1$ to m do in parallel
      begin
        $j \leftarrow k-i+1$
        if $i=0$ then $e[j \bmod 3][i] \leftarrow 0$
        else if $j=0$ then $e[j \bmod 3][i] \leftarrow i$
        else if $1 \leqslant j \leqslant n$ then
          $e[j \bmod 3][i] \leftarrow \min\{e[(j-1) \bmod 3][i]+1, e[j \bmod 3][i-1]+1, e[(j-1) \bmod 3][i-1]+(x_i \neq y_j)\}$
      end
if $k \geqslant m$ and $e[j \bmod 3][m] < minval$ then $minval \leftarrow e[j \bmod 3][m]$
  end
output $minval$

接下来给出基于 DMM 的并行近似字符串匹配算法的时间复杂度。显然 $e[0][0]$、$e[0][1]$ 和 $e[1][0]$ 的初始化可由一个线程完成，其时间复杂度为 $O(1)$。另外，我们可以让矩阵 $\boldsymbol{e}$ 的列数为 w 的整倍数，这可以实现对矩阵 $\boldsymbol{e}$ 的无冲突访问。从上面的改进算法可知，对应每个 k 值，我们需要读取矩阵 $\boldsymbol{e}$ 的两行并把结果写入矩阵 $\boldsymbol{e}$ 相应行中。根据**引理** 1 可知，此过程需要的时间为 $O(m/w+ml/p)$。又因 k 的值是从 1 到 $n+m-1$，所以整个改进算法需要的时间为 $(n+m-1)\cdot O(m/w+ml/p)=O(mn/w+mnl/p)$。

引理 2　对于长度分别为 m 和 $n(m<n)$ 的俩个字符串 X 和 Y，基于 DMM 的并行近似字符串匹配算法的时间复杂度为 $O(mn/w+mnl/p)$。其中 w 代表存储带宽；l 代表共享存储器访问延迟；p 代表线程个数。

6　基于 HMM 模型的并行近似字符串匹配算法

基于 HMM 的并行近似字符串匹配算法如下：给定两个字符串 X 和 Y，长度分别为 m 和 n，且假设它们被存储在 HMM 的全局存储器中。如果 HMM 包含 d 个 DMM，则假设 $n \geqslant dw$，$p \geqslant dw$。因为 dw 个线程可被 HMM 并行执行，这也是做如上假设的原因。

由 ED 距离的定义容易得出 $n-m \leqslant \mathrm{ED}(X, Y) \leqslant n$。因此如果 $n>2m$，则有 $\mathrm{ED}(X, Y)>m$。另外由 $\mathrm{ASM}(X, Y)$ 的定义可知 $\mathrm{ASM}(X, Y) \leqslant m$。因此在匹配算法中，可以忽略长度大于 $2m$ 的 Y 的子串。$\mathrm{ASM}(X, Y)$ 可被重新定义如下：$\mathrm{ASM}(X, Y)=\min\{\mathrm{ED}(X, Y') \mid Y'$ 是 Y 的子串且长度小于或等于 $2m\}$。假设 $Y_0, Y_1, \cdots, Y_{d-1}$ 为 Y 的 d 个子串，且 Y 的任一长度小于或等于 $2m$ 的子串都被至少一个 $Y_i(0 \leqslant i \leqslant d-1)$ 所包含，则 $\mathrm{ASM}(X, Y)$ 的定义还可被写成：$\mathrm{ASM}(X, Y)=\min\{\mathrm{ED}(X, Y_i) \mid 0 \leqslant i \leqslant d-1\}$。我们可对所有 $\mathrm{ASM}(X, Y_i)$ 进行并行计算，最终得到 $\mathrm{ASM}(X, Y)$ 的值。设 $Y_i=y_{is}y_{is+1}\cdots y_{(i+1)s+2m-1}(0 \leqslant i \leqslant d-1$，$s=(n-2m)/d$，则 Y 的任一长度小于或等于 $2m$ 的子串一定被至少一个 $Y_i(0 \leqslant i \leqslant d-1)$ 所包含。例如，如果 $n=1\ 024$、$m=32$、$d=4$，则可得 $s=240$，$Y_0=y_0y_1\cdots y_{303}$，$Y_1=y_{240}y_{241}\cdots y_{543}$，$Y_2=y_{480}y_{481}\cdots y_{783}$，$Y_3=y_{720}y_{721}\cdots y_{1023}$。此时，可以确定长度小于或等于 $2m=64$ 的子串至少被 Y_0、Y_1、Y_2、Y_3 中的某一个所包含。基于 HMM 的并行近似字符串匹配算法步骤如下：

