基于CPU-GPU混合计算平台的RNA二级结构预测算法并行化研究*

夏 飞,朱强华,金国庆
(海军工程大学 电子工程学院,湖北 武汉 430033)

摘 要:RNA二级结构预测是生物信息学领域重要的研究方向,基于最小自由能模型的Zuker算法是目前该领域最典型使用最广泛的算法之一。本文基于CPU+GPU的混合计算平台实现了对Zuker算法的并行和加速。根据CPU和GPU计算性能的差异,通过合理的任务分配策略,实现二者之间的并行协作计算和处理单元间的负载平衡;针对CPU和GPU的不同硬件特性,对Zuker算法在CPU和GPU上的实现分别采取了不同的并行优化方法,提高了混合加速系统的计算性能。实验结果表明,CPU处理单元在混合系统中承担了14%以上的计算任务,与传统的多核CPU并行方案相比,采用混合并行加速方法可获得15.93的全局加速比;与最优的单纯GPU加速方案相比,可获得16%的性能提升,并且该混合计算方案可用于对其它生物信息学序列分析应用的并行和加速。

关键词:生物信息学;RNA二级结构预测;最小自由能;混合加速方法
中图分类号:TP302 **文献标志码**:A **文章编号**:1001-2486(2013)06-0138-09

Accelerating RNA secondary structure prediction applications based on CPU-GPU hybrid platforms

XIA Fei, *ZHU Qianghua*, *JIN Guoqing*
(Electronic Engineering College, Naval University of Engineering, Wuhan 430033, China)

Abstract: Prediction of ribonucleic acid (RNA) secondary structure remains to be one of the most important research areas in bioinformatics. The Zuker algorithm is one of the most popular methods of free energy minimization for RNA secondary structure prediction. However, general-purpose computers including parallel computers or multi-core computers exhibit parallel efficiency of no more than 50% on Zuker. For this problem, a CPU-GPU hybrid computing system that accelerates the Zuker algorithm applications for RNA secondary structure prediction is proposed. The computing tasks were allocated between CPU and GPU for parallel cooperate execution. Performance differences between the CPU and the GPU in the task-allocation scheme were considered to obtain workload balance. To improve the hybrid system performance, the Zuker algorithm was optimally implemented with special methods for CPU and GPU architecture. A speedup of 15.93× over optimized multi-core SIMD CPU implementation and performance advantage of 16% over optimized GPU implementation were shown in the experimental results. More than 14% of the sequences were executed on CPU in the hybrid system. To the best of our knowledge, our implementation combining CPU and GPU is the only accelerator platform implementing the complete Zuker algorithm. Moreover, the hybrid computing system is proven to be promising and applicable to accelerate other bioinformatics applications.

Key words: bioinformatics; RNA secondary structure prediction; minimal free energy model; hybrid acceleration

核糖核酸(RNA)是一类重要的生物分子,自20世纪80年代中期具有催化性质的RNA被发现以来,RNA所起的各种重要生物化学功能正在引起人们的日益关注和重视。RNA的空间结构是识别RNA分子的重要依据和功能研究的基础和前提。虽然实验手段是获取二级结构的最可靠方法,但是由于RNA分子难结晶而且降解快,采用实验方法测定分子结构很困难,并且代价高昂。近年来,采用计算机和数学模型预测RNA二级结构的方法被广泛采用,成为RNA结构和功能研究领域的热点问题。

目前有两类最重要的RNA二级结构预测方法,一种是基于序列比较分析模型,它将未知序列与已知结构的RNA家族的共变模型进行比对,从而判断序列是否属于该RNA家族,并进一步地得到该序列的二级结构。上述方法适用于样本数量较多的情况,并且基于已有的知识。在没有先验知识的情况下,最可靠的是采用最小自由能模型,它采用一套实验测定的自由能参数作为打分标准,从能量的角度衡量RNA分子结构的主要特

* **收稿日期**:2013-01-29
基金项目:国家自然科学基金资助项目(61202127);国家863计划资助项目(2008AA01A201)
作者简介:夏飞(1980—),男,湖南常德人,博士,助理研究员,E-mail:xcyphoenix@nudt.edu.cn

征。由于碱基配对可以使 RNA 分子的能量降低,结构趋于稳定,因此最小自由能算法(minimum free energy)认为在一定的温度下,RNA 分子通过构象调整达成某种热力学平衡,使自由能最小,从而形成最稳定的状态,此时的二级结构即被认为是 RNA 的真实二级结构[1]。

最小自由能算法由 M. Zuker 于 1981 年提出[2],又被称为 Zuker 算法。其计算对象不是简单的碱基配对数量,而是子序列的自由能。算法基本思想是:基于 RNA 二级结构中各种子结构的自由能具有独立性和可加性假设,采用试验方法测定的各种子结构自由能参数表,将序列所有可能形成的子结构的自由能相加,整条 RNA 序列的最小自由能等于所有可能的子结构能量之和的最小值。

Zuker 算法是目前最好的针对单条序列的结构预测算法[3],尤其是针对小分子 RNA 的结构预测取得了很好的预测结果[4]。因此该算法被广泛应用于 Mfold[5]、RNAfold[6]、ViennaRNA[7] 等著名 RNA 二级结构预测软件。Zuker 算法的空间复杂度为 $O(n^2)$,时间复杂度为 $O(n^4)$[1],n 为 RNA 序列的长度。根据实验观测数据,通常会在限制内部环长度的情况下对原始 Zuker 算法进行优化,将时间复杂度降低为 $O(n^3)$[8],但是计算开销仍然很大。随着单序列长度的增加和 RNA 数据库中序列数量的急剧膨胀,现有的软件预测方法已不能满足研究需求。

