Thực hiện song song thuật toán AES bằng ngôn ngữ lập trình CUDA trên GPU NVIDIA

Đinh Tiến Thành, Phạm Mạnh Tuấn

***Tóm tắt*— Bài báo này trình bày một phương pháp cài đặt song song hóa thuật toán mật mã AES trên thiết bị xử lý đồ họa GPU (Graphic Processing Unit) của NVIDIA với ngôn ngữ lập trình CUDA. Kết quả cài đặt và đánh giá cho thấy, việc cài đặt song song hóa thuật toán AES trên thiết bị GPU đem lại hiệu suất cao và cao hơn nhiều so với việc cài đặt thuật toán này trên CPU (Central Processing Unit). Điều này đã mở ra khả năng nâng cao hiệu suất thực thi cho các thuật toán mật mã khác khi thực hiện cài đặt trên GPU.**

***Abstract*— This paper presents an installation method parallelization algorithms AES encryption on the device graphics processor GPU with NVIDIA CUDA programming language. Results install and evaluate the installation shows parallelization AES algorithm on GPUs bring high performance and much higher than with the installation of this algorithm on the CPU. This opened up the possibility of performance enhancement implementation for other cryptographic algorithms implemented on the GPU settings.**

Từ khóa— AES; CUDA; GPU.

Keywords— AES; CUDA; GPU.

I. GIỚI THIỆU CHUNG

Nhu cầu tính toán trong lĩnh vực khoa học, công nghệ ngày càng cao và đã trở thành một thách thức lớn. Từ đó, các giải pháp nhằm tăng tốc độ tính toán đã được ra đời. Tuy nhiên, hiệu năng của CPU không được gia tăng tương xứng như mức gia tăng tốc độ xung của CPU.

Việc gia tăng tốc độ xung của CPU nhanh chóng chạm ngưỡng tối đa. Trong quá trình tăng tốc độ xung của CPU, các nhà sản xuất đã gặp phải vấn đề về nhiệt độ của CPU tăng quá giới hạn. Nguyên nhân chính là do số lượng và mật độ các cổng logic trên một đơn vị diện tích trên chip ngày càng gia tăng. Nhiệt độ của CPU quá cao và các giải pháp tản nhiệt khí đã đến mức giới hạn không thể đáp ứng được khả năng làm mát khi CPU hoạt động ở xung quá cao như vậy.

Trước tình hình này, các nhà nghiên cứu vi xử lý đã chuyển hướng sang phát triển công nghệ đa lõi, với cơ chế xử lý song song trong các hệ thống tính toán nhằm tăng hiệu năng và tiết kiệm năng lượng. Việc nghiên cứu các hệ thống tính toán có hiệu năng cao và tiết kiệm năng lượng hiện nay chú trọng vào các vấn đề tăng lõi, thay đổi kiến trúc chip, kiến trúc máy tính nói chung và cả lập trình. Một trong các công nghệ xử lý song song ra đời đó là GPU. Ban đầu, việc chế tạo GPU chỉ với mục đích là tăng tốc độ xử lý đồ họa. Nhưng khi thiết bị GPU NV30 của NVIDIA ra đời, GPU bắt đầu được ứng dụng vào những công việc khác như: hỗ trợ tính toán dấu chấm động đơn và tăng khả năng tính toán lên đến hàng nghìn lệnh,... Vì vậy, đã phát sinh ra ý tưởng dùng GPU để xử lý, tính toán song song những chương trình không thuộc đồ họa ([1]). Việc ứng dụng GPU vào việc xử lý tính toán song song được giải quyết bằng công nghệ kiến trúc thiết bị hợp nhất cho tính toán (Compute Unified Device Architecture - CUDA) của NVIDIA. Với CUDA, các lập trình viên có thể phát triển các ứng dụng song song trong nhiều lĩnh vực khác nhau như: Điện toán đám mây; mã hóa; sắp xếp; tìm kiếm; mô phỏng các mô hình vật lý; chuẩn đoán y khoa; thăm dò dầu khí....

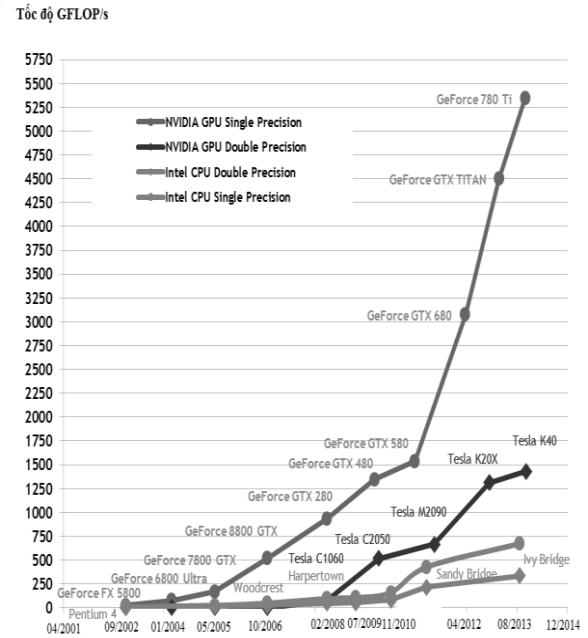
Bài báo này được trình bày với bố cục như sau: Sau Mục Giới thiệu chung, Mục II trình bày về công nghệ lập trình song song dữ liệu bằng CUDA trên GPU NVIDIA. Mục III giới thiệu về thuật toán AES. Mục IV trình bày việc thực hiện song song thuật toán AES trên GPU của NVIDIA bằng CUDA. Mục V sẽ đánh giá hiệu suất chương trình và cuối cùng là Mục Kết luận.