(1) 每个 $\mathrm{DMM}(i)$ 从全局存储器中读取字符串 X 和对应的 Y_i 到共享存储器中；

(2) 所有 $\mathrm{DMM}(i)$ 并行计算对应的 $\mathrm{ASM}(X,$

Y_i)；

(3)每个 DMM(i)把 ASM(X, Y_i)的计算结果写入全局存储器中；

(4)计算 min{ ASM(X, Y_i) | $0 \leqslant i \leqslant d-1$}。

假设线程数为 p 且 HMM 包含 d 个 DMM，每个 DMM 包含 p/d 个线程。下面我们给出基于 HMM 的并行算法时间复杂度。在步骤(1)中 d 个 DMM 从全局存储器中读取字符串 X，换句话说就是 p 个线程从全局存储器中读取 md 个字符，根据**引理** 1 可知，这需要 $O(md/w+mdL/p)$ 单位时间。与此类似，每个 Y_i 的读取需要 $O[(s+2m)d/w+(s+2m)dL/p]$ 单位时间。根据**引理** 1 还可得，字符串 X 和 Y_i 的写入(写入到共享存储器)分别需要 $O(m/w+ml/p)$ 单位时间和 $O[(s+2m)/w+(s+2m)l/p]$ 单位时间。所以当 $s<n/d$ 时，步骤(1)的时间复杂度为 $O[(s+m)d/w+(s+m)dL/p]=O[(n+md)/w+(n+md)L/p]$。在步骤(2)中每个 DMM($i$)将独立计算对应的 ASM($X$, Y_i)，根据**引理** 2，可知这需要 $O[(s+2m)m/w+(s+2m)ml/(p/d)] \leqslant O[(n+md)m/dw+(n+md)ml/p]$ 单位时间。在步骤(3)中每个 DMM(i)的一个线程将 ASM(X, Y_i)的结果写入全局存储器。又因 HMM 有 d 个 DMM，所以步骤(3)的时间复杂度为 $O(d+L) \leqslant O(n/w+L)$。在步骤(4)中 UMM 的 p 个线程求得所有 ASM(X, Y_i)中最小值，我们可用参考文献[19]中所提的算法得到这个最小值。步骤(4)的时间复杂度为 $O(d/w+dL/p+L)$。最终我们可得计算 ASM(X, Y)的总时间为 $O[(n+md)/w+m(n+md)/dw+(n+md)L/p+m(n+md)l/p]$。

引理 3 对于长度分别为 m 和 $n(m<n)$的两个字符串 X 和 Y，基于 HMM 的并行近似字符串匹配算法的时间复杂度为 $O[(n+md)/w+m(n+md)/dw+(n+md)L/p+m(n+md)l/p]$。其中，$d$ 代表 HMM 中 DMM 个数；w 代表存储带宽；l 代表共享存储器访问延迟；L 代表全局存储器访问延迟；p 代表线程个数。

