Kỷ yếu Hội nghị Quốc gia lần thứ VIII về Nghiên cứu cơ bản và ứng dụng Công nghệ thông tin (FAIR); Hà Nội, ngày 9-10/7/2015
DOI: 10.15625/vap.2015.000214

ỨNG DỤNG GIẢI THUẬT SONG SONG TRÊN HỆ THỐNG CPU-GPU CHO BÀI TOÁN TÌM KIẾM MOTIF

Nguyễn Tấn An[1], Trần Văn Lăng[2], Nguyễn Gia Khoa[3]

[1] Học viện Công nghệ Bưu chính Viễn thông.
[2] Viện Cơ học và Tin học ứng dụng, Viện Hàn lâm Khoa học và Công nghệ Việt Nam.
[3] Trường Cao đẳng Kinh tế - Kỹ thuật Thành phố Hồ Chí Minh.

nguyenan6391@gmail.com, langtv@vast.ac.vn, nguyengiakhoa@yahoo.com

TÓM TẮT *- Bài toán tìm kiếm motif trên trình tự ADN là một bài toán phực tạp và mất nhiều thời gian để giải quyết. Đã có rất nhiều thuật toán được đề xuất và giải quyết tốt cho bài toán này, nhưng về vấn đề thời gian vẫn là thách thức lớn. Bên cạnh đó, hiện nay công nghệ tính toán song song trên GPU rất phổ biến, vì vậy thực hiện song song hóa bài toán tìm kiếm motif trên GPU sẽ là giải pháp nhằm cải thiện vấn đề thời gian. CUDA và OpenCL là 2 công nghệ lập trình trên GPU phổ biến nhất hiện nay. Trong bài báo này, chúng tôi tiến hành song song hóa thuật toán Pattern Branching tìm motif trên GPU bằng hai công nghệ CUDA và OpenCL nhằm đánh giá so sánh hiệu suất giữa chúng.*

Từ khóa *- motif ADN, CUDA, OpenCL, song song, sinh tin học.*

I. GIỚI THIỆU

Trong vài thập kỷ qua, với sự phát triển mạnh mẽ của công nghệ sinh học, một khối lượng lớn dữ liệu sinh học phân tử (gene, protein, genome) đã được thu thập, lưu trữ và chia sẻ tại các ngân hàng dữ liệu thế giới như: GenBank, EMBL, DDBJ, PDB… Trong đó bài toán tìm trình tự motif nhằm tìm ra các đoạn trình tự nucleotide hay amino acid phổ biến trong các dãy trình tự DNA, RNA hay protein, bản thân motif đại diện cho chức năng, cấu trúc hoặc thành viên trong họ, từ đó phân tích chúng góp phần xác định tính năng sinh học. Việc đi tìm những mẫu trình tự tương tự hoặc so với những mẫu có sẵn để tìm ra tính năng sinh học của gene giúp ích rất nhiều cho việc nghiên cứu và đưa ra phương pháp chữa trị và ngăn ngừa các căn bệnh nan y.

Đã có nhiều thuật toán được thiết kế để giải quyết bài toán này, mỗi thuật toán có những ưu khuyết điểm riêng của nó. Nhìn chung, dựa vào phương pháp tiếp cận các thuật toán tìm kiếm motif được phân làm hai nhóm:

Nhóm phương pháp tiếp cận dựa trên không gian mẫu như Consensus, Gibbs sampling, MEME, các phương pháp này thực hiện kiểm tra dựa trên một tập hợp mẫu cho trước từ đó xác định các vị trí xuất hiện trực tiếp của motif, mặc dù phương pháp tiếp cận dựa trên không gian mẫu có nhiều sự lựa chọn mô hình thống kê phù hợp (Stormo and Hartzell 1989; Lawrence et al. 1993; Bailey and Elkan 1994; Hughes et al. 2000; Workman and Stormo 2000; Thijs et al. 2001) tuy nhiên nó vẫn phụ thuộc nhiều vào mô hình lựa chọn đầu tiên và yêu cầu phải chạy nhiều lần để tăng xác suất tốt hơn cho thuật toán.

Nhóm phương pháp thứ hai tiếp cận dựa trên chuỗi mẫu như thuật toán Teiresias, MITRA, bằng cách sử dụng một chuỗi trung tâm (trong ADN được biểu diễn bằng 4 kí tự alphabet) giả định như là một motif, phương pháp chuỗi mẫu thực hiện tìm kiếm vét cạn trên tất cả tập 4^l motif ứng viên cho một motif với chiều dài l và đảm bảo rằng motif tối ưu được tìm thấy. Tuy nhiên các phương pháp này đòi hỏi rất nhiều thời gian và không gian tính toán.

GPGPU (General-purpose computing on Graphics processing units – tính toán chung trên bộ xử lý đồ họa) đã hoàn thành nhiệm vụ tính toán mà ban đầu được thực hiện bởi CPU, với bộ xử lý đồ họa được thiết kế để phục vụ công việc đồ họa đã thực hiện được các phép tính toán thông thường không liên quan đến xử lý đồ họa. Do cơ chế song song cao của các GPU hiện đại và tinh giản lập trình, một bộ xử lý dòng (stream processor) có thể sử dụng để xử lý các công việc khác đồ họa, chẳng hạn như giải phương trình vi phân. Đặc biệt, với mô hình SIMD (Single Instruction Multiple Data – Đơn dòng lệnh đa dữ liệu), quá trình tổ chức và chuyển đổi dữ liệu sẽ tốn ít thời gian hơn giúp hiệu suất trên GPGPU là tốt hơn nhiều so với các chương trình trên CPU truyền thống. Đây là công nghệ mới trong những năm gần đây mặc dù GPU được trình bày trong nhiều năm, lập trình với các ngôn ngữ như C vẫn rất khó, các lập trình viên cảm thấy khó khăn trong việc dịch các vấn đề toán học vào trong đồ họa. Năm 2006, khi NVIDIA phát hành CUDA giúp cho việc lập trình dễ dàng hơn nhiều, GPGPU trở nên phổ biến.

