Báo cáo giải quyết bài toán Weapon – Target Assignment

**1. Đặt vấn đề**

- Sau khi đã chạy thử thuật toán, ta cần xây dựng ứng dụng có giao diện cho phép người dùng nhập đầu vào và hiển thị ra kết quả

- Xem xét với trường hợp số lượng vũ khí và mục tiêu là khác nhau

- Tiếp tục tìm các phương án cải thiện tốc độ của thuật toán

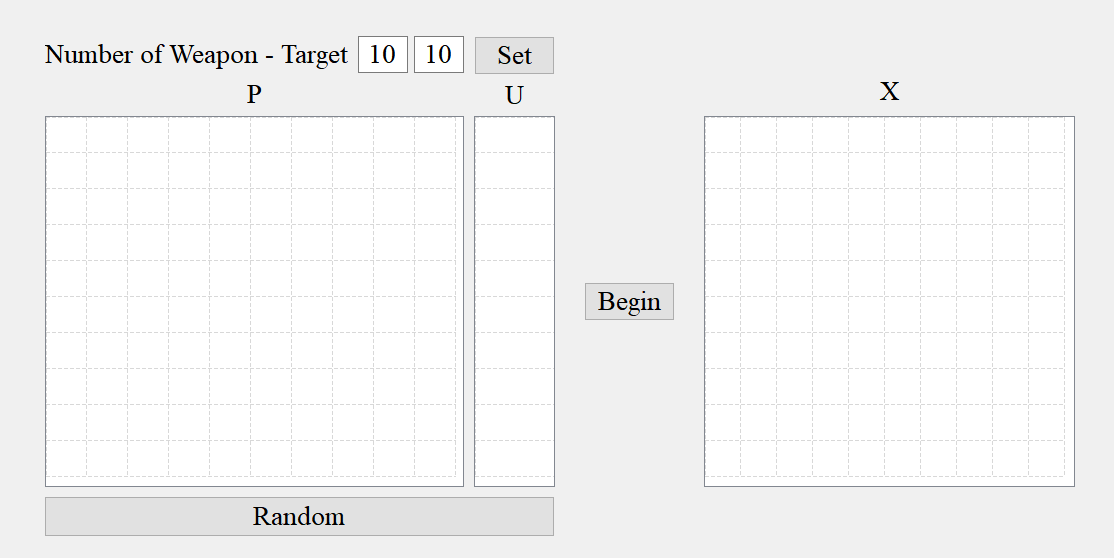
**2. Phương án giải quyết**

**2.1. Xây dựng giao diện**

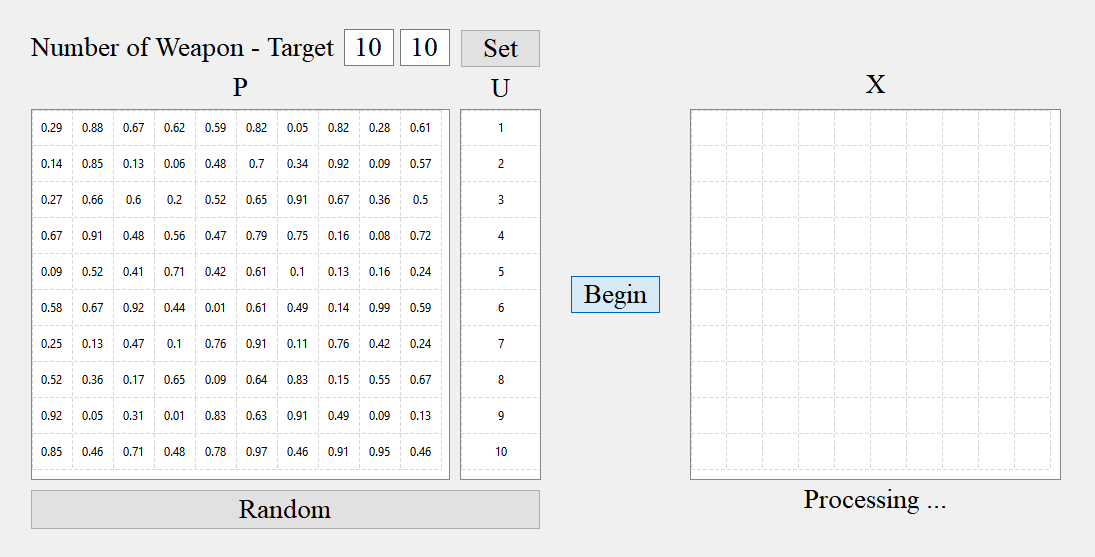
Ta sẽ sử dụng công cụ QtCreator để xây dựng giao diện, với yêu cầu như sau

* Có thể nhập số lượng mục tiêu, số lượng vũ khí
* Có thể nhập bảng giá trị của p và u
* Nếu không có giá trị p và u nhập vào, có thể tự xuất dữ liệu random
* Có khả năng hiển thị kết quả và thời gian chạy thuật toán