从**引理 3** 可知，基于 HMM 的并行算法的延迟开销为 $O[(n+md)L/p+m(n+md)l/p]$，且当 $p=md$ 时延迟开销所用的时间可被减少到 $O(nL/md+nl/d+L+ml)$。当 m 和 d 的值较小时，算法时间复杂度将由延迟开销所主导。也就是说，需要用数量多于 md 的线程来减少延迟开销。具体地说，就是可以增加线程数量为 $p=mD(d<D \leqslant n)$。字符串 Y 的子串 Y_i 的个数也由 d 变为 D，即可得子串 $Y_0, Y_1, \cdots, Y_{D-1}(d<D \leqslant n)$，且每个子串 $Y_i(0 \leqslant i \leqslant d-1)$的长度为 $S+2m$，$S=(n-2m)/D$。我们用 m 个线程计算每个 ASM(X, Y_i)。换句话说就是 $p=mD$ 个线程将由 d 个 DMM 执行，每个 DMM 用 mD/d 个线程计算 D/d 个 ASM(X, Y_i)的值。至此，算法时间复杂度可改写如下：步骤(1)的时间复杂度为 $O[(n+mD)/w+(n+mD)L/p]$；步骤(2)的时间复杂度为 $O[(n+mD)m/dw+(n+mD)ml/p]$；步骤(3)的时间复杂度为 $O(D+L)<O(n/w+L)$；步骤(4)的时间复杂度为 $O(D/w+DL/p+L)$。总时间复杂度为 $O[(n+mD)/w+m(n+mD)/dw+(n+mD)L/p+m(n+mD)l/p]=O[(n+p)/w+m(n+p)/dw+nL/p+mnl/p+L+ml]$。

定理 4 对于长度分别为 m 和 $n(m<n)$的两个字符串 X 和 Y，基于 HMM 的并行近似字符串匹配算法的时间复杂度为 $O[(n+p)/w+m(n+p)/dw+nL/p+mnl/p+L+ml]$。其中，$d$ 代表 HMM 中 DMM 个数；w 代表存储带宽；l 代表共享存储器访问延迟；L 代表全局存储器访问延迟；p 代表线程个数。

7 实验与分析

实验分别在 Intel Xeon CPU X7460(2.66GHz)和 GeForce GTX 580 GPU 上实现了串行算法和基于 HMM 的并行算法。编译环境分别为 gcc 4.4.4-14 和 CUDA 4.2，操作系统版本为 Ubuntu 10.10

(32-bit)。GeForce GTX 580 GPU 有 16 个流处理器，且每个流处理器每次可执行 32 个线程，也就是说，线程组线程个数为 32。表 1 给出了当字符串 Y 的长度为 4 M(=2^{22})，X 的长度为 32、64、128、256、512、1 024 时的串行算法和并行算法的运行时间。在并行算法的实现中，通过改变 Y 的子串 Y_i 的长度找出最短运行时间。通过改变 Y 的子串 Y_i 的长度，可得到不同子串个数，分别为 16、32、64、128、256、512、1 024、2 048。显然子串个数等于 CUDA 线程块(block)个数。实验中字符串 X 和 Y 是由 8-bit 的随机 0 或 1 组成，$x_i \neq y_j$ 的概率为 1/2。这样的字符串对 GPU 实现是非常不利的，因为对 x_i 是否等于 y_i 的判断会导致 GPU 线程组的 branch divergence 问题。从表 1 可知，当字符串 Y 的长度为 4 M(=2^{22})、X 的长度为 1 024、CUDA 线程块个数等于 16 时，GPU 实现比 CPU 实现可达到 66.1 倍加速。

在 GPU 实现中，我们用到了 CUDA 所提供的 Shuffle 命令[6]来找出最终的 ASM 最小值。因为 Shuffle 命令是在 Register 间完成的，这节省了 CUDA 共享存储器空间并减少了并行步骤中同步命令的使用。图 5 给出了只有 3 个线程组情况下选出最小值的方法。

根据**定理 4** 可知，所实现的算法时间复杂度为 $O[(n+p)/w+m(n+p)/dw+nL/p+mnl/p+L+ml]$。它由两部分组成，分别为 $O[(n+p)/w+m(n+p)/dw]$和 $O(nL/p+mnl/p+L+ml)$。显然前部分表达了存储带宽和理论模型中的 DMM 个

表 1　基于 HMM 的并行字符串匹配算法在 GeForce GTX 580 中执行时间(毫秒)