Sự phát hành CUDA của NVIDIA đã làm nên các phần mềm GPU và các hệ thống phần cứng tính toán dữ liệu song song trên thiết bị. CUDA không cần dùng giao diện lập trình ứng dụng đồ họa (API-application programming interfaces) và có thể sử dụng một cách dễ dàng để xử lý, sử dụng ngôn ngữ C phát triển, do đó không cần phải học quá nhiều cú pháp mới. Kiến trúc CUDA gồm phần cứng và phần mềm: phần cứng hỗ trợ công nghệ CUDA (từ dòng G80 trở về sau) và phần mềm bao gồm các trình điều khiển có liên quan, trình biên dịch nvcc,… Cả phần cứng và phần mềm phải đáp ứng các yêu cầu để chạy công nghệ CUDA.

OpenCL (Open Computing Language) là một framework mới cho lập trình trên platform không đồng nhất. Platform đó có thể bao gồm CPU, GPU và các thành phần khác của bộ vi xử lý giúp hỗ trợ tính toán. OpenCL được xây dựng với ngôn ngữ dựa trên C99 cho hàm nhân (hàm chạy trên một thiết bị OpenCL) và một API dùng để định

nghĩa và điều khiển platform. OpenCL cũng tương tự như 2 tiêu chuẩn mở khác là Open Graphics Language (OpenGL) và Open Algorithms Language (OpenAL). Hai tiêu chuẩn được sử dụng trong đồ họa 3D và âm thanh máy tính. OpenCL là mở rộng năng lực của GPU do đó nó có thể dùng cho các công việc khác đồ họa. OpenCL được quản lí bởi tổ chức phi lợi nhuận là Khronos Group. Đầu tiên nó được phát triển và mang nhãn hiệu của Apple và sau đó được hoàn thiện bằng cách hợp tác với đội ngũ kỹ thuật từ AMD, IDM, Intel và NVIDIA. Sau đó, Apple đã đệ trình dự thảo này qua cho Khronos Group. Bây giờ GPU của cả NVIDIA và AMD đều hỗ trợ OpenCL.

Thuật toán tìm motif đầu tiên được song song hóa dựa trên thuật toán MEME. Triển khai của CUDA-MEME trên các GPU được trình bài trong [1], thuật toán MEME đã được song song hóa và chạy trên GPU cài đặt bằng công nghệ CUDA. Dựa trên phân tích hiệu suất của chúng, tốc độ bình quân của CUDA-MEME tăng 20.5.

Thuật toán mCUDA-MEME [2] là một mở rộng của CUDA-MEME. CUDA-MEME chạy trên một máy tính đơn với GPU, trong khi mCUDA-MEME chạy trên một cụm máy tính với GPU. Các CUDA-MEME sẽ trao đổi các gói tin với nhau thông qua giao thức giữa các GPU.

Trong việc thực hiện CUDA-Gibbs sampling, người ta chọn giá trị $S_{updating}$ theo yêu cầu thay vì lựa chọn ngẫu nhiên. Loại bỏ các yếu tố ngẫu nhiên làm giảm cơ hội tìm được motif tốt. Tuy nhiên, bù lại đó số lượng mẫu thử tăng lên. Các chiến lược tối ưu hóa thuật toán bao gồm: cải thiện cách tính điểm PSSM và sắp xếp lại các tiến trình nhằm tối thiểu SIMD (*Single Instruction Multiple Data*) phân kỳ. Bằng chiến lược này thực hiện trên CUDA, tốc độ thuật toán cải thiện tốt hơn 10 lần [3].

Chương trình song song [4] trình bày một cách tiếp cận song song hiệu quả khác cho bài toán tìm kiếm motif trên GPU sử dụng phương pháp BitBased. Thuật toán BitBased ban đầu được đề xuất cho CPU [5], nó đã giải quyết bài toán tìm kiếm motif với l = 21 và số đột biến k = 8 trong 1,1 giờ.

Bài báo này thực hiện song song hóa thuật toán Pattern Branching, thuật toán được đề xuất bởi Pevzner và Sze [6], thuật toán được cho là triển vọng so với tìm kiếm motif bằng phương pháp ngẫu nhiên hay phương pháp chọn mẫu bằng xác suất, thuật toán được cài đặt bằng CUDA và OpenCL chạy trên GPU nhằm đánh giá hiệu suất giữa 2 công nghệ này.

Các phần của bài báo như sau: phần 2 giới thiệu về bài toán tìm motif, phần 3 trình bày về thuật toán pattern branching, công nghệ CUDA và OpenCL, đồng thời trình bày cách thức song song hóa thuật toán pattern branching trên GPU. Kết quả đạt được sẽ được trình bày trong phần 4 và phần cuối cùng là kết luận.

II. BÀI TOÁN TÌM MOTIF

A. Định nghĩa motif

Motif là một trình tự nucleotide hay amino-acid có (hoặc có thể có) chức năng sinh học nào đó.

Bài toán tìm motif trên ADN có thể được định nghĩa như sau: Cho một tập trình tự ADN, tìm các chuỗi giống nhau hay gần giống nhau (trường hợp một vài nucleotide bị đột biến) xuất hiện trên tất cả các trình tự.