目前主要存在三类针对 Zuker 算法的并行处理方式。一种采用集中共享多处理器或者 cluster 结构,采用粗粒度的任务划分策略,将计算任务分配到多个处理器上并行执行。Tan[9] 等人在具有 16 个 Opteron 处理器的集群上获得了 8 倍的加速比。在共享存储并行计算机上,Tan[10] 等人在具有 32 个处理器的系统 DAWNING 4000 上取得了 19 倍的加速比;Mathuriya[11] 等人在具有 32 个计算核心的 IBM P5570 服务器上对 GTfold 软件进行加速取得了 19.8 倍的加速效果。为了增强单节点的处理能力,基于多核处理器平台的 Zuker 算法并行化解决方案逐渐引起研究者的关注。文献[12]在 Intel Q6600 四核处理器上对 RNAfold 程序实现并行,获得了 3.88 倍的加速效果。文献[13]基于 IBM Cyclops64 多核处理器采用了粒度更细的线程级任务划分策略,当序列规模为 2048bps 时,与通用微处理器相比,使用 64 个 core 可获得大约 30 倍的加速效果。

近年来,随着 FPGA 器件(Field Programmable Gate Array)和 GPU (Graphics Processing Unit) 计算性能的飞速提升,采用基于 FPGA 或 GPU 的硬件加速器实现对生物信息学应用的加速计算逐渐成为研究热点。在 FPGA 平台上,文献[14]针对 Zuker 算法内部环计算部分提出了一种细粒度并行处理方案,但并未考虑整个算法的设计与实现,也没有考虑任务划分和数据访存调度的问题,并且只给出了模拟结果。Dou[15] 和 Jacob[16] 等人先后对 Zuker 算法实现了细粒度并行优化,并成功对完整的 Zuker 算法实现了硬件加速,与通用 CPU 相比分别取得了一到两个数量级的加速效果。在 GPU 平台上,G. Rizk[17] 对基于 Zuker 算法设计的应用程序 Unafold 进行了加速,与单核 Xeon CPU 相比可获得 33 倍的加速比性能提升。

由于在大规模并行计算平台上对 Zuker 算法实现加速面临使用和维护成本较高的问题,近年来随着 GPU 在高性能计算中的应用越来越广泛[18],CPU 和 GPU 的异构协同并行计算在高性能计算领域已经被证明是有效的。

随着 CPU 和 GPU 计算能力的不断提升,如何发挥两者的计算资源优势成为一个值得研究的问题。在现有的 CPU + GPU 的异构系统中,一般由 CPU 负责程序流程控制,GPU 则负责对核心计算部分实现加速,CPU 的计算性能没有得到充分的利用和开发。

1　研究背景

1.1　Zuker 算法简介

算法的理论依据是碱基配对可以使 RNA 分子的能量降低,结构更加稳定,所以真实的 RNA 二级结构的热力学自由能应当趋于最小[3]。算法将 RNA 结构分解为 5 种基本子结构,如图 1 所示,并且构成二级结构的基本子结构的自由能具有可加性和相对独立性,整条序列的最小自由能等于所有可能的子结构能量之和的最小值[19]。

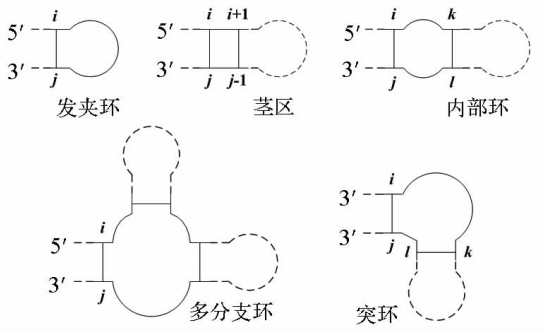

图 1　RNA 二级结构中的基本子结构

Fig. 1　Basic substructure in RNA sequence

Zuker 算法的输入是单条 RNA 序列,输出是碱基配对结果。假设 $R = r_1 r_2 r_3 \ldots r_n$ 是长度为 n 的

RNA 序列，r_i 表示序列中的碱基，下标 i 分别表示碱基 r_i 在序列中的序号，同时也是元素在自由能矩阵中的坐标。计算过程是从 r_1 到 r_n，每一步都通过比较所有可能子结构的最小能量值得到目前为止子序列 $r_{1\cdots}r_j$ 的最小自由能（$1 \leqslant j \leqslant n$）。计算过程涉及向量 $\boldsymbol{W}$，4 个上三角矩阵：$\boldsymbol{V}$，$\boldsymbol{VBI}$，$\boldsymbol{VM}$，$\boldsymbol{WM}$ 以及 3 个自由能函数 $e\boldsymbol{S}(i,j)$，$e\boldsymbol{H}(i,j)$，$e\boldsymbol{L}(i,j,k,l)$，迭代公式如下[8]：

$$\begin{cases} W(j) = \min\{W(j-1),\ \min[V(i,j)+W(i-1)]\},\ (1 \leqslant i < j,\ j > 0) & (1)\\ V(i,j) = \min[eH(i,j),\ eS(i,j)+V(i+1,j-1),\ VBI(i,j),\ VM(i,j)],\ (i<j) & (2)\\ VBI(i,j) = \min[eL(i,j,k,l)+V(k,l)],\ (i<k<l<j) & (3)\\ VM(i,j) = \min[a+WM(i+1,h-1)+WM(h,j-1)],\ (i+1<h \leqslant j-1) & (4)\\ WM(i,j) = \min\{V(i,j)+b,\ \min[WM(i,h-1)+WM(h,j)]\},\ (i<h \leqslant j) & (5) \end{cases}$$