II. CÔNG NGHỆ LẬP TRÌNH SONG SONG DỮ LIỆU CUDA TRÊN GPU NVIDIA

A. Công nghệ GPU

Trong vài năm gần đây, năng lực tính toán và băng thông bộ nhớ của các bộ xử lý đồ họa GPU đã được tăng lên đáng kể so với CPU. Cụ thể, tính đến tháng 12/2014, GPU thế hệ Geforce GTX của

Bài báo được nhận vào ngày 05/12/2016. Bài báo được nhận xét bởi phản biện thứ nhất vào ngày 20/12/2016 và được chấp nhận đăng vào ngày 28/12/2016. Bài báo được nhận xét bởi phản biện thứ hai vào ngày 20/12/2016 và được chấp nhận đăng vào ngày 05/01/2017.



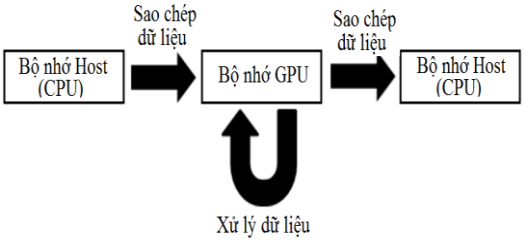
Hình 1. So sánh năng lực tính toán giữa CPU-GPU

NVIDIA đã đạt được năng lực tính toán tới 5200 GFLOPS (năng lực xử lý của các bộ vi xử lý), gấp hơn nhiều lần so với mức 750 GFLOPS của bộ xử lý bốn lõi Intel Core i7 3.2 GHz, tương tự cho băng thông bộ nhớ của GPU là 330 GB/s so với CPU chỉ có 60 GB/s tại cùng thời điểm ([1]). Hình 1 thể hiện sự tăng tốc về năng lực tính toán của các bộ xử lý đồ họa NVIDIA so với bộ vi xử lý Intel.

Tuy nhiên, sự vượt trội về hiệu năng này không đồng nghĩa với sự vượt trội về công nghệ. GPU và CPU được phát triển theo hai hướng khác biệt: trong khi công nghệ CPU cố gắng tăng tốc cho một nhiệm vụ đơn lẻ thì công nghệ GPU lại tìm cách tăng số lượng nhiệm vụ có thể thực hiện song song. Chính vì vậy, trong khi số lượng lõi tính toán trong CPU đạt đến con số 8 lõi thì số lượng lõi xử lý GPU đã đạt đến 2560 lõi vào năm 2016 (Geforce GTX 1080). Để tăng năng lực tính toán, người ta đã giảm tính linh động của các lõi xử lý trong GPU. Hiện tại, các lõi xử lý của GPU tại một thời điểm chỉ thực hiện được một tác vụ duy nhất, do vậy GPU rất thích hợp với những bài toán xử lý song song dữ liệu trong đó cùng một đoạn mã chương trình được thực thi song song cho nhiều bộ dữ liệu khác nhau. Vấn đề này phù hợp với yêu cầu thực tế khi đa số các bài toán yêu cầu năng lực tính toán lớn đều có thể quy về dạng xử lý song song dữ liệu này.

B. Ngôn ngữ lập trình CUDA

Bên cạnh việc phát triển các bộ xử lý đồ họa có năng lực tính toán lớn, các hãng sản xuất cũng



Hình 2. Sơ đồ hoạt động truyền, xử lý dữ liệu giữa CPU và GPU

quan tâm tới môi trường phát triển ứng dụng cho các bộ xử lý đồ họa này. CUDA ([1, 2]) là môi trường phát triển ứng dụng cho các bộ xử lý đồ họa của NVIDIA, bao gồm một ngôn ngữ lập trình song song dữ liệu cùng với các công cụ biên dịch, gỡ rối và giám sát thực thi cho các ứng dụng trên các bộ xử lý này. Một số đặc điểm chính của ngôn ngữ lập trình do CUDA hỗ trợ (gọi tắt là ngôn ngữ CUDA) gồm:

- Ngôn ngữ CUDA là mở rộng của ngôn ngữ C, do vậy quen thuộc với đa số người phát triển ứng dụng.
- Mã CUDA chia làm 2 phần: Phần thực thi trên CPU và phần thực thi trên GPU. Dữ liệu sẽ được sao chép từ bộ nhớ CPU sang GPU để xử lý, tại đây GPU sẽ gọi các hàm kernel, khi đó dữ liệu sẽ được xử lý song song trên hàng ngàn tiến trình riêng biệt ([1]) (Hình 2). Mỗi tiến trình có một định danh riêng dùng để xác định nhiệm vụ của tiến trình đó.

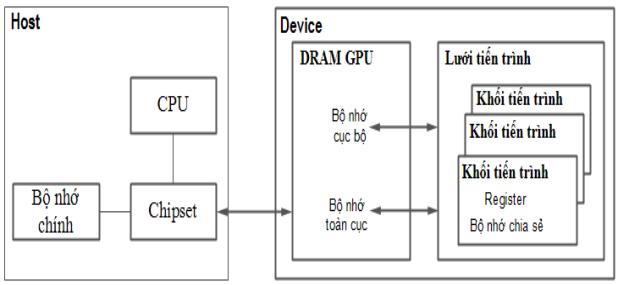
Để tránh sự phụ thuộc vào phần cứng, CUDA cho phép người lập trình tùy ý xác định số lượng tiến trình song song. Tuy nhiên, các tiến trình này cần được phân theo từng khối (block) với số lượng không quá 1024 khối (đối với các GPU đời mới với năng lực tính toán lớn hơn hoặc bằng 2.0). Cách phân khối này giúp người lập trình không cần quan tâm tới năng lực của phần cứng, đồng thời giúp việc tổ chức thực thi đạt được hiệu quả trên các GPU khác nhau.

Bộ nhớ được tổ chức phân cấp bao gồm các lớp sau ([2, 3]): bộ nhớ chính, bộ nhớ toàn cục, bộ nhớ cục bộ, bộ nhớ chia sẻ (Hình 3).

Bộ nhớ chính là vùng bộ nhớ dành cho phần mã của CPU. Chỉ có phần mã này có thể truy nhập và sửa đổi thông tin.