Ví dụ cho 5 chuỗi trình tự như sau, trường hợp không có đột biến:

1. cctgatagacgctatctggctatcc**acgtacgt**aggtcctctgtgcgaatctatgcgtttccaaccat
2. agtactggtgtacatttgat**acgtacgt**acaccggcaacctgaaacaaacgctcagaaccagaag
3. aa**acgtacgt**gcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaattt
4. agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatctt**acgtacgt**ataca
5. ctgttatacaacgcgtcatggcggggtatgcgttttggtcgtcgtacgctcgatcgtta**acgtacgt**c

Chúng ta tìm được motif trong tập 5 trình tự trên là: **acgtacgt.**

Trường hợp với 2 đột biến, xem tập 5 trình tự như sau:

1. cctgatagacgctatctggctatcc**aGgtAcgt**aggtcctctgtgcgaatctatgcgtttccaaccat
2. agtactggtgtacatttgat**CcgTacgt**acaccggcaacctgaaacaaacgctcagaaccagaag
3. aa**acGtaCgt**gcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaattt
4. agcctccgatgtaagtcatagctgtaactattacctgccacccctattacatctt**acGTacgt**ataca
5. ctgttatacaacgcgtcatggcggggtatgcgttttggtcgtcgtacgctcgatcgtta**acgTacCt**c

Chúng ta tìm được các motif như các trình tự nucleotide được in đậm ở trên.

Vấn đề tìm kiếm motif có những sự khó khăn như sau:

– Không biết được các mẫu của motif

– Không biết được vị trí xuất hiện của motif trong trình tự.

– Các motif có thể khác nhau trong các trình tự.

B. Khoảng cách hamming và láng giềng của l-mer.

- Khoảng cách hamming

Một đoạn trình tự nucleotide ngắn với chiều là l kí hiệu là l-mer, khoảng cách hamming của 2 đoạn l-mer được tính dựa trên số kí tự không khớp của 2 đoạn l-mer đó.

Ví dụ: d(ACGT, ATGT) = 1, ACGT khác ATGT tại nucleotide thứ 2.

Gọi S là tập các trình tự ADN gồm có n trình tự, khoảng cách hamming giữa một đoạn l-mer A với tập trình tự S được tính như sau:

Bước 1: Tính khoảng cách hamming của đoạn l-mer A đó đến các trình tự Si

$$d(A, Si) = \min\{ d(A, P) | P \in Si\} \qquad (i = 1, \dots, n)$$

Trong đó: P là biểu thị cho một đoạn l-mer trong Si.

Bước 2: Tính tổng khoảng cách của đoạn l-mer A đến mẫu S.

$$d(A, S) = \sum_{i=1}^{n} d(A, Si)$$

Khoảng cách hamming của đoạn l-mer đến tập trình tự S chính là điểm số để đánh giá đoạn l-mer nhằm tìm ra đoạn motif tốt nhất.

- Láng giềng của l-mer

Láng giềng của một đoạn l-mer là tập l-mer B được định nghĩa bởi công thức sau $B = D_{=1}(A)$, trong đó D là khoảng cách của các l-mer trong tập B đến l-mer A bằng 1.

Ví dụ: Cho l-mer A (TTG), tập láng giềng của A là:

ATG **C**TG **G**TG T**A**G T**C**G T**G**G TT**A** TT**C** TT**T**

- Láng giềng tốt nhất: M là là láng giềng tốt nhất của A, $M \in D_{=1}(A)$ và $d(M, S)$ là nhỏ nhất.

III. PHƯƠNG PHÁP

A. Giải thuật Pattern Branching

Đặt một M là một motif – được xem như là một pattern có chiều dài l, và đặt A0 là một thể hiện của M trong mẫu với chính xác k đột biến, gọi d là khoảng cách Hamming, ta có d(M,A0) = k, tức khoảng cách Hamming của pattern M đến chuỗi A0 bằng k, ta nói $M \in D_{=k}A0$, D được định nghĩa như một tập hợp các chuỗi với khoảng cách chính xác từ k đến A0. Thuật toán Pattern Braching sẽ định nghĩa một hàm BestNeighbor nhằm tìm đường đi từ chuỗi A0 đến láng giềng tốt nhất A1 của nó trong tập $D_{=1}A0$, theo cách đó thì một nucleotide bị thay đổi. Rồi từ láng giềng tốt nhất A1 đó ta tiến hành xét đến pattern láng giềng tốt nhất A2 trong tập $D_{=1}A1$. Cứ như vậy ta tiến hành tìm láng giềng tốt cho nhất cho A0 khi đến k đột biến cho phép.

$$A0 \rightarrow A1 \rightarrow A2 \rightarrow \dots \rightarrow Ak.$$

Thuật toán:

Đầu vào:

Tập hợp các chuỗi trình tự S = {S1,S2,…,Sn}.

Chiều dài của motif là l.

Số lượng đột biến là k.

Đầu ra:

Motif có chiều dài l với k đột biến.

Thuật toán:

1. PatternBranching(S, l, k)
2. Motif ← arbitrary motif pattern
3. For each l-mer A0 in S

```
4.  For j ← 0 to k
5.  {
6.      If d(Aj, S) < d(Motif, S)
7.      Motif ← Aj
8.      Aj+1 ← BestNeighbor(Aj)
9.      OutputMotif
10. }
```

B. CUDA và OpenCL

CUDA và OpenCL cung cấp 2 giao diện khác nhau cho lập trình GPU NVIDIA. Cả 2 đều gọi một đoạn mã thực thi trên GPU thông qua hàm nhân kernel. Tùy theo mỗi ngôn ngữ có sự khởi tạo, khai báo khác nhau. Sau đây là một số định nghĩa tương đồng của 2 ngôn ngữ.