向量 $\boldsymbol{W}$ 中的元素 $W(j)$ 存储序列前 j 个碱基片断 $r_{1\cdots}r_j$ 的最小自由能。当 $j=n$ 时，能量计算过程结束，此时 $W(n)$ 的值便是整条序列的最小自由能，根据 Zuker 算法的假设，它对应着序列最可能的二级结构。从公式(1)可以看出，$W(j)$ 的计算依赖于其自身左侧元素，$W(1)$ 到 $W(j-1)$ 以及 V 矩阵的第 j 列元素 $V(*,j)$。

$V(i,j)$ 表示当 r_ir_j 构成碱基对时，子序列 $r_{i\cdots}r_j$ 具有的最小自由能。根据公式(2)，它是 5 种可能的子结构对应自由能的最小值，其中 $eH(i,j)$ 表示由碱基对 r_ir_j 封口的发夹环的能量，$eS(i,j)$ 表示由碱基对 r_ir_j 和 $r_{i+1}r_{j-1}$ 构成的茎区的自由能。

$VBI(i,j)$ 表示当碱基对 r_ir_j 和 r_kr_l 封闭起一个突环或内部环时，子序列 $r_{i\cdots}r_j$ 的最小自由能。根据公式(3)，$VBI(i,j)$ 的计算依赖于 $V(i,j)$ 的左下区域以及通过查表获得的内部环（包括突环）自由能函数值，即 $eL(i,j,k,l)$。

$VM(i,j)$ 表示当碱基对 r_ir_j 封闭起一个多分支环时，子序列 $r_{i\cdots}r_j$ 具有的最小自由能。根据公式(4)，$VM(i,j)$ 的计算依赖于元素 $WM(i,j)$ 的下一行 $WM(i+1,*)$ 和前一列 $WM(*,j-1)$。$WM(i,j)$ 是为了简化 $VM(i,j)$ 的计算，提高运算速度而引入的辅助矩阵，它表示多分支环上的一段子序列 $r_{i\cdots}r_j$ 对应的最小自由能。$WM(i,j)$ 的计算依赖于自身左侧同行，下方同列的元素以及 $V(i,j)$。

实验结果表明，公式(1)～(5)计算占到整个算法执行时间的 99% 以上。因此加速自由能函数的计算是提升性能的关键。

1.2 多核 CPU 体系结构简介

在摩尔定律的引领下，以多核处理器为代表的先进体系结构逐渐取代单核处理器成为通用微处理器的主要发展方向。图 2 是典型的四核 CPU 的体系结构。每个 core 每次可以运行一个线程，内部包含一组寄存器用来保存线程状态，大量的功能单元被用于实现算术或逻辑运算。

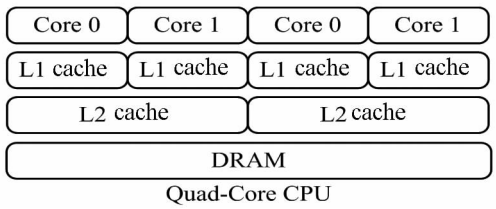

图 2 四核 CPU 体系结构

Fig. 2 Quad-core CPU architecture

多核处理器的设计理念是通过多个相对简单的计算核心的并行工作来提高处理器的整体性能。为了让多个核并行工作，需要对原有的单线程序进行并行设计优化。程序员需要考虑多核平台的结构，更多地掌握处理器结构的细节知识才能实现有效并行。由于程序特征的复杂性导致并行的复杂度和效果存在显著差异，多核的使用不一定能够获得显著的性能提升，如何让多个计算核心高效地协同工作是并行程序设计的关键和挑战。

目前面向多核 CPU 的并行编程方法，如 POSIX Thread（pThread）library[20]，OpenMP library[21]，Intel threading building blocks[22]有效简化了多核平台上并发软件的开发，使得多核 CPU 逐渐成为强有力的计算平台。此外还可以通过 SIMD 向量处理指令开发数据并行性。例如，Intel 的 SSE（Streaming SIMD Extensions）指令集包含 70 个新指令和 8 个 128 位寄存器[23]。SSE2 中增加了新的操作类型，包括最大化和最小化操作。每个 128 位寄存器能够被划分来执行 4 个 32 位整数、单精度浮点数或者 8 个 16 位短整型数以及 16 个字节数据的并行操作。

1.3 GPU 体系结构简介

GPU 最初被开发作为专用的加速部件来满足计算机图形和计算机游戏的性能需求。随着

CPU 多核技术的发展和图形处理要求的不断提高,GPU 朝着更加通用、并行性更高的多核处理器体系结构方向发展,其计算能力和存储带宽得到不断的提高。面向 GPU 通用计算的编程模型或方法如 NVIDIA's CUDA[24], BrookGPU[25], ATI's Stream SDK[26], SH[27] 和 OpenCL[28] 的开发和不断完善极大地简化了 GPU 应用的开发。

图 3 描述了 NVIDIA GPU 的体系结构。该 GPU 包含一个可扩展的多线程处理单元阵列,每个处理单元称为流式多处理器(SM)。每个 SM 包含 8 个标量处理器 SP (Scalar Processor) 核,用来执行实际指令。每个 SM 可以执行独立的计算,但其中的所有 SP 核心必须同步执行计算指令。这种单指令多线程(SIMT)是 GPU 的基本计算模式。线程是 GPU 的基本运算单元,多个线程构成线程块 block,所有的线程块构成网格 grid。这些结构化的线程集在代码的核心上发射来处理设备内存中的数据。同一个块中的线程共享片上存储资源,并通过同步点来实现协同计算。