Bộ nhớ toàn cục là vùng bộ nhớ mà tất cả các tiến trình của GPU có thể truy nhập. Người lập trình có thể chuyển dữ liệu từ bộ nhớ chính sang

bộ nhớ này thông qua một số hàm thư viện của CUDA. Bộ nhớ này thông thường được sử dụng để lưu trữ các dữ liệu đầu vào và đầu ra cho các tiến trình song song trên GPU.



Hình 3. Sơ đồ các bộ nhớ trong lập trình CUDA

Bộ nhớ cục bộ là vùng bộ nhớ được cấp phát cho các biến cục bộ của từng tiến trình GPU và không thể truy nhập được từ các tiến trình khác.

Bộ nhớ chia sẻ là vùng bộ nhớ mà chỉ các tiến trình trong cùng một block mới có thể truy nhập được. Đây là bộ nhớ tích hợp ngay trên chip xử lý nên tốc độ truy xuất dữ liệu cao hơn rất nhiều so với bộ nhớ toàn cục. Bộ nhớ này thường được sử dụng để lưu trữ các dữ liệu chia sẻ tạm thời nhằm tăng tốc độ quá trình sử dụng bộ nhớ.

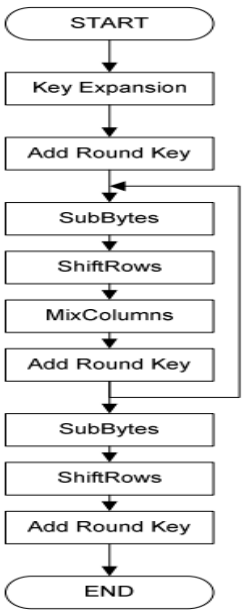
Với khả năng song song hoá dữ liệu cho rất nhiều tiến trình chạy đồng thời, GPU là giải pháp thích hợp và đầy tiềm năng trong các lĩnh vực yêu cầu năng lực tính toán lớn như: điện toán đám mây, dữ liệu lớn, trí tuệ nhân tạo, điều tra số và mật mã học. Phần chính dưới đây sẽ giới thiệu ngôn ngữ lập trình CUDA trong việc cài đặt để tăng tốc độ thực thi thuật toán mật mã AES.

III. MÔ TẢ TIÊU CHUẨN MẬT MÃ AES

A. Tiêu chuẩn mật mã AES

Tiêu chuẩn mật mã AES được công bố với ba phiên bản (AES-128, AES-192, AES-256) đều làm việc trên các khối dữ liệu 128 bit và có độ dài khóa tương ứng là 128 bit, 192 bit và 256 bit. Thuật toán AES thực hiện theo 4 pha với số vòng được qui định tương ứng với từng phiên bản (hay độ dài khóa) ([4]). Các pha thực hiện là:

Pha 1. Key Expansion - Khối mở rộng khóa nhận đầu vào là khóa có độ dài 128 bit, 192 bit, 256 bit và thực hiện quá trình mở rộng khóa để sử dụng trong các giai đoạn tiếp theo. Kích cỡ khóa được mở rộng liên quan trực tiếp đến số vòng được thực hiện trong thuật toán. Đối với khóa đầu vào là 128 bit thì thuật toán sẽ thực hiện 10 vòng lặp và kích cỡ khóa mở rộng là 352 bit. Đối với khóa đầu vào là 192, 256 bit thì số vòng lặp được thực hiện tương ứng là 12, 14 vòng và khóa mở rộng cũng sẽ có kích cỡ lần lượt là 624 và 960 bit. Trong mỗi vòng lặp, một phần của khóa mở rộng được sử dụng trong bước AddRoundKey.



Hình 4. Mô tả các bước thực hiện trong thuật toán AES

Pha 2. Initial Round - Khởi tạo vòng lặp

AddRoundKey - Tại bước này, một phần của khóa mở rộng được kết hợp với các khối trạng thái dữ liệu. Quá trình kết hợp được thực hiện bằng cách XOR từng bít của khóa con với khối dữ liệu.

Pha 3. Middle Rounds - Vòng lặp

1. Subbytes - Các byte được thay thế thông qua bảng tra S-box. Đây chính là quá trình thay thế phi tuyến của thuật toán. Hộp S-box này được tạo ra từ một phép biến đổi khả nghịch trong trường hữu hạn GF(2^8) có tính chất phi tuyến. Để chống lại các tấn công dựa trên các đặc tính đại số, hộp S-box này được tạo nên bằng cách kết hợp phép nghịch đảo với một phép biến đổi affine khả nghịch.

2. ShiftRows - Các hàng được dịch vòng một số bước nhất định. Trong chuẩn AES, hàng đầu được giữ nguyên. Mỗi byte của hàng thứ 2 được dịch vòng trái một vị trí. Tương tự, các hàng thứ 3 và 4 được dịch vòng 2 và 3 vị trí. Do vậy, mỗi cột khối đầu ra của bước này sẽ bao gồm các byte ở đủ 4 cột khối đầu vào.

3. MixColumns - Bốn byte trong từng cột được kết hợp lại theo một phép biến đổi tuyến tính khả nghịch. Mỗi khối 4 byte đầu vào sẽ cho một khối 4 byte ở đầu ra với tính chất là mỗi byte ở đầu vào đều ảnh hưởng tới cả 4 byte đầu ra. Cùng với bước ShiftRows, MixColumns đã tạo ra tính chất khuếch tán cho thuật toán. Mỗi cột được xem

như một đa thức trong trường hữu hạn và được nhân với đa thức $c(x) = 3x^3 + x^2 + x + 2$ (modulo $x^4 + 1$). Vì thế, bước này có thể được xem là phép nhân ma trận trong trường hữu hạn.

4. AddRoundKey - Tương tự như AddRoundKey trong pha 2.

Pha 4. Final Round – Kết thúc vòng lặp.