Giao diện sau khi xây dựng



Giao diện lúc đang trong quá trình chạy thuật toán



Giao diện khi đã chạy xong thuật toán



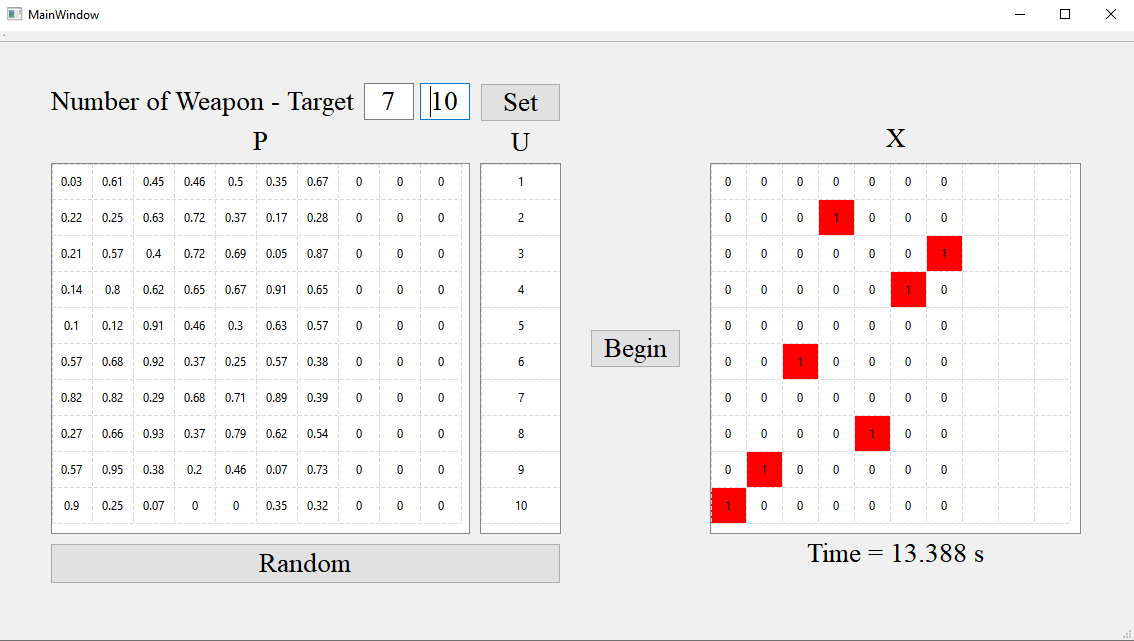
**2.2. Xem xét với trường hợp số lượng mục tiêu và vũ khí là khác nhau**

**2.2.1 Số lượng mục tiêu nhiều hơn số lượng vũ khí**

- Để đảm bảo điều kiện thực hiện thuật toán, ta cần phải có đủ phần tử của ma trận p, do vậy đối với số lượng mục tiêu nhiều hơn vũ khí, ta sẽ tạo ra một số lượng vũ khí giả, với xác suất bắn trúng đối với tất cả các mục tiêu là bằng 0.

- Do hàm mục tiêu có dạng , với pij = 0 thì giá trị của x­ij là bao nhiêu cũng không ảnh hưởng đến kết quả thu được. Như vậy ta có thể kết luận phương pháp trên là sử dụng được.

- Chạy thử thuật toán với số lượng mục tiêu là 10, số lượng vũ khí là 7 ta thu được kết quả như sau



**2.2.2 Số lượng mục tiêu ít hơn số lượng vũ khí**

- Khi số lượng mục tiêu ít hơn số lượng vũ khí, để tăng khả năng tiêu diệt, sẽ có trường hợp nhiều hơn 1 vũ khí tấn công vào 1 mục tiêu. Ta sẽ thực hiện thuật toán như sau

- Tương tự như trường hợp trên, ta tạo ra các mục tiêu giả, với xác suất bắn trúng mục tiêu đó của toàn bộ vũ khí là bằng 0 và mức độ ưu tiên của nó cũng bằng 0. Như vậy hàm mục tiêu sẽ không bị ảnh hưởng bởi số mục tiêu giả đó.

- Chạy thuật toán lần thứ nhất, ta sẽ thu được kết quả trong đó có các vũ khí nhắm vào các mục tiêu giả. Ta sẽ đánh dấu lại những vũ khí đó.

- Tiếp theo ta sẽ chạy thuật toán lần thứ hai, với số lượng mục tiêu vẫn như cũ tuy nhiên số vũ khí chỉ là những vũ khí mà ta đã đánh dấu ở lần 1. Đưa bài toán về trường hợp thứ nhất – số lượng mục tiêu nhiều hơn số lượng vũ khí. Sau đó ta thực hiện chạy thuật toán với trường hợp kể trên.