Sự tương đồng giữa các đơn vị tính toán và không gian bộ nhớ:

Bảng 1. Sự tương đồng giữa các đơn vị tính toán và không gian bộ nhớ

CUDA	OpenCL
thread	**work-item**
block	**work-group**
global memory	
constant memory	
shared memory	**local memory**
local memory	**private memory**

Cú pháp hàm nhân:

Bảng 2. Sự tương đồng các cú pháp hàm nhân

CUDA	OpenCL
__global__ (hàm)	**__kernel**
__device__ (hàm)	không cần khai báo
__constant__ (biến)	**__constant**
__device__ (biến)	**__global**
__shared__ (biến)	**__local**
_syncthreads()	**barrier()**

Sự khác nhau về một số API khởi tạo nền tảng, bộ nhớ:

Bảng 3. Một số API khởi tạo nền tảng, bộ nhớ

CUDA	OpenCL
Không có hàm tương đồng	**clCreateCommandQueue()**
cuModuleLoad()	**clCreateProgramWithSource() or clCreateProgramWithBinary()**
Không có hàm tương đồng	**clBuildProgram()**
cuModuleGetFunction()	**clCreateKernel()**
cuMemAlloc()	**clCreateBuffer()**
cuMemcpyHosttoDevice()	**clEnqueueWriteBuffer()**
cuMemcpyDevicetoHost()	**clEnqueueReadBuffer()**
cuParamSeti()	**clSetKernelArg()**

cuParamSetSize()	Không có hàm tương đồng; hàm này là một phần trong clSetKernelArg()
cuLaunchGrid()	**clEnqueueNDRangeKernel()**
cuMemFree()	**clReleaseMemObj()**

Về cấu trúc của một chương trình:

Cấu trúc của một chương trình CUDA đơn giản hơn so với một chương trình OpenCL, do CUDA chỉ sử dụng trên nền tảng các GPU NVIDIA hỗ trợ CUDA, trong khi OpenCL lại sử dụng đa nền tảng chính vì vậy quá trình khai báo truy vấn thiết bị OpenCL tương đối phức tạp hơn là CUDA chỉ cần chọn GPU để truy vấn. Dưới đây là cấu trúc chương trình giữa CUDA và OpenCL:

Bảng 4. Cấu trúc chương trình giữa CUDA và OpenCL

CUDA	OpenCL
1. Chọn GPU. 2. Cấp phát bộ nhớ trên GPU. 3. Chuyển dữ liệu từ CPU vào GPU. 4. Gọi hàm kernel để tính toán. 5. Chuyển dữ liệu từ GPU sang CPU. 6. Lặp lại bước 3 đến 5 nếu cần. 7. Giải phóng bộ nhớ.	1. Truy vấn các thiết bị OpenCL. 2. Tạo context liên kết OpenCL và thiết bị. 3. Tạo program chứa mã nguồn. 4. Định nghĩa kernel để gọi thực thi program. 5. Tạo memory objects trên host hay device. 6. Sao chép dữ liệu vào device. 7. Cung cấp các đối số cho kernel. 8. Đưa kernel vào hàng đợi để thực thi. 9. Sao chép kết quả từ device về host. 10. Giải phóng vùng nhớ.

OpenCL là một chuẩn mở dùng cho cả GPU, CPU và các thiết bị khác còn CUDA chỉ dành cho lập trình song song trên GPU NVIDIA, chính vì vậy OpenCL không thể hoàn toàn cung cấp đầy đủ các tính năng có thể sử dụng trên GPU NVIDIA như CUDA, chẳng hạn: mã OpenCL trong hàm nhân không hỗ trợ hàm printf, điều này gây khó khăn rất nhiều trong việc kiểm tra kết quả tính toán, OpenCL 1.2 trở lên mới hỗ trợ bộ nhớ texture 1D trên GPU NVIDIA, bộ nhớ texture là bộ nhớ toàn cục chỉ đọc có khả năng truy xuất nhanh hơn so với bộ nhớ toàn cục global, trong khi CUDA đã hỗ trợ bộ nhớ texture 1D, 2D và 3D.

C. Song song giải thuật Pattern Branching trên GPU

- Tổ chức dữ liệu

Ở một hệ thống đa xử lý sẽ bị giới hạn bởi số lượng các thanh ghi, từ đó nếu một luồng sử dụng một số lượng lớn thanh ghi thì số biến trong hoạt động trong cùng một khối sẽ ít hơn, giảm hiệu năng của GPU. Để cải thiện hiệu năng GPU thì cần giảm số lượng thanh ghi sử dụng càng nhiều càng tốt. Với mỗi trình tự đầu vào có chiều dài L có L-l+1 đoạn l-mer, nếu đoạn l-mer được biểu diễn bằng cách sử dụng mảng kí tự thì mỗi đoạn l-mer sẽ cần đến l byte bộ nhớ, như vậy không gian bộ nhớ để lưu các đoạn l-mer này rất tốn kém, thay vào đó ta sử dụng mảng số nguyên integer để biểu diễn 1 l-mer, như vậy chỉ mất 4 byte hoặc 8 byte cho 1 l-mer. Ví dụ , 4-mer CGGA có thể biểu diễn bằng cách sử dụng 1 số nguyên, con số này sẽ được biểu diễn dưới dạng nhị phân là 01101000 (trong đó mỗi nucldeotide sẽ được biểu diễn bằng 2 bit, A là 00, C là 01, G là 10 và T là 11), với cách biểu diễn này với đoạn l-mer có $l \leq 16$ chỉ cần sử dụng 4 byte số nguyên, $l \leq 32$ chỉ cần dùng 8 byte số nguyên và với cách biểu diễn nhị phân này ta có thể tính khoảng cách hamming giữa các đoạn l-mer dễ dàng bằng các phép tính bit. Do đó có thể chuyển mảng kí tự của chuỗi trình tự đầu vào thành mảng số nguyên, mỗi số nguyên tại vị trí i sẽ biểu diễn đoạn l-mer tại vị trí i của chuỗi trình tự đầu vào. Bằng cách chuyển sang mảng số nguyên như vậy, một luồng trên GPU chỉ cần đọc một con số nguyên thay vì đọc l byte kí tự. Bằng cách đó chương trình sẽ giảm số lượng thanh ghi trong quá trình đọc ghi dữ liệu đầu vào. Các đoạn l-mer đầu vào là dữ liệu không thay đổi trong quá trình xử lý thuật toán ta sử dụng bộ nhớ toàn cục của GPU để lưu chúng.