B. Một số kết quả nghiên cứu về cài đặt AES có liên quan

Những bộ xử lý đồ họa thương mại ban đầu không đem lại hiệu quả cho công việc thực thi thuật toán mã hóa vì khả năng lập trình và hỗ trợ tính toán dấu chấm động kém. Công nghệ lập trình đa dụng CUDA của NVIDA và SKD của AMD ra đời, đã tạo điều kiện để nhiều công trình mật mã được thực hiện trên các nền tảng này. Việc cài đặt thuật toán mật mã AES trên GPU có thể kể đến những công trình nghiên cứu công bố gần đây như [3, 5, 6]. Trong [3], với phiên bản AES có chiều dài khóa 256 bit, nhóm tác giả đã công bố tốc độ thực hiện mã hóa trên GPU tăng tối đa là 14,5 lần so với CPU đối với dữ liệu đầu vào có dung lượng là 68 MB.

IV. SONG SONG HÓA CÀI ĐẶT AES TRÊN GPU VỚI CUDA

A. Khả năng song song hóa cài đặt các phép biến đổi trong AES

Theo mô tả thì thuật toán AES mã hoá khối dữ liệu đầu vào chủ yếu dựa vào các phép biến đổi như: SubBytes, MixColumns, ShiftRows,... Do đó, việc song song hoá cài đặt AES có thể đưa về việc cài đặt song song hoá các tính toán trong từng phép biến đổi. Việc này có thể thực hiện được vì trong mỗi phép biến đổi, dữ liệu đầu vào và đầu ra là một ma trận trạng thái kích thước giống nhau, chỉ thay đổi các phần tử bên trong chúng. Phần sau sẽ phân tích khả năng song song hóa cài đặt các phép biến đổi trong AES.

1. Expand Key

Quy trình mở rộng khoá của thuật toán AES được thực hiện theo cách tuần tự, tức là các khóa con được sinh theo thứ tự. Do đó, biện pháp song song không áp dụng vào bước này. Tuy nhiên, phép toán XOR trong quy trình thì có thể thực hiện song song, nhưng lượng công việc lại quá ít, và bước mở rộng khoá lại chỉ tác động lên khoá đầu vào. Do đó, nếu chuyển qua song song hóa sẽ tốn thêm nhiều thời gian để phân tách và kết xuất dữ liệu.

2. SubBytes – InvSubBytes

Bước này thực hiện việc thay thế các phần tử của ma trận trạng thái bằng các giá trị tương ứng thông qua bảng S-box.

```
for (i=0;i<16;i++)

{
x=i & 0x03;
y= i >>2;
state    [4*x    +    y]    =
sbox[state[4*x +y]];
}
```

Trong đoạn mã trên của thao tác SubBytes, các lệnh thực hiện trong vòng lặp là độc lập, do đó có thể cài đặt song song hoá thao tác này.

3. ShiftRows –-InvShiftRows

Thao tác này cố sắp xếp lại vị trí các phần tử trong ma trận xử lý. Việc sắp xếp lại các phần tử này là độc lập với nhau và chỉ thực hiện trên 3 hàng của ma trận.

```
for(i = 0;i < 16; i++)
{
x= i & 0x03;
y= i >> 2;
state[4*x  +  y]  =  state[4*x
+((y + x) & 0x03)];
}
```

Tương tự như thao tác SubBytes, các dòng lệnh trong vòng lặp là độc lập và có thể song song hoá việc cài đặt được phép biến đổi này.

4. MixColumns - InvMixColumns

Như đã nói ở trên, thao tác này thực hiện việc nhân lần lượt từng cột của ma trận trạng thái với một đa thức bậc 3. Việc này có thể biểu diễn dưới dạng phép nhân 2 ma trận. Vì vậy, các biến đổi này hoàn toàn có thể cài đặt song song hóa.

5. AddRoundKey

Trong bước này sẽ thực hiện phép kết hợp ma trận trạng thái với khoá con vừa sinh ra để tạo ra một ma trận mới.

```
for(i = 0; i < 16; i++)
{
x= i & 0x03;
y= i >> 2;
state [4*x + y] = state[4*x +
y]  ^  ((  keysched[y]  &  (0xff
<< (x*8))) >>(x*8));
}
```

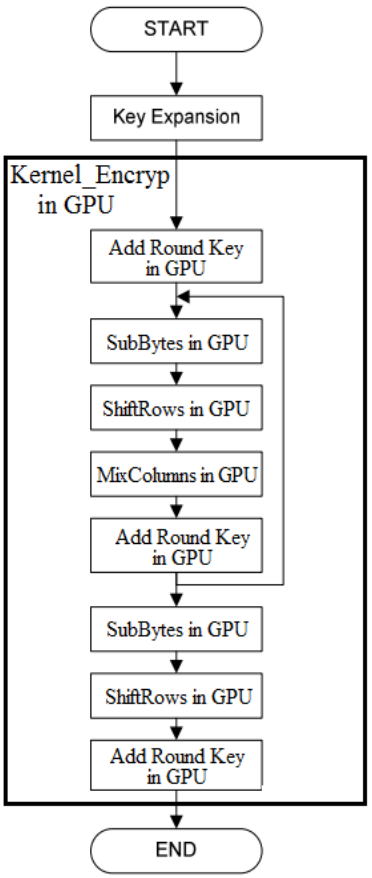
Khoá con là một ma trận cùng kích thước với ma trận trạng thái, nên các thao tác ở đây là thực hiện phép XOR từng phần tử của 2 ma trận với nhau để tạo ra phần tử mới tương ứng. Các thao tác này là độc lập. Do đó, có thể áp dụng biện pháp song song hoá để xử lý phép biến đổi này.

B. Cài đặt song song hóa thuật toán mật mã AES

Phương pháp tiếp cận của chúng tôi là chỉ khai thác khả năng song song cài đặt các phép biến đổi ở mức độ khối dữ liệu mà không thay đổi thuật toán ban đầu. Tức là mỗi một tác vụ sẽ nhận đầu vào là một khối dữ liệu, qua giải thuật sẽ chuyển đổi thành bản mã. Điều này cho thấy, tổng số lượng công việc tỷ lệ thuận với khả năng song song được khai thác.