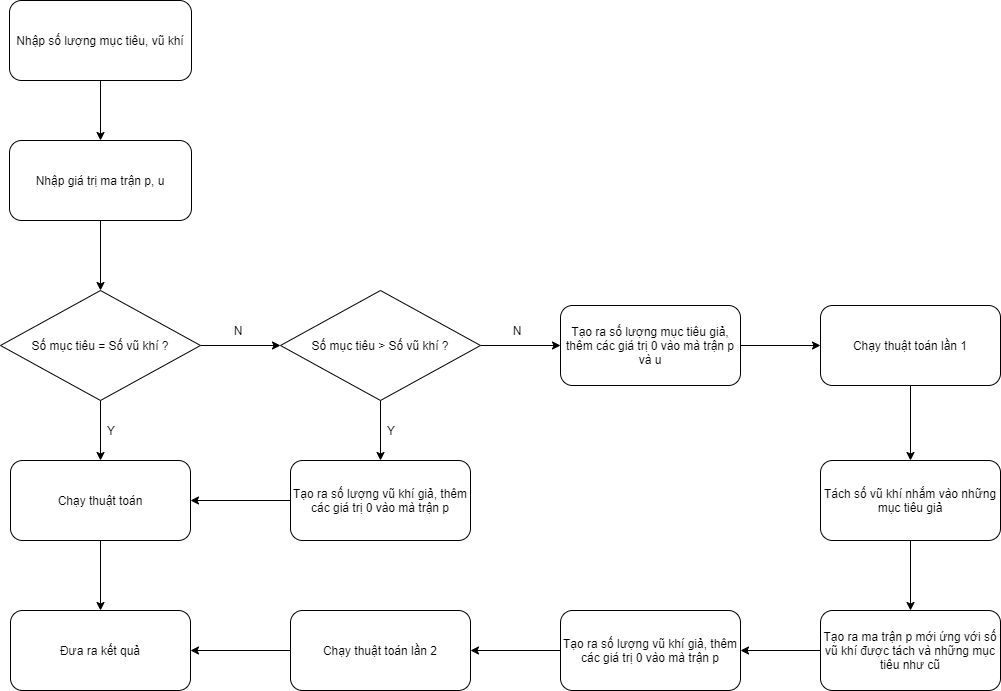
- Tổng hợp của hai lần chạy thuật toán, ta sẽ thu được kết quả

- Chạy thử thuật toán với số vũ khí là 10, số mục tiêu là 7 ta thu được kết quả như sau



**2.2.3 Sơ đồ khối của thuật toán**

Tổng kết lại ta có sơ đồ khối như sau



**2.3 Tìm các phương án để cải thiện tốc độ của thuật toán**

**2.3.1 Sử dụng Multi Thread**

Thuật toán đang sử dụng bao gồm 1 vòng lặp lớn được tính toán một cách tuần tự, với đầu ra của chu kỳ tính toán thứ k được lấy làm đầu vào của chu kỳ tính toán thứ k+1. Bên trong của vòng lặp này bao gồm 2 bước.

Bước 1 là tính toán giá trị đạo hàm của các biến trạng thái

Bước 2 là Update giá trị đầu ra

Trong mỗi bước trên ta sẽ có các vòng lặp có thể được tính toán một cách đồng thời do chúng không phụ thuộc vào kết quả tính toán của nhau. Ý tưởng sẽ là tối ưu hóa thời gian tính toán cho vòng trong.

Qt có hỗ trợ đa luồng với thư viện QThread. Ngoài ra, ta có thể sử dụng API cấp cao hơn của Qt để tạo đa luồng đó là QtConcurrent. Việc thực hiện sẽ đơn giản hơn nhiều, ta có thể thực hiện một ví dụ đơn giản như sau.

#include <QCoreApplication>

#include <qtconcurrentrun.h>

#include <QThread>

#include <time.h>

int n =10;

void **myRunFunction**(int stt)

{

for(int a=0; a<1000; a++)

{

a = a+1;

a = a-1;

}

}

int **main**(int argc, char \*argv[])

{

QCoreApplication a(argc, argv);

//pallarel loop

clock\_t t1 =clock();

for (int k=0; k<1000; k++)

{

QFuture<void> t[n];

for (int i=0; i<n; i++)

{

t[i] = QtConcurrent::run(myRunFunction, i);

t->waitForFinished();

}

}

printf("%5.2f", (double)(clock()-t1)/CLOCKS\_PER\_SEC);

//Normal\_loop

t1 = clock();

for (int k=0; k<1000; k++)

{

for (int i=0; i<n; i++)

{

for(int a=0; a<1000; a++)

{

a = a+1;

a = a-1;

}

}

}

clock\_t t2 = clock();

printf("%5.2f", (double)(t2-t1)/CLOCKS\_PER\_SEC);

return a.exec();

}

Ý tưởng sẽ là thực hiện vòng lặp với 1000 chu kỳ. Trong đó mỗi chu kỳ lại thực hiện n vòng lặp for. Mỗi vòng lặp đó