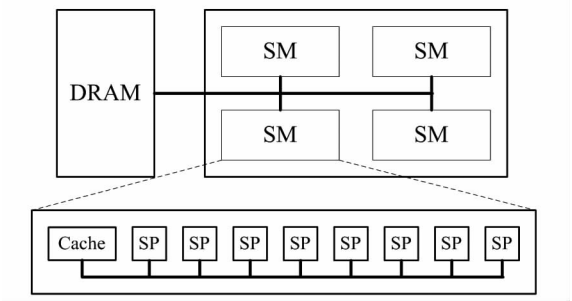

图 3　NVIDIA GPU 体系结构

Fig. 3　NVIDIA GPU architecture

GPU 采用了层次化的存储体系结构。存储速度最快的是容量很小的共享存储器和寄存器。寄存器由编译器进行分配,而共享存储器则由程序员分配。常量、纹理和全局存储则在片外 DRAM 中分配。纹理和常量存储是只读的并被缓存起来。全局存储的速度最慢,一次访问需要上百个时钟周期。

GPU 编程的重中之重是在数据并行体系结构上寻找最佳的解决问题的方法,同时充分利用针对特定 GPU 体系结构的优化方法。在现有的 GPU 通用计算应用编程接口[24-27]中,NVIDIA 公司推出的 CUDA(Computer Unified Device Architecture)通用并行计算架构和基于 C 语言扩展的通用并行编程模型是目前 GPU 应用开发的主流工具[29]。在 CUDA 架构下,一个 block 块内的线程以 warp 为单位分组创建、管理、调度和执行,warp 组内包含 32 个 ID 连续的线程。当 warp 组中所有的线程都执行相同的指令路径时,计算效率才能达到最高。

2　CPU-GPU 混合计算系统

2.1　系统结构

CPU-GPU 混合计算系统由多个 CPU 和 GPU 加速器组成。图 4 显示了由两个 CPU 和两个 GPU 构成的混合系统结构。CPU 和 GPU 之间通过 I/O hub chipset 实现通信,CPU 和 GPU 则分别通过 Quick Path Inter connect link (QPI)和 PCIE 接口与 I/O Hub 连接,CPU 之间通过 QPI link 连接。CPU 和 GPU 都有独立的存储器。每个 CPU 包含 4 个计算核心,每个核拥有独立的 L1 cache,每两个核共享一个 L2 cache。CPU 在系统中主要负责 Zuker 算法应用软件的启动、动态任务划分、GPU 计算启动和结果回收、能量矩阵的回溯处理等。GPU 主要负责多条序列的能量矩阵的并行计算。GPU 计算期间,CPU 也负责对部分自由能矩阵的并行计算。

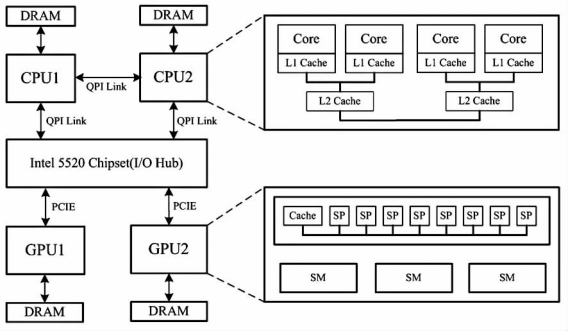

图 4　CPU-GPU 混合加速系统结构

Fig. 4　CPU-GPU hybrid accelerating system

系统启动后,CPU 首先载入 RNA 序列,执行内存分配、序列对齐、任务划分等预处理。任务分配完成后,CPU 和 GPU 并行对自由能矩阵进行计算。当 GPU 完成能量矩阵计算任务后,CPU 从 GPU 设备读回结果。最后,由 CPU 执行回溯并将 RNA 二级结构信息反馈给用户。

2.2　系统任务分配和调度策略

本文提出的混合加速系统充分挖掘 CPU 和 GPU 的性能潜力来获取更高的计算性能。其中的关键因素是 CPU 和 GPU 之间的高效协同和平衡的任务划分。假设给定 N 条序列,其编号为 1,2,3,…,N,任务划分的主要过程是获取序列编号的边界数 B,编号小于等于 B 的序列交由 CPU 处理,而序列编号大于 B 的序列则交由 GPU 处理。为了实现两类不同处理单元间的负载平衡,需要预先对 CPU 和 GPU 平台对 RNA 序列的处理能力

进行评估。

图 5 是本文提出的混合任务分配与执行策略。算法的输入为 N 条长度为 L 的序列和单条序列在 CPU 和 GPU 上预估的平均执行时间。输出为序列的自由能参数和二级结构信息。算法执行过程分为三步:1)基于单条序列在 CPU 和 GPU 上的平均执行时间计算 GPU 对 CPU 的加速比;2)计算边界分配值 B;3)启动 CPU 和 GPU 计算。在第三步中,当序列[B: N]发送到 GPU 设备后,多核 CPU 立即启动对序列[1: B]的多线程并行处理。在 Intel 编译器的调度下,每个线程负责单条序列能量矩阵的计算。CPU 在 GPU 的计算完成后,读回 GPU 生成的能量矩阵,并执行后续操作,生成并输出每条序列的最低能量和结构信息。

Algorithm: Task allocation and execution scheme between CPU and GPU

Input Parameters:
N: RNA sequence number; L: RNA sequence length;
T_{1CPU}: Avg. execution time for single RNA sequence on CPU;
T_{1GPU}: Avg. execution time for single RNA sequence on GPU;

Output Parameters:
E[1; N]: minimal energy of the N sequences;
S[1; N]: secondary structure information of N sequences;

Variables Define:
K: performance ratio GPU vs. CPU;
B: the boundary allocation value for RNA sequences;