Table 1　Running time (milliseconds) of HMM based parallel ASM algorithm on GeForce GTX 580

$\|X\|=m$	CUDA 线程块个数								串行算法时间（CPU）	加速倍数
	16	32	64	128	256	512	1 024	2 048		
32	178.2	89.23	44.29	23.46	23.53	23.64	23.90	24.38	701.8	29.9
64	178.6	89.71	46.74	29.13	29.13	29.18	29.39	30.01	1 364	46.8
128	181.8	92.66	55.81	48.40	48.51	48.92	50.16	53.36	2 683	55.4
256	187.0	112.2	95.77	93.16	91.83	93.83	100.1	113.3	5 295	57.7
512	236.5	191.1	184.4	181.3	185.0	197.9	224.1	277.9	10 560	58.2
1 024	419.6	423.8	432.3	449.4	483.4	551.5	687.7	960.0	27 720	66.1

注：$|Y|=4M(=2^{22})$

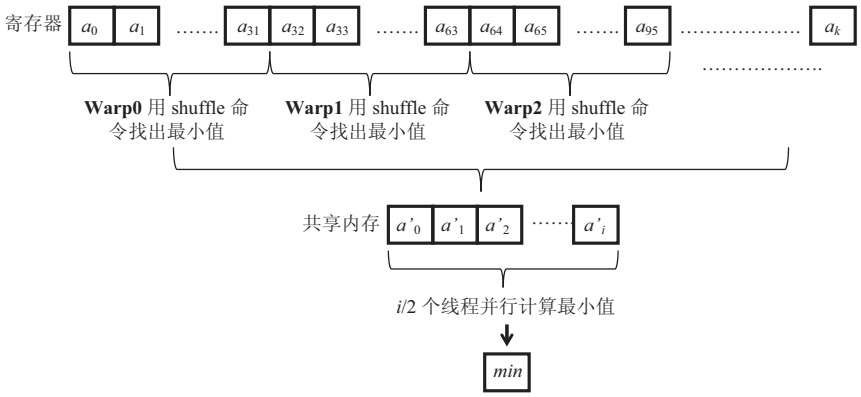

图 5　基于 Shuffle 命令的 GPU 最小值计算方法

Fig. 5　GPU computation of minimum value with Shuffle operation

数 d(GPU 实现中的 CUDA 线程块个数)对整个算法的影响，而后部分表达了存储器延迟对整个算法的影响。为了均衡这两个因素，需要选择适当的线程个数，即对 p 选择适当的值。例如当字符串 X 的长度 $m=512$ 时，设置 CUDA 线程块的个数为 128(此时线程数 $p=512\times128=65\,536$)即可得最少执行时间 181.3 ms。这也说明实验结果是符合理论分析。

表 2 给出了我们的 GPU 实现与参考文献中所提到的 GPU 实现之间的对比。从表中可知，尽管参考文献[12,13]中使用了较快的比特位操作，我们的 GPU 实现还是达到了更高的吞吐量(Throughput)。而 Liu 等[11]所提到的算法实现了最高的吞吐量，这是因为他们对所提到的算法对匹配长度进行了限制(K-mismatch)。

8 结 论

在本文中，我们提出了针对 NVIDIA GPU 多重存储构架的理论模型——HMM 模型。基于 HMM 模型我们可以量化给定算法在基于 CUDA 的 GPU 中执行时间复杂度。同时，我们提出了基于 HMM 模型的并行近似字符串匹配算法，并在 NVIDIA 的 GeForce GTX 580 GPU 中实现了该算法。当假设 HMM 模型有 p 个线程，d 个流处理器，存储器带宽为 w，全局存储器存取延迟为 L，共享存储器存取延迟为 l，则所提出的并行算法时间复杂度为 $O[(n+p)/w+m(n+p)/dw+nL/p+mnl/p+L+ml]$。当字符串 X 的长度为 1 024、字符串 Y 的长度为 4 M($=2^{22}$)时，我们的 GPU 实现仅用了 419.2 ms，而 CPU 实现用了 27 720 ms，我们的 GPU 实现可达到了 66 倍的加速。

参 考 文 献

[1] NVIDIA Corporation. NVIDIA CUDA C programming guide version 7.5 [OL]. [2015-08-23]. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[2] Wen-Mei WH. GPU Computing Gems Emerald Edition [M]. Elsevier, 2011.