- Song song thuật toán

Với cách tổ chức dữ liệu như trên, với mỗi đoạn l-mer của một trình tự, sẽ đưa vào thực hiện thuật toán Pattern Branching trên một luồng, các luồng sẽ thực hiện đồng thời sẽ là giải pháp song song hóa thuật toán Pattern Branching.

Quá trình xử lý của thuật toán trên hệ thống CPU kết hợp GPU như sau:

Input: Tập trình tự S có n trình tự, mỗi trình tự có chiều dài L, chiều dài motif l, số đột biến cho phép k.

Output: Motif M.

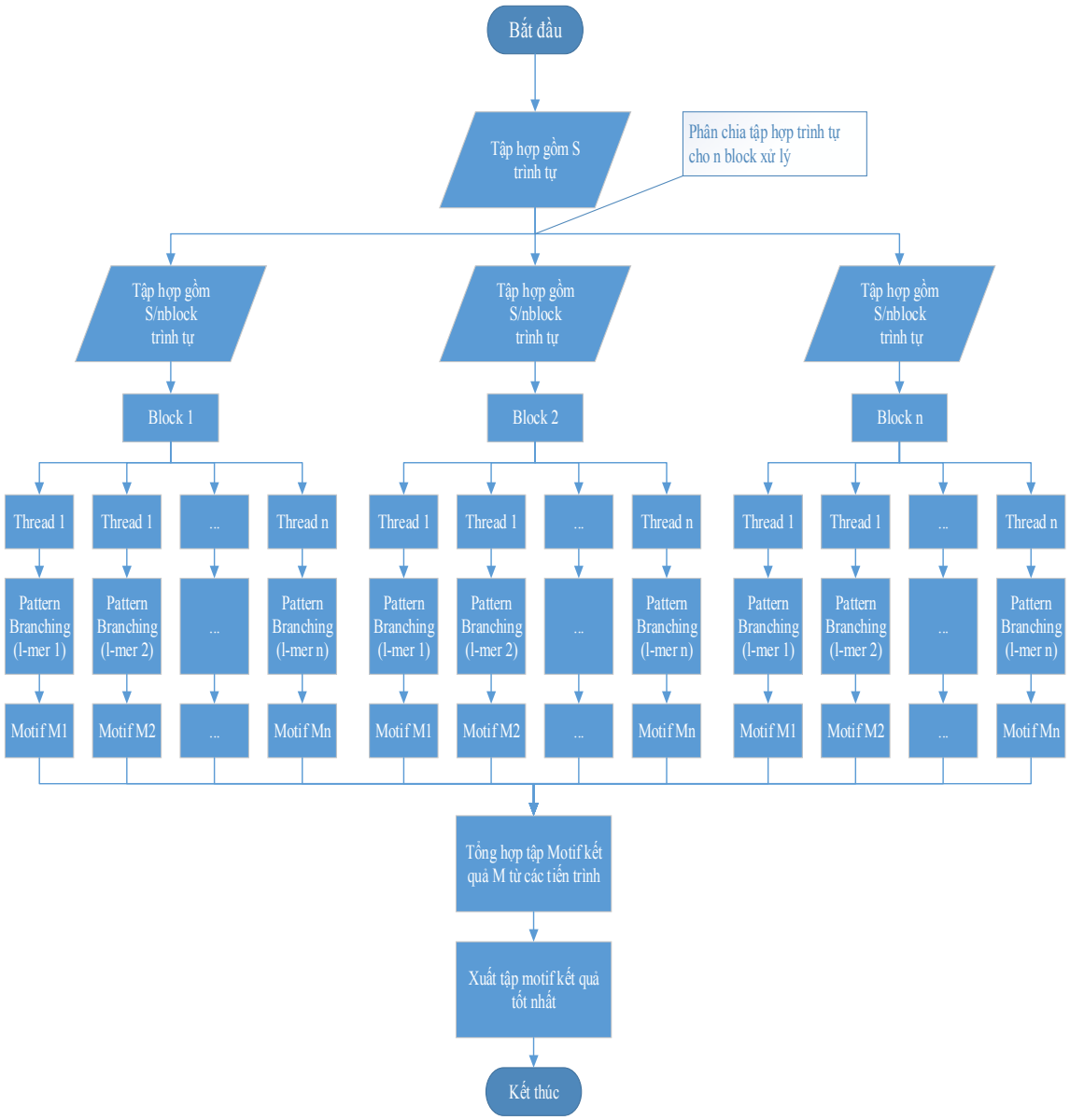
CPU: Khởi tạo tập tất cả các l-mer từ tập dữ liệu trình tự đầu vào cho GPU.