Step 1:
$K = T_{1GPU}/T_{1CPU}$ // get performance ratio of GPU vs. CPU

Step 2:
B = N/(K+1) // compute the boundary value for allocation

Step 3:
S_{31}: Send the sequences in [B; N] to device memory
S_{32}: Start the GPU computing
S_{33}: *Multi-thread parallel processing over multi-sequence on CPU*
for sequence s in [1;B] do
compute energy matrices W, V, WM;
end for
S_{34}: if (GPU computing done)
Receive matrices W, V, WM matrix data form GPU device memory for sequences in [B; N];
Do S_{35};
else go to S_{34};
S_{35}: *backtrack according to the W, V and WM matrices*
for sequence s in [1; N] do
compute E[s];
compute S[s];
end for

图 5 混合任务分配和执行策略

Fig. 5 Mixed task allocation and execution strategy

假设 CPU 和 GPU 上单条序列的平均执行时间分别为 T_{1CPU} 和 T_{1GPU}。当 CPU 和 GPU 单元的执行几乎完全重叠时,整个序列集合的能量矩阵填充时间最小。因此可得公式 $T_{1CPU} \times B = T_{1GPU} \times (N - B)$,从而计算得到 $B = N \cdot \alpha$,其中 α 称为分配比例,其表达式为 $\alpha = \dfrac{1}{K+1}$,式中的 K 表示 GPU 对 CPU 的加速比,$K = T_{1CPU}/T_{1GPU}$。

3 性能优化策略

3.1 多核 CPU 实现

针对多核 CPU 的性能优化策略包括编译优化,SSE 和多线程优化三个方面。首先,通过设置 O2 优化选项提升原有软件的性能。第二,使用 SSE 指令来加速矩阵的计算。矩阵元素 $VM(i,j)$ 依赖于 ***WM*** 的行 $WM(i, *)$ 和列 $WM(*, j)$。$VM(i, j)$ 的计算分两步:①***WM*** 矩阵第 i 行和第 j 列对应元素相加;②在相加结果中选取最小值作为元素 $VM(i, j)$ 的值。每次分别读取 ***WM*** 矩阵第 i 行和第 j 列中的 4 个元素存储在 128 位的寄存器中,然后使用 SSE 加法指令实现多个元素的并行计算。

第三种优化方法是基于 OpenMP 实现多序列间的粗粒度并行处理。通过支持 OpenMP 库的 Intel 编译器,将不同序列的能量矩阵计算过程映射至不同的处理线程。每个线程对分配到本地的自由能矩阵元素实现串行计算,而多个线程独立访问各自存储空间,实现高性能的粗粒度并行。

3.2 GPU 实现

GPU 上的优化策略包括多层次并行、存储优化、线程调度优化和 Tiling 方法。

3.2.1 多层次并行

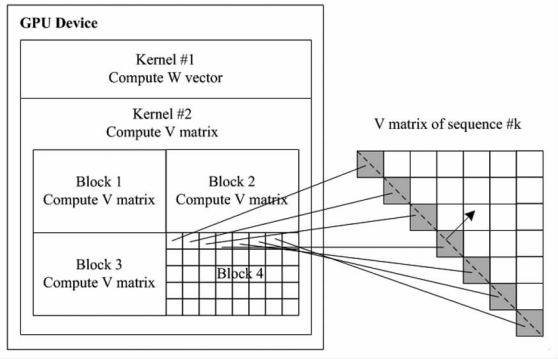

图 6 GPU 端的层次化并行策略

Fig. 6 Hierarchical parallel strategy on GPU

多层次并行策略包含三个并行层次,其中第一层为内核级并行。在图 6 中,GPU 设备包含#1 和#2 两个内核,用于实现 W 向量和 V 矩阵的并行计算。第二层为序列级并行,即在同一个内核中多条序列被不同的块处理。当序列的长度较小时,一个块能够装下多条序列。在图 6 中,内核#2 就包含多个块,每个块负责一条或多条序列 V 矩阵的计算。第三层为单条序列自由能矩阵中对角元素计算的并行。在图 6 中,V 矩阵对角线上的元素被映射到同一个块的不同线程上实现并行计算。

3.2.2 存储优化

GPU 设备端采用的存储优化策略有:①采用数据类型转换、直接赋值、冗余数据精简等方法减少 GPU 设备的参数存储需求。②使用数组存储转置后的矩阵列数据,提高对矩阵列元素的存储访问性能。③将序列数据和能量矩阵分别存放到

shared memory 中和 global memory 中,减少平均访存开销。④将能量矩阵中的部分数据存入 shared memory 中实现数据重用。⑤使用 shared memory 和 register 缓存计算过程产生的中间结果,提高访存带宽和访存速度。

3.2.3　线程调度

通过将对角线元素的计算映射到同一个 warp 组内的线程上实现细粒度并行。由于碱基 i 和 j 配对情况的不同会导致程序执行不同的分支,因此,同一个 warp 组中的线程的执行路径可能不同。为了避免该问题,本文在提出的混合系统中,将具有相同执行路径的矩阵元素映射到同一个 warp 组内具有连续 ID 的线程上,从而有效提升算法的并行效率。

3.2.4　Tiling 方法

采用分块方法加速单序列自由能矩阵的计算。图 7 展示了被分成小块的 ***VM*** 矩阵的计算过程。***VM*** 矩阵对角线上的元素在不同的线程中并行计算。计算的方向是从主对角线出发向右上方移动,直到到达右上角。阴影部分表示已算完的 ***VM*** 和 ***WM*** 矩阵元素(在同一个矩阵中显示)。分块中的元素 $VM(i,j)$ 依赖于 ***WM*** 矩阵的第 i 行和第 j 列,被分成了图 7 中标号为①、②、③的三部分。在已有的 ***WM*** 矩阵数据中,$VM(i,j)$ 计算所需的标有序号②的中间段数据被提前计算并被存入共享存储区中实现快速访问。例如,图 7 中位于 T2 对角线上的数据块中的 ***VM*** 元素计算所需的带有序号②标识的中间段数据都被提前计算并缓存。当计算对角线从 T1 上移至 T2 时,计算所需的阴影区域内的元素可从共享存储区直接读取,从而加速对角线 T2 上的元素的计算过程。