Quá trình song song hóa cài đặt các phép biến đổi đã nêu trong mục *A* dựa trên phương pháp thiết kế giải thuật song song bằng phương pháp "chia để trị".

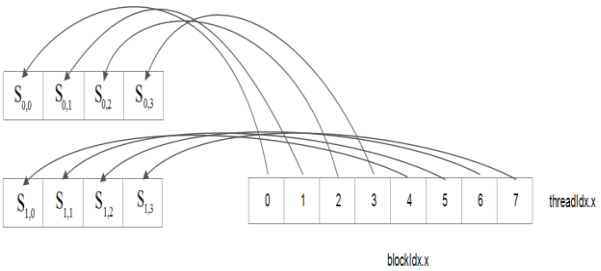
Mô hình song song hóa cài đặt quá trình mã hóa AES được mô tả như Hình 5, quá trình giải mã thực hiện cài đặt theo cách tương tự.



Hình 5. Sơ đồ phương thức song song hóa cài đặt quá trình mã hóa AES

Cụ thể phương pháp song song hóa cài đặt thuật toán AES trên GPU với CUDA được thực hiện như sau:

Phép biến đổi SubBytes sẽ được thực thi song song trên GPU. Các byte trong ma trận trạng thái dữ liệu sẽ được trỏ tới bởi một chương trình con. Theo Hình 6, hai hàng của mảng trạng thái dữ liệu được các chương trình con trong khối các chương trình con trỏ tới.



Hình 6. Các chương trình con trỏ tới từng byte trong ma trận trạng thái dữ liệu

Khi đó các chương trình con sẽ thực hiện song song công việc tra bảng giá trị và thay thế tương ứng.

```
    __device__                  void
SubByte(unsigned   char     *dataA,
int Number_of_matrix, int tid)
    {
    if  (tid<  Number_of_matrix  *
16)
    dataA[tid]                    =
const_Box[(dataA[tid]  >>  4)  *  16
+ (dataA[tid] & 0x0f)];
    }
```

Phép biến đổi ShiftRows cũng được thực thi song song trên GPU. Bằng kỹ thuật sử dụng con trỏ tương tự như phép SubBytes, thì các chương trình con sẽ lần lượt trỏ tới các byte trong ma trận trạng thái. Và các con trỏ chương trình con này cũng sẽ đồng thời thực hiện các phép dịch vòng trái song song.

```
  __device__   void ShiftRow(unsigned
char   *dataA,  int  Number_of_matrix,
int tid)
  {
   if ((tid % 16)>3)
   {
     int x = tid % 16;
     int y = tid / 16;
     if ((x >= 4 && x <= 7))
     {
     int temp = dataA[y * 16 + 4];
       dataA[tid] = dataA[tid + 1];
       dataA[y * 16 + 7] = temp;
     }
```

```
    if (x >= 8 && x <= 9)
    {
      int temp = dataA[tid];
      dataA[tid] = dataA[tid + 2];
      dataA[tid + 2] = temp;
    }
    if (x >= 12 && x <= 15)
    {
      int temp = dataA[y * 16 + 15];
      dataA[tid] = dataA[tid - 1];
      dataA[y * 16 + 12] = temp;
    }
  }
}
```

Phép biến đổi MixColumns được thực thi song song trên GPU bằng cách sử dụng con trỏ các chương trình con như hai phép biến đổi trên. Các chương trình con khi trỏ tới các byte trong ma trận trạng thái sẽ thực hiện song song phép nhân ma trận.

```
  __device__                          void
MixColumn(unsigned   char     *dataA,
unsigned     char     *dataB,     int
Number_of_matrix, int tid)
  {
   dataB[tid] = dataA[tid];
   if (tid < Number_of_matrix * 16)
   {
     int x = tid % 16;
     if (x == 0) //02 03 01 01
       dataA[tid]                    =
GPUgfmultby02(dataB[tid   +   0])   ^
GPUgfmultby03(dataB[tid   +   4])   ^
GPUgfmultby01(dataB[tid   +   8])   ^
GPUgfmultby01(dataB[tid + 12]));
     .....................
     if (x == 15)//03 01 01 02
       dataA[tid]                    =
GPUgfmultby03(dataB[tid   -   12])   ^
GPUgfmultby01(dataB[tid   -   8])   ^
GPUgfmultby01(dataB[tid   -   4])   ^
GPUgfmultby02(dataB[tid - 0]));
   }
  }
```

Phép Add Round Key thực thi song song trên GPU: Các chương trình con lần lượt trỏ tới các byte trong mảng trạng thái và cả các byte trong mảng khóa đã được mở rộng. Chúng sẽ thực hiện phép XOR song song cùng lúc.

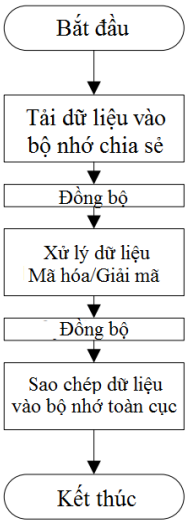
```
  __device__                      void
AddRoundKey(unsigned            char
*dataA,    int        Round,    int
Number_of_matrix,   unsigned   char
*dataW, int tid)
  {
   int x = tid % 16;
  dataA[tid]   =   (dataA[tid]   ^
dataW[Round * 16 + x]);
  }
```