GPU:Với mỗi thread $\in \{0, 1, \dots, (L - l + 1)\}$ của các block $\in \{0, \dots, (n - 1)\}$

- Thực hiện thuật toán Pattern Banching với mỗi l-mer tương ứng với từng thread.
- Xuất motif M tương ứng với mỗi thread.

CPU: Tổng hợp tập các motif M từ các thread. Tìm motif M tốt nhất.

Sơ đồ song song hóa thuật toán Pattern Branching:



Hình 1. Sơ đồ song song hóa thuật toán Pattern Branching

IV. KẾT QUẢ THỰC NGHIỆM

Thuật toán Pattern Branching được cài đặc xử lý song song và tuần tự trên hệ thống CPU intel Core i5-4210U 1.7Ghz, 4GB Ram và GPU NVIDIA Geforce 820M. Chương trình xử lý song song được lập trình trên hai bộ công cụ là CUDA và OpenCL để tiến hành so sánh hiệu suất của chúng. Chương trình sẽ sử dụng bộ dữ liệu từ [7] để tiến hành đánh giá kết quả của thuật toán Pattern Branching xử lý song song và xử lý tuần tự. Các tập trình tự thử nghiệm gồm 20 trình tự với chiều dài mỗi trình tự là 600, đây là số lượng trình tự và chiều dài được đặt ra trong bài toán thách thức tìm motif [8], chiều dài và số đột biến của các motif cần tìm khác nhau trong từng trường hợp. Sau đây là bảng kết quả và thời gian thực thi thuật toán (không tính thời gian khởi tạo).

Bảng 5. Bảng kết quả so sánh thời gian thực thi

Tập dữ liệu	(l,k)	Motif kết quả	Thời gian thực hiện thuật toán		
			Tuần tự CPU	CUDA	OpenCL
9.2.txt	(9,2)	GTTCAGCGT (1) GTTCAGCGT (2) GTTCAGCGT (3) GTTCAGCGT (4)	19.11s	5.51s	2.49s
12.3.txt	(12,3)	CCTGCCAGAAAA (1) CCTGCCAGAAAA (2) CCTGCCAGAAAA (3) CCTGCCAGAAAA (4)	18.89s	5.13s	2.69s
15.4.txt	(15,4)	ATCGAGCTTTGACAA (1) ATCGAGCTTTGACAA (2) ATCGAGCTTTGACAA (3) ATCGAGCTTTGACAA (4)	24.22s	6.88s	4.18s
17.6.txt	(17,6)	ATTAGAGCGCACATTCT (1) ATTAGAGCGCACATTCT (2) ATTAGAGCGCACATTCT (3) ATTAGAGCGCACATTCT (4)	26.62s	8.87s	4.99s
18.5.txt	(18,5)	TGTAAGAATTGTACCTTC (1) TGTAAGAATTGTACCTTC (2) TGTAAGAATTGTACCTTC (3) TGTAAGAATTGTACCTTC (4)	30.41s	10.07s	5.84s
19.6.txt	(19,6)	TACATCAGCGGTGGATGTT (1) CTACATCAGCGGTGGATGT (2) CTACATCAGCGGTGGATGT (3) CTACATCAGCGGTGGATGT (4)	30.1s	9.08s	5.46s
21.6.txt	(21,6)	GCGCGACGGACTTACGTCTTC (1) GCGCGACGGACTTACGTCTTC (2) GCGCGACGGACTTACGTCTTC (3) GCGCGACGGACTTACGTCTTC (4)	36.32s	12.39s	7.9s
23.7.txt	(23,7)	TAATCGTGCTTTGTACCCCCGGA (1) TAATCGTGCTTTGTACCCCCGGA (2) TAATCGTGCTTTGTACCCCCGGA (3) TAATCGTGCTTTGTACCCCCGGA (4)	35.76s	12.69s	8.58s
24.7.txt	(24,7)	AATTACTTTCCGATAAAGTGGATC (1) AATTACTTTCCGATAAAGTGGATC (2) AATTACTTTCCGATAAAGTGGATC (3) AATTACTTTCCGATAAAGTGGATC (4)	41.11s	14.66s	10.21s
27.8.txt	(27,8)	ACAAAATTTACACCTGGGCTGTTCGCC (1) ACAAAATTTACACCTGGGCTGTTCGCC (2) ACAAAATTTACACCTGGGCTGTTCGCC (3) ACAAAATTTACACCTGGGCTGTTCGCC (4)	52.46s	17.41s	12.76s
28.9.txt	(28,9)	TGCCCTGTGCTATCATTCATGTACAGCG (1) TGCCCTGTCCTATCATTCATGTACGGCG (2) TGCCCTGTCCTATCATTCATGTACGGCG (3) TGCCCTGTCCTATCATTCATGTACGGCG (4)	41.78s	15.18s	11.98s
29.8.txt	(29,8)	AGCAGGCCTGTGCACGGGGATGAAGTCTC (1) AGCAGGCCTGTGCACGGGGATGAAGTCTC (2) AGCAGGCCTGTGCACGGGGATGAAGTCTC (3) AGCAGGCCTGTGCACGGGGATGAAGTCTC (4)	45.33s	16.69s	13.2s
30.9.txt	(30,9)	CTGCCATTGGAAGGAGCATTCGCTGCTACT (1) CTGCCATTGGAAGGAGCATTCGCTGCTACT (2) CTGCCATTGGAAGGAGCATTCGCTGCTACT (3) CTGCCATTGGAAGGAGCATTCGCTGCTACT (4)	52.28s	19.82s	15.54s