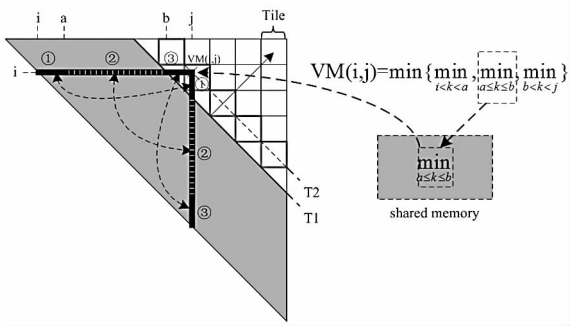

图 7　矩阵分块策略

Fig. 7　Matrix partitioning strategy

4　实验和结果

4.1　实验环境

系统中的主机配置为 Intel Xeon E5620 Quad core 2.4GHz CPU,24GB 内存,ASUS Z8PE-D12 主板,Intel 5520 芯片组, Windows 7 操作系统和 Microsoft Visual Studio 2010 开发环境。GPU 设备型号为 NVIDIA GTX 280,1.5GB 显存,CUDA 4.0 集成开发环境。Zuker 算法应用程序来自奥地利 Vienna 大学生物信息中心开发的结构预测软件包 ViennaRNA - 1.8.4[30]。

4.2　CPU 性能

选取随机生成的 A,B,C,D 四组 RNA 序列进行测试,每组包含 1024 条长度相同的 RNA 序列。每条序列的平均执行时间通过计算每组所有序列的能量矩阵来获取,性能调优过程逐步采用 O2 编译优化、SSE 和多线程处理的方法。实验结果如表 1 所示。

表 1　CPU 优化效果对比(时间单位:s)

Tab. 1　Optimization results of different performance tuning methods on CPU

Opt. methods	A(L = 68)		B(L = 120)		C(L = 154)		D(L = 221)	
	Time	Sp.	Time	Sp.	Time	Sp.	Time	Sp.
None	7.449	1	35.282	1	66.270	1	155.954	1
M1	3.930	1.90	18.631	1.89	34.795	1.90	82.266	1.90
M1 + M2	4.052	1.84	19.164	1.84	34.901	1.90	80.177	1.95
M1 + M2 + M3	1.081	6.89	4.737	7.45	8.424	7.87	19.059	8.18

Note: Opt methods: M1—02, M2—SSE2, M3—Multi-thread

在单个 Xeon E5620 核上采用 O2 优化方法可以获得平均 1.90 的加速比。由于序列长度较短,SSE2 方法对于前三组序列没有明显效果,原因是对短序列引入的控制开销要比原来的 VM 规约操作的开销要大。而采用了多线程处理后,每组序列的计算性能都达到了最高。

4.3　GPU 性能

表 2 显示了在 GPU 上逐步采用不同性能调优方法时每组序列的平均执行时间。A,B,C,D 四组序列包含的序列数量均为 1200 条,采用随机

方法生成。以 GPU 上的多层次优化方法的性能作为比对基准。从表 2 可以看出,随着优化措施的加强,每组序列的计算性能都得到了显著提升。采用线程调度和分块方法后,每组序列都获得了最佳平均执行时间。根据参考文献[17],一条长度为 120bps 的 RNA 序列在 GTX 280 上执行二级结构预测的平均执行时间为 0.473ms,而采用本文提出的 GPU 优化方法的平均执行时间为 0.293ms,可获得 1.62 倍的加速比,明显优于相关工作。

表 2 三种 GPU 优化策略加速效果对比(时间单位:s,加速比(Sp.))

Tab. 2 Optimization results of different performance tuning methods on GPU

Opt. methods	A($L=68$)		B($L=120$)		C($L=154$)		D($L=221$)	
	Time	Sp.	Time	Sp.	Time	Sp.	Time	Sp.
M1	0.452	1	2.402	1	3.530	1	20.067	1
M1 + M2	0.199	2.27	0.855	2.81	1.561	2.26	8.876	2.26
Ml + M2 + M3	0.068	6.65	0.293	8.20	0.754	4.68	3.270	6.14

M1:hierarchy parallelism, M2:memory optimization, M3:thread scheduling and tiling

表 3 不同序列组的任务划分比例估计(时间单位:s)

Tab. 3 The estimated task allocation ratio for different sequence groups

	A($L=68$)	B($L=120$)	C($L=154$)	D($L=221$)
Avg. CPU exe. time	1.081	4.737	8.424	19.059
Avg. GPU exe time	0.068	0.293	0.754	3.270
Speedup GPU VS. CPU	15.90	16.17	11.17	5.83
Estimated allocation ratio	5.92%	5.83%	8.22%	14.64%

表 4 四组序列在不同平台上的执行时间和加速比(时间单位:s)

Tab. 4 Execution time and speedup on different platforms for four sequence groups

Opt. methods	A($L=68$)		B($L=120$)		C($L=154$)		D($L=221$)	
	Time	Sp.	Time	Sp.	Time	Sp.	Time	Sp.
Opt. quad. CPU	20.248	1	90.370	1	163.612	1	380.824	1
Opt. GPU	1.360	14.89	5.860	15.42	15.080	10.85	65.400	5.83
Hybird system	1.280	15.82	5.673	15.93	13.870	11.80	56.386	6.75