Cuối cùng, tất các các phép biến đổi này được thực thi bởi từng hàm con riêng rẽ. Các hàm con này sẽ được thực thi trong hàm kernel là Kernel_Cryp. Hàm Kernel_Cryp sẽ được khai báo số lượng các lưới chương trình con, số lượng khối chương trình con trong một lưới chương trình con và số lượng chương trình con trong một khối chương trình con để thực thi song song các hàm con bên trong nó.

```
  _global__                          void
Kernel_Cryp(unsigned  char    *dataA,
unsigned   char       *dataB,    int
Number_of_matrix,    unsigned    char
*dataW)  // Hàm kernel
  {
   //thứ tự thread trên 1 grid
   int col = blockDim.x*blockIdx.x +
threadIdx.x;
   int row = blockDim.y*blockIdx.y +
threadIdx.y;
   int     tid     =     col     +
row*gridDim.x*blockDim.x;
   int round = 0;
   __shared__      unsigned      char
S_Key[240];
   for (int i = 0; i < 240; i++)
     S_Key[i] = dataW[i];
   AddRoundKey(dataA,0,Number_of_mat
rix, S_Key, tid);
   __syncthreads();
   for  (round  =  1;  round  <  14;
round++)
   {
     SubByte(dataA,Number_of_matrix,
tid);
     __syncthreads();
     ShiftRow(dataA,Number_of_matrix,
tid);
     __syncthreads();
     MixColumn(dataA,dataB,Number_of_
matrix,tid);
     __syncthreads();
     AddRoundKey(dataA,round,Number_o
f_matrix,S_Key,tid);
     __syncthreads();
   }
   SubByte(dataA,Number_of_matrix,
tid);
   __syncthreads();
   ShiftRow(dataA,Number_of_matrix,
tid);
   __syncthreads();
```

```
    AddRoundKey(dataA,14,Number_of_ma
trix, S_Key, tid);
    __syncthreads();
  }
```

Để tối ưu truy cập bộ nhớ toàn cục thì việc thực thi các chương trình con được chia thành 3 giai đoạn (Hình 7).



Hình 7. Quá trình xử lý dữ liệu trong chương trình cài đặt

Theo Hình 7, các khối chương trình con sẽ không làm việc trực tiếp với dữ liệu trong bộ nhớ toàn cục. Vì bộ nhớ toàn cục có dung lượng lớn nhưng lại có độ trễ cao và băng thông thấp, dẫn tới làm chậm quá trình mã hóa. Quá trình tối ưu truy cập bộ nhớ toàn cục được thực hiện nhờ chiến lược sử dụng bộ nhớ chia sẻ như sau:

- Đầu tiên, cần xác định phần dữ liệu trong bộ nhớ toàn cục mà mỗi khối chương trình con cần để xử lý.
- Tiếp theo, cho mỗi khối chương trình con chuyển phần dữ liệu cần xử lý từ bộ nhớ toàn cục vào bộ nhớ chia sẻ của mình.
- Sau đó, các chương trình con trong mỗi khối sẽ tiến hành xử lý với dữ liệu đã được lưu ở trong bộ nhớ chia sẻ cho mỗi khối. Trong quá trình xử lý, có thể lưu tạm kết quả xử lý ở bộ nhớ chia sẻ (hoặc bộ nhớ thanh ghi) và đồng bộ lại khi kết thúc. Bằng cách này, nếu dữ liệu ở bộ nhớ chia sẻ được dùng lại nhiều lần và kết quả xử lý được cập nhật nhiều lần thì sẽ tiết kiệm được đáng kể số lần truy xuất xuống bộ nhớ toàn cục.
- Cuối cùng, thực hiện ghi kết quả từ bộ nhớ chia sẻ xuống bộ nhớ toàn cục.

Một cách tối ưu hữu ích khác cũng được nhóm tác giả cài đặt trong chương trình là sử dụng bộ nhớ hằng. Trong cài đặt AES trên CPU thường sử dụng một vài bảng có kích thước16x16 byte để tra cứu. Các bảng này là hằng số và không đổi cho tất cả các chương trình con. Vì vậy, khi cài đặt AES trên GPU có thể chuyển các bảng này vào bộ nhớ hằng để có thể tăng tốc độ tra cứu lên gấp hàng trăm lần mà không phải truy cập tới bộ nhớ toàn cục nữa.

V. ĐÁNH GIÁ HIỆU SUẤT CHƯƠNG TRÌNH CÀI ĐẶT

A. Phương pháp kiểm tra và đánh giá chương trình cài đặt

Phương pháp kiểm tra và đánh giá chương trình là dựa trên sự đúng đắn của việc cài đặt: Cần đảm bảo chắc chắn rằng việc cài đặt AES trên GPU không thay đổi chức năng của thuật toán theo các tiêu chí sau:

- Chắc chắn rằng chương trình mã hóa với đầu vào đã biết phải cho ra bản mã tương ứng: Các cặp đầu vào rõ/mã được lấy đúng so với công bố của Viện tiêu chuẩn và công nghệ quốc gia Hoa Kỳ ([7]).
- Chắc chắn rằng việc giải mã bản mã phải cho ra bản rõ ban đầu.

B. Hiệu suất chương trình cài đặt

Để đánh giá hiệu suất chương trình cài đặt AES trên GPU, nhóm tác giả đã so sánh với kết quả cài đặt AES theo phương pháp tuần tự trên CPU với tiêu chí thời gian thực hiện trên cùng một bản rõ. Các thông tin phần cứng cài đặt thuật toán trên GPU và CPU như sau:

Cài đặt AES-256 tuần tự trên CPU: thực hiện trên Visual Studio 2013 Utimate trên máy tính xách tay với CPU Intel Core i7-4800MQ 2.7GHz.