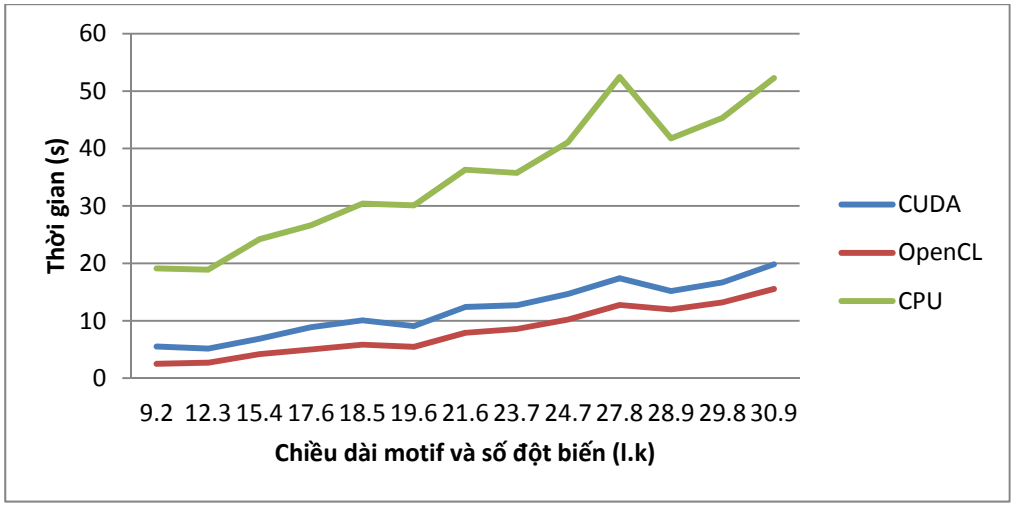
(1): Motif kết quả của tập data test.

(2): Motif kết quả của thuật toán Pattern Branching xử lý tuần tự.

(3): Motif kết quả của thuật toán Pattern Branching xử lý song song bằng CUDA.

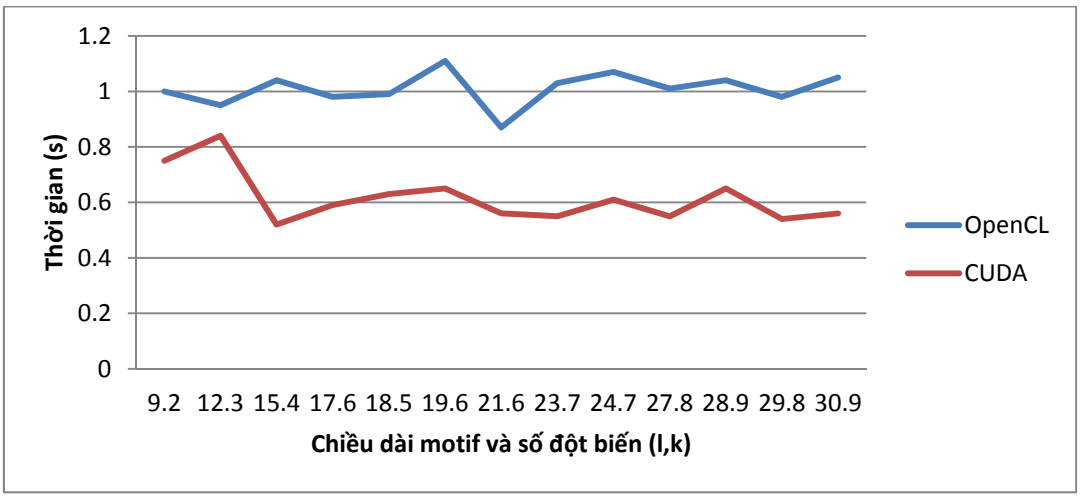
(4): Motif kết quả của thuật toán Pattern Branching xử lý song song bằng OpenCL.

Bằng cách song song hóa thuật toán Pattern Branching xử lý trên GPU, thời gian chạy của thuật toán nhanh hơn từ 2 đến 3 lần so với xử lý tuần tự trên CPU. Tuy nhiên nhìn chung tốc độ của thuật toán còn phụ thuộc vào số đột biến của tập dữ liệu đầu vào và đoạn motif, ví dụ như ở 2 tập dữ liệu kiểm nghiệm là 27.8.txt và 28.9.txt, với tập dữ liệu 27.8 tuy có chiều dài l đầu vào là 27 ngắn hơn so với tập dữ liệu 28.9 có chiều dài l là 28 nhưng thời gian lại xử lý chậm hơn, do ở tập dữ liệu 27.8 có số đoạn l-mer có chỉ số score tốt nhiều hơn nên thời gian tính toán của thuật toán chậm hơn. Ngoài ra thuật toán vẫn chưa hoàn toàn chính xác 100% so với tập dữ liệu mẫu, trường hợp tập dữ liệu 19.6.txt, điểm số của đoạn motif thuật toán tìm được là 121 so với đoạn motif kết quả của tập dữ liệu kiểm tra có điểm số là 125. Trường hợp với tập dữ liệu 28.9.txt, thuật toán tìm ra motif khác với motif mẫu tại 1 nucleotide do quá trình tìm láng giềng sẽ chọn những nucleotide có điểm số tốt nhất, tuy nhiên trong trường hợp này lại có 2 nucleotide có điểm số bằng nhau nên thuật toán chọn một trong hai loại nucleotide là láng giềng tốt nhất tiếp theo. Tuy nhiên xét về số đột biến cho phép thì đoạn motif tìm được này vẫn đúng.