4.4 混合系统性能

基于 CPU-GPU 异构混合系统,本文选取了 4 组随机生成的 RNA 序列进行测试。每组包括 20000 条长度相同的序列。4 组序列的长度分别为 68、120、154、221bps。表 1 和表 2 显示了每组序列中单条序列的平均执行时间。对每组序列,GPU 相对于 CPU 的加速比通过 CPU 和 GPU 上的平均执行时间求得。任务分配比例通过 GPU 对 CPU 的加速比计算得到。其结果如表 3 所示。

1)分配比例评估

图 8 展示了当任务分配比例以 2% 的步长从 2% 增加到 30% 时混合系统执行时间的测试结果。从图中可以看出,四组序列的最短执行时间对应的最佳分配比例分别约等于 4%,4%,6% 和 14%。A,B,C 三组序列的估计分配比例稍微偏

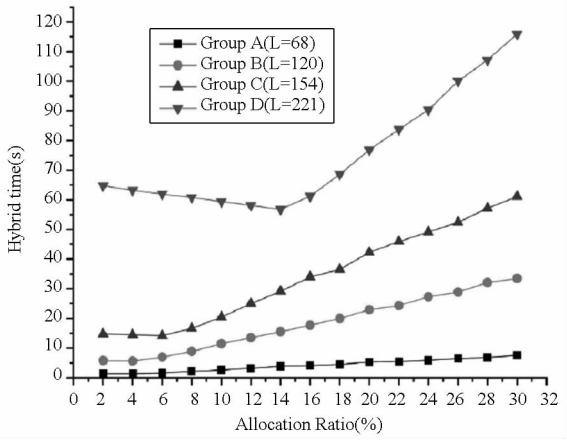

图 8 四组序列在不同分配比例下的执行时间

Fig. 8 System Exe. time for four sequence groups

离实际最佳值,主要原因是 CPU 和 GPU 的平均执行时间没有得到准确估计。这个误差对于短序列是可以接受的,其实际执行时间和最佳分配比例的执行时间十分接近。对于平均长度较长的 D 组序列,其最佳分配比例和估计值非常接近。由此可以推断基于 CPU 和 GPU 平均执行时间的分配方法是可行的。在 D 组序列中,超过 14% 的计算任务分配到 CPU 处理,与单纯的 GPU 加速方法相比,充分利用了闲置的多核 CPU 的计算能力。

2)混合系统加速比

我们测试了四组序列在多核 CPU、GPU 以及 CPU-GPU 混合系统上的执行时间。为了和单纯基于 CPU 或者 GPU 的并行实现进行比较,系统的执行时间中均包含了任务分配和数据通信开销。表 4 列出了四组序列在不同平台上的执行时间和加速比。对于 B 组数据而言,和优化的四核 CPU 并行实现相比,采用 GPU 和混合系统并行方案可分别获得 15.42 和 15.93 倍的加速效果,和运行在单核 Xeon E5620 CPU 上的串行程序相比可获得超过 50 倍的性能提升。对 D 组数据而言,和 CPU 并行实现相比,采用 GPU 和混合系统并行方案可分别获得 5.83 和 6.75 倍的加速效果。也就是说,和最优的单纯 GPU 加速方案相比,采用混合系统加速方案可以获得 16% 的性能提升。

4.5　讨论

由表 4 的结果可以看出,对 B 组序列采用混合加速方法获得的加速比最高。由此可以推断混合系统的计算性能对于该组数据能够得到较好的发挥。随着序列长度的增加,采用混合系统的加速比在减小。这主要是因为 Zuker 算法的不同部分有着不同的计算复杂性和 GPU 实现效率。对于 D 组乃至更长的序列,受限于 Zuker 算法复杂的数据相关和紧耦合的同步关系,GPU 的计算效率在降低。从最后一栏可以看到,D 组中 GPU 对 CPU 的加速比为 5.83,比其它各组都低。由此推断,尽管混合加速系统的主要思想是在 GPU 执行的基础上充分发挥多核 CPU 的计算性能,但混合系统的性能仍然依赖于 GPU 的计算能力。对 D 组序列而言,超过 85% 的计算任务仍然由 GPU 承担。

充分发挥混合系统计算能力的关键是实现 CPU 和 GPU 之间负载平衡的任务划分。对于输入序列,首先统计 CPU 和 GPU 上序列的平均执行时间,进而计算 GPU 对 CPU 的加速比。然后用加速比值计算任务分配的边界值 B,从而指导 CPU 和 GPU 之间的任务分配。根据边界值 B,将输入数据分成两部分,分别发送到 CPU 和 GPU 平台实现并行处理。

尽管本文仅基于由一个 CPU 和一个 GPU 加速器构成的原型系统评估了混合加速方法的性能,但是对于由多个 CPU 和 GPU 构成的硬件平台,文章提出的方法依然适用。在具有多个 CPU 和 GPU 运算部件的硬件平台上,仍然需要对单个部件的计算能力进行预估,从而指导任务分配。主要区别在于对由多个运算部件构成的计算平台,需要确定一组任务分配边界值。例如,对于由两个 CPU 和两个 GPU 设备构成的混合计算系统,则需要确定 3 个边界值来指导 4 个运算部件的任务量划分。

5　结论

Zuker 算法作为 RNA 二级结构预测的经典方法广泛应用于目前的科学研究。本文在深入研究 CPU 和 GPU 体系结构和程序优化方法的基础上,提出了一种新的 CPU + GPU 的混合加速方法实现 Zuker 算法的并行计算。针对 CPU 和 GPU 平台的计算特点和性能差异,通过合理的任务分配策略,实现了二者之间的并行协作计算和处理单元间的负载平衡;针对 CPU 和 GPU 平台的不同硬件特性,采用多种策略对 Zuker 算法在 CPU 和 GPU 上的实现分别进行了优化,提高了混合加速系统的计算性能。

实验结果表明,本文提出的 CPU + GPU 的混合式加速系统能够同时有效发挥 CPU 和 GPU 的计算能力,CPU 处理单元在混合系统中承担了 14% 以上的计算任务,与单纯的多核 CPU 并行方案相比,采用混合并行加速方法可获得 15.93 的全局加速比;与单纯的 GPU 加速方案相比,可获得 16% 的性能提升。实验结果证明了采用 CPU + GPU 的混合式加速方法的可行性,并且该方法可用于对生物序列分析领域的其他应用实现并行和加速计算。

参考文献(References)

[1] Zuker M, Stiegler P. Optimal computer folding of large RNA sequence using thermodynamics and auxiliary information[J]. Nucleic Acids Research, 1981, 9(1): 133 - 148.