[3] Ogawa K, Ito Y, Nakano K. Efficient canny edge detection using a GPU [C] // Proceedings of International Conference on Networking and Computing, 2010: 279-280.

[4] Man D, Uda K, Ito Y, et al. A GPU implementation of computing Euclidean distance map with efficient memory access [C] // IEEE 2011 Second International Conference on Networking and Computing, 2011: 68-76.

[5] Uchida A, Ito Y, Nakano K. Fast and accurate template matching using pixel rearrangement on the GPU [C] // IEEE 2011 Second International Conference on Networking and Computing, 2011: 153-159.

[6] Ito Y, Ogawa K, Nakano K. Fast ellipse detection algorithm using Hough transform on the GPU [C] //

表 2 与其他 GPU 实现的比较

Table 2 Comparison to other GPU implementations

相关文献	字符串 X 长度	字符串 Y 长度	使用 NVIDIA GPU	运行时间（ms）	使用的方法	吞吐量 (MB/s)
本文算法	1 024	4 M	GTX580	419.1	动态编程	39.10
Liu *et al.*[11]	256	8 M	GTX260	100	海明距离	81.92
Xu *et al.* [12]	17	1	GeForce 310M	31.25	Bit Parallel	2.17
Onsjo *et al.*[13]	1 024	160	NVIDIA T1	150 000	Bit Parallel	4.36

IEEE 2011 Second International Conferenceon Networking and Computing, 2011: 313-319.

[7] Sellers PH. The theory and computation of evolutionary distances: pattern recognition [J]. Journal of Algorithms, 1980, 1(4): 359-373.

[8] Ukkonen E. Algorithms for approximate string matching [J]. Information and Control, 1985, 64(1): 100-118.

[9] Myers G. A fast bit-vector algorithm for approximate string matching based on dynamic programming [J]. Journal of the ACM, 1999, 46(3): 395-415.

[10] Utan Y, Inagi M, Wakabayashi S, et al. A GPGPU implementation of approximate string matching with regular expression operators and comparison with its FPGA implementation [C] // Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2012.

[11] Liu Y, Guo LJ, Li JB, et al. Parallel algorithms for approximate string matching with k mismatches on CUDA [C] // 2012 IEEE 26th International Conference on Parallel and Distributed Processing Symposium Workshops & PhD Forum, 2012: 2414-2422.

[12] Xu K, Cui W, Hu Y, et al. Bit-parallel multiple approximate string matching based on GPU [J]. Procedia Computer Science, 2013, 17: 523-529.

[13] Onsjo M, Aono Y. Online Approximate String Matching with CUDA [OL]. 2009-12-18[2015-08-23]. http://pds13.egloos.com/pds/200907/26/57/pattmatch-report.pdf.

[14] Man D, Uda K, Ueyama H, et al. Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs [J]. International Journal of Networking and Computing, 2011, 1(2): 260-276.

[15] NVIDIA Corporation. NVIDIA CUDA C best practice guide [OL]. 2010 [2015-08-23]. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3tyDaSGZp.

[16] Nishida K, Nakano K, Ito Y. Accelerating the dynamic programming for the optimal polygon triangulation on the GPU [M] // Algorithms and Architectures for Parallel Processing, Springer Berlin Heidelberg, 2012: 1-15.

[17] Aho AV, Hopcroft JE, Ullman JD. Data Structures and Algorithms [M]. Addison Wesley, 1983.

[18] Rivest RL, Leiserson CE. Introduction to Algorithms [M]. McGraw-Hill, Inc., 1990.

[19] Harris M, Sengupta S, Owens JD. Parallel prefix sum (scan) with CUDA [M] // GPU Gems, 2007: 851-876.