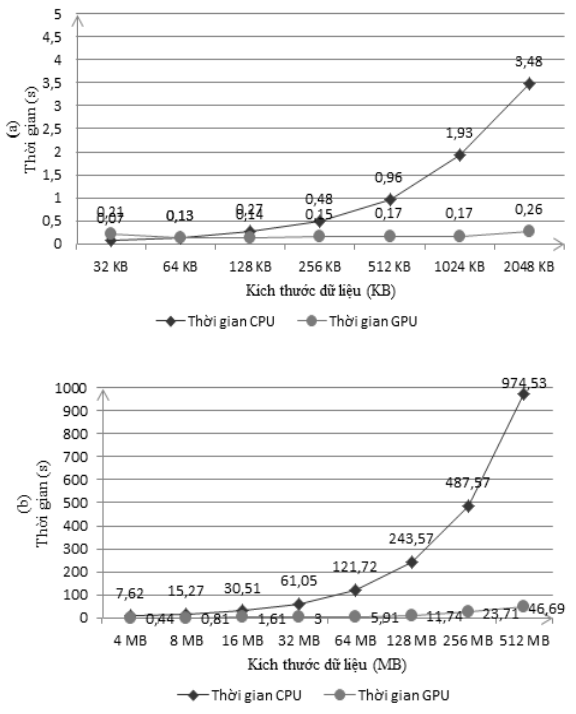
Cài đặt AES-256 trên GPU: Thực hiện trên Visual Studio 2013 Utimate và CUDA Toolkit 7.5 trên máy tính xách tay với CPU Intel Core i7-4800MQ 2.7GHz, card đồ họa NVIDIA Quadro K1100M.

Kết quả tốc độ chạy thử nghiệm chương trình cài đặt được thể hiện trong Bảng 1.

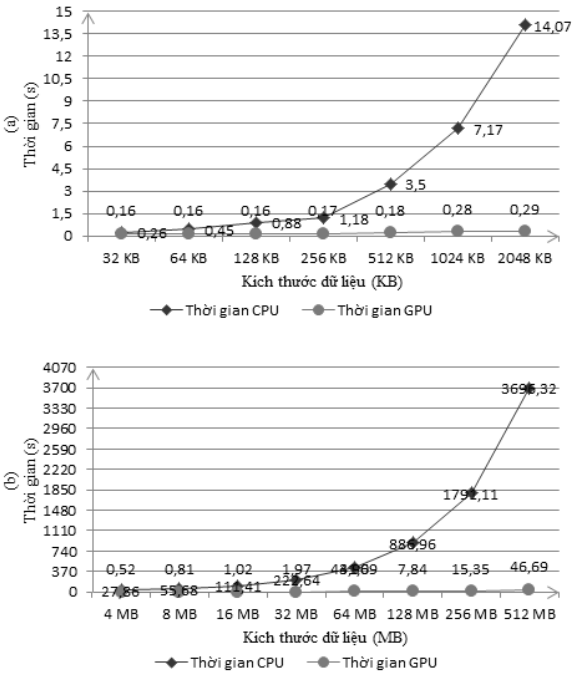
BẢNG 1. TỐC ĐỘ CHƯƠNG TRÌNH CÀI ĐẶT

Dung lượng tập tin	Thời gian CPU (giây)		Thời gian GPU (giây)		Tỉ số tăng tốc (CPU/GPU)	
	Mã hóa	*Giải mã*	*Mã hóa*	*Giải mã*	*Mã hóa*	*Giải mã*
32 KB	0,071	0,262	0,209	0,160	0,34	1,64
64 KB	0,131	0,445	0,134	0,161	0,98	2,76
128 KB	0,256	0,879	0,140	0,162	1,83	5,43
256 KB	0,483	1,179	0,145	0,168	3,33	10,59
512 KB	0,959	3,493	0,171	0,178	5,61	19,62
1 MB	1,926	7,171	0,173	0,276	11,13	25,98
2 MB	3,482	14,072	0,261	0,291	14,72	48,36
4 MB	7,623	27,859	0,443	0,522	17,23	53,37
8 MB	15,274	55,680	0,811	0,814	18,8	54,8
16 MB	30,505	111,411	1,607	1,023	18,98	56,47
32 MB	61,051	222,635	3,004	1,971	20,32	56,24
64 MB	121,719	441,087	5,912	3,962	20,59	56,3
128 MB	243,573	886,956	11,744	7,837	20,74	57,8
256 MB	487,566	1791,106	23,712	15,351	20,56	58,21
512 MB	974,527	3696,315	46,689	46,688	20,87	58,27

Sử dụng các dữ liệu thu được để xây dựng biểu đồ so sánh khả năng xử lý giữa CPU và GPU trong trường hợp mã hóa (Hình 8) và giải mã (Hình 9).

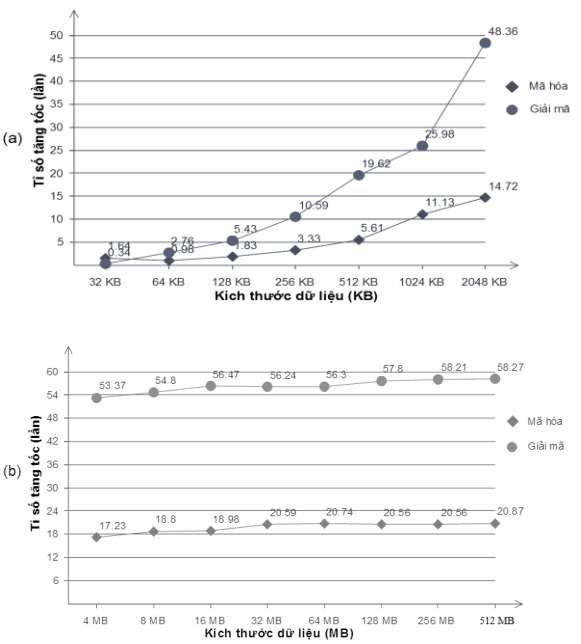


Hình 8. Biểu đồ so sánh thời gian thực hiện mã hóa AES giữa CPU và GPU theo dung lượng tập tin



Hình 9. Biểu đồ so sánh thời gian thực hiện giải mã AES giữa CPU và GPU theo dung lượng tập tin

Nhận xét về tỷ lệ tăng tốc độ thực thi thuật toán AES giữa CPU và GPU trong quá trình thử nghiệm chương trình cài đặt. Với những mức kích thước dữ liệu rất nhỏ thì tốc độ thực thi của CPU nhanh hơn của GPU. Khi kích thước dữ liệu tăng (đến khoảng 500MB) thì tốc độ GPU tăng lên đáng kể so với CPU.