[2] Durbin R, et al. Biological sequence analysis: probabilistic models of proteins and nucleic acids[M]. Cambridge U. K.: Cambridge University Press, 1998.

[3] Gardner P P, Giegerich R. A comprehensive comparison of comparative RNA structure prediction approaches[J]. BMC

Bioinformatics, 2004 (5): 140 – 157.
[4] Mount D W. Bioinformatics: sequence and genome analysis [M]. New York: Cold Spring Harbor Laboratory Press, 2001.
[5] Zuker M. Mfold web server for nucleic acid folding and hybridization prediction[J]. Nucleic Acids Research, 2003, 31(13): 3406 – 3415.
[6] Hofacker I, et al. Fast folding and comparison of RNA secondary structures[J]. Monatshefte Fur Chemie/Chemical Monthly, 1994, 125(2): 167 – 188.
[7] Mathews D H, Sabina J, Zucker M, et al. Expanded sequence dependence of thermodynamic parameters provides robust prediction of RNA secondary structure [J]. Journal of Molecular Biology, 288(5): 911 – 940.
[8] Lyngso R B, Zuker M. Fast evaluation of internal loops in RNA secondary structure prediction[J]. Bioinformatics, 1999, 15(6): 440 – 445.
[9] Tan G M, Feng S Z, Sun N H. Locality and parallelism optimization for dynamic programming algorithm in bioinformatics [C]//Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, Florida, USA, ACM 2006: No. 78.
[10] Tan G M, Feng S Z, Sun N H. An optimized and efficiently parallelized dynamic programming for RNA secondary structure prediction[J]. Journal of Software, 2006, 17: 1501 – 1509.
[11] Mathuriya A, Bader D, Heitsch C, et al. GTfold: a scalable multicore code for RNA secondary structure prediction[C]// Proceedings of the 2009 ACM Symposium on Applied Computing, ACM 2009: 981 – 988.
[12] Wu G, Wang M, Dou Y, et al. Exploiting fine-grained pipeline parallelism for wavefront computations on multicore platforms [C]//2009 International Conference on Parallel Processing Workshops, IEEE 2009: 402 – 408.
[13] Tan G M, Sun N H, Gao G R. A parallel dynamic programming algorithm on a multi-core architecture[C]//19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2007: 135 – 144.
[14] Tan G, Xu L, Feng S, et al. An experimental study of optimizing bioinformatics applications[C]//Proceedings of the IEEE International Parallel and Distributed Processing Symposium. Phodes Island, Greece, 2006: 25 – 29.
[15] Dou Y, Xia F, Zhou X, et al. Fine-grained parallel application specific computing for RNA secondary structure prediction on FPGA [C]//Proceedings of the IEEE International Conference on Computer Design (ICCD), 2008: 240 – 247.
[16] Jacob A, Buhler J, Chamberlain R. Rapid RNA folding: Analysis and acceleration of the Zuker recurrence [C]// Proceedings of 18th IEEE Annual International Symposium on Field- Programmable Custom Computing Machines (FCCM), IEEE 2010: 87 – 94.
[17] Rizk G, Lavenier D. GPU accelerated RNA folding algorithm [C]//Computational Science – ICCS 2009: 1004 – 1013.
[18] TOP500 Supercomputing Sites. 2010, [EB/OL] http://www.top500.org.
[19] Lyngso R B, Zuker M, Pedersen C N S. Internal loops in RNA secondary structure prediction[C]//Proceedings of the third Annual International Conference on Computational Molecular Biology, 1999.
[20] Barney B. POSIX Threads Programming. 2010, [EB/OL] https://computing.llnl.gov/tutorials/pthreads/.
[21] Menon R. Parallel programming in OpenMP[M]. Morgan Kaufmann, 2001.
[22] Reinders J. Intel threading building blocks [M]. O'Reilly, 2007.
[23] Intel. Intel 64 and IA – 32 architectures software developer manual[EB/OL]. [2012 – 11 – 10]. http://www.intel.com/products/processor/manuals.
[24] Corporation N. CUDA 2.0 programming guide[EB/OL]. [2012 – 10 – 05]. http://www.nvidia.com/.
[25] Buck I, Foley T, Horn D, et al. Brook for GPUs: stream computing on graphics hardware[J]. ACM Transactions on Graph 2004, 23: 777 – 786.
[26] ATI: Technical overview ATI stream computing[EB/OL]. [2012 – 10 – 05]. http://developer.amd.com/sdks/amdappsdk/pages/default.aspx
[27] McCool M, Du Toit S. Metaprogramming GPUs with Sh AK [M]. Peters Wellesley, 2004.
[28] Group K. OpenCL[EB/OL]. [2012 – 10 – 05] http://www.khronos.org/opencl/.
[29] NVIDIA Corporation. CUDA Programming Guide 3.2[EB/OL]. [2012 – 10 – 05]. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.
[30] Gruber A, Lorenz R, Bernhart S, et al. The vienna RNA websuite[J]. Nucleic Acids Research, 2008, 36 (suppl 2): W70.