Hình 2. Biểu đồ so sánh thời gian thực thi

Về mặt thời gian thực thi giữa CUDA và OpenCL, OpenCL có thời gian thực thi nhanh hơn khoảng 3,4s so với CUDA, mặc dù thời gian khởi tạo ban đầu cho các platform và bộ nhớ đệm của CUDA là nhanh hơn so với OpenCL (Hình 3), do trong OpenCL, mã của hàm nhân được viết thành một mã riêng, trên đó khai báo toán bộ các biến cũng như các định nghĩa ban đầu, vì vậy tất cả đều được chuyển vào thiết bị để thực thi, giảm thời gian truyền dữ liệu tự host vào thiết bị, riêng CUDA, trình biên dịch nvcc cho phép người lập trình viết các đoạn mã dùng trên host và GPU chung với nhau và trình biên dịch này sẽ tự tách riêng chúng khi biên dịch, chính vì vậy một số định nghĩa và biến khai báo ban đầu được viết đơn giản hơn, tuy nhiên khi gọi hàm nhân nó lại tốn nhiều thời gian để lấy dữ liệu từ host.



Hình 3. Biểu đồ so sánh thời gian khởi tạo giữa CUDA và OpenCL

V. KẾT LUẬN

Bài báo này đã trình bày cách thức song song thuật toán Pattern Branching xử lý trên GPU bằng 2 công nghệ OpenCL và CUDA. Thông qua kết quả trong bài báo, độ chính xác của thuật toán không thay đổi và với việc xử lý song song trên hệ thống GPU đã cải thiện được vấn đề thời gian hai đến ba lần so với xử lý tuần tự trên CPU, mặt khác bài báo cũng cho thấy thời gian thực thi thuật toán này của OpenCL mang lại cho thuật toán này là nhanh hơn so với CUDA.

VI. TÀI LIỆU THAM KHẢO

[1] Y. Liu, B. Schmidt, W. Liu, D. Maskell, "CUDA-MEME: Accelerating Motif Discovery in Biological Sequences Using CUDA-enabled Graphics Processing Units," *Pattern Recognition Letters 31, 2170-2177*, 2009.

[2] Yongchao Liu, Bertil S., Doulas L. M, "An ultrafast scalable many-core motif discovery algorithm for multiple GPUs.," in *10th IEEE International Workshop on High Performance Computation Biology (HiCOMB 2011)*, 2011, pp. 428-434.

[3] Jhoirene Barasi Clemente, *Finding planted (l,d)-Motifs in Parallel using RandomProjection on GPUs*. Department of Computer Science, University of the Philippines-Diliman, Mar. 2012.

[4] N. S. Dasari, Desh R., and Z. M, "Solving Planted Motif Problem on GPU," *In Proceedings of the International Workshop on GPUs and Scientic Applications (GPUScA 2010)* , 2010.

[5] N. S. Dasari, Desh R., and Z. M, "An Ecient Multicore Implementation of Planted Motif Problem," *In Proceedings of the International Conference on High Performance Computing and Simulation*, pp. 9-15, 2010.

[6] Alkes Price, Sriam Ramabhadran and Pavel A. Pevzner, "Finding subtle motifs by branching from sample strings," *Bioinformatics*, vol. 19, no. Department of Computer Science and engineering, University of California at San Diego, La Jolla, CA 92093-0114, USA, pp. ii149-55, 2003.

[7] Qiang Yu, Hongwei Huo, Yipu Zhang, Hongzhi Guo, "PairMotif. A New Pattern-Driven Algorithm for Planted (l,d) DNA Motif Search ," *Plos ONE 7(10): e48442.* , 2012.

[8] P. Pevzner and S. H. Sze, "Combinatorial Approaches to Finding Subtle Signals in DNA Sequences," *Proceeding of 8th Int. Conf. Intelligent Systems for Molecular Biology (ISMB), 26978, 2000*.

[9] Yasmeen Farouk, Tarek ElDeeb, Hossam Faheem, "Massively Parallelized DNA Motif Search on FPGA," *Bioinformatics - Trends and Methodologies*, pp. 107-120, 2011.

[10] Buhler J, Tompa M, "Finding motifs using Random projecions.," *Journal of Computational Biology 9*, pp. 225-242, 2002.

[11] Trang Hong Son, Tran Van Lang, and Le Van Vinh, "Finding the motif from DNA Sequences using Grid Computing System," *International Journal of Computer Science and Telecommunications*, vol. 4, no. 1, pp. 8-13, January 2013.

[12] NVIDIA coporation, *NVIDIA CUDA C programing guide*, 32nd ed. CA, USA: Nvidia, 2011.

[13] Matthew Scarpino, *OpenCL In Action*: Manning Pulications Co, ISBN: 9781617290176, 2012.

IMPLEMENT PARALLEL ALGORITHM ON CPU-GPU SYSTEM TO SOLVING FINDING MOTIF PROBLEM

Nguyen Tan An[1], Tran Van Lang[2], Nguyen Gia Khoa[3]

[1] Post and Telecommunications Institute of Technology,
11 Nguyen Dinh Chieu, Dist. 3, HCMC, Vietnam

[2] Institute of Applied Mechanics and Informatics, VAST,
1 Mac Dinh Chi, HCMC, Vietnam

[3] Ho Chi Minh City Technical and Economic College
215 Nguyen Van Luong, Dist. 6, HCMC, Vietnam

ABSTRACT *- The finding motif problem from DNA sequences is one of high complex problem and take a lot of time to resolve. Many algorithms were proposed for well solving the problem but excution time is still a challenge. Beside, now parallel computing on GPU is popular, so parallelizable approach to solve finding motif problem on GPU would be a solution to reduce time excution. CUDA and OpenCL are the most popular programmings on GPU. In this paper, the parallelzation Pattern Branching algorithm is implemented on GPU by OpenCL and CUDA to compare the performance of them.*