Hình 10. Biểu đồ thể hiện tỷ lệ tăng tốc độ giữa CPU và GPU khi kích thước dữ liệu tăng

Tốc độ xử lý chênh lệch lớn nhất là khi tốc độ xử lý của GPU gấp khoảng 21 lần khi mã hóa và khoảng 58 lần khi giải mã so với CPU (Hình 10). Kết quả này cao hơn so với kết quả được công bố trong [3].

Khi kích thước dữ liệu nhỏ, tốc độ thực thi thuật toán AES trên GPU chậm hơn so với CPU do tốn thời gian truyền dữ liệu từ bộ nhớ của CPU sang bộ nhớ của GPU. Ngược lại, đối với kích thước dữ liệu lớn so với bộ nhớ CPU nhỏ thì các khối dữ liệu sẽ được thực thi tuần tự theo hình thức hàng đợi, còn GPU có kích thước bộ nhớ lớn hơn sẽ thực hiện được nhiều khối dữ liệu cùng lúc nên xử lý nhanh hơn.

BẢNG 2. SO SÁNH TỶ SỐ TĂNG TỐC GPU/CPU VỚI KẾT QUẢ TRONG [6]

Dung lượng tập tin (KB)	Tỷ số tăng tốc (CPU/GPU) [6]	Tỷ số tăng tốc (CPU/GPU)
32	n/a	0,34
64	n/a	0,98
128	n/a	1,83
256	n/a	3,33
512	0,1	5,61
1024	0,1	11,13
2048	0,2	14,72
4096	0,2	17,23
8192	0,3	18,8
16384	0,5	18,98
32768	1,1	20,32
65536	2,0	20,59
131072	2,7	20,74
262144	4,0	20,56
524288	5,1	20,87

VII. KẾT LUẬN

Trong bài báo này, nhóm tác giả đã phân tích khả năng song song hóa cài đặt thuật toán mật mã AES. Từ đó, đưa ra phương pháp cài đặt song song hóa thuật toán này với ngôn ngữ lập trình CUDA. Nhóm tác giả đã thực hiện cài đặt thuật toán mật mã AES-256 với trình biên dịch Visual Studio 2013 Utimate và CUDA Toolkit 7.5 trên máy tính xách tay với CPU Intel Core i7-4800MQ 2.7GHz, card đồ họa NVIDIA GPU Quadro K1100M. Kết quả cài đặt và đánh giá cho thấy việc cài đặt song song hóa thuật toán AES-256 trên GPU cho hiệu suất cao hơn rất nhiều so với việc cài đặt thuật toán này trên CPU. Điều này cho thấy, khả năng tính toán song song rất lớn của GPU có thể ứng dụng vào tăng hiệu suất thực thi các thuật toán mật mã khác. Hơn nữa, kết quả của nhóm tác giả cho thấy tỷ lệ tăng tốc độ thực hiện thuật toán AES trên GPU so với CPU cao hơn kết quả đã công bố trong [3]. Thực tế, tốc độ thực thi của thuật toán cơ bản phụ thuộc vào 2 yếu tố: phương pháp cài đặt song song hóa và card đồ họa GPU sử dụng. Vì vậy, để có kết quả tốt hơn thì phải tiếp tục nghiên cứu tối ưu phương pháp cài đặt song song hóa và nghiên cứu cài đặt song song hóa các thuật toán mật mã trên hệ thống có nhiều GPU chạy song song và trên hệ thống Cluster với nhiều máy tính tích hợp sẵn GPU.

TÀI LIỆU THAM KHẢO

[1]. NVIDIA, "CUDA C Programming Guide", Vesion 7.5, 2015.

[2]. Aaron Riley, "Getting Started with CUDA", BookRix GmbH & Co. KG 81669 Munich, 2016.

[3]. Michael Kipper, Joshua Slavkin, Dmitry Denisenko, "Implementing AES on GPU Final Report", University of Toronto, 2009.

[4]. FIPSP.197, "Advanced encryption standard (AES)", 2001.

[5]. Svetlin A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography", In ICSPC 2007.

[6]. Keisuke Iwai, Naoki Nishikawa, Takakazu Kurokawa, "Acceleration of AES encryption on CUDA GPU", International Journal of Networking and Computing, 2012.

[7]. NIST, "AES test vectors", http://csrc.nist.gov/groups/STM/cavp/documents/aes/KAT_AES.zip

SƠ LƯỢC VỀ TÁC GIẢ

ThS. Đinh Tiến Thành

Đơn vị công tác: Học viện Kỹ thuật mật mã - Ban Cơ yếu Chính phủ

Email: thanhhvkt@yahoo.com

Quá trình đào tạo: Nhận bằng Kỹ sư và Thạc sĩ chuyên ngành Kỹ thuật mật mã tại Học viện Kỹ thuật mật mã năm 2000 và 2008.

Hướng nghiên cứu hiện nay: An toàn thông tin và Khoa học mật mã

ThS. Phạm Mạnh Tuấn

Đơn vị công tác: Công ty TNHH MTV 129, Ban Cơ yếu Chính phủ.

Email: tuanpm.129@gmail.com

Quá trình đào tạo: Nhận bằng kỹ sư tại Học viện Công nghệ Bưu chính Viễn thông năm 2003. Nhận bằng Thạc sĩ tại Học viện Kỹ thuật quân sự năm 2008, hiện đang làm Nghiên cứu sinh tại Học viện Công nghệ Bưu chính Viễn thông.

Hướng nghiên cứu chính hiện nay: Thiết kế và thực thi các thuật toán mật mã trên các thiết kế phần cứng; Nghiên cứu tổng thể giải pháp bảo mật dữ liệu thoại, video trên các môi trường truyền thông khác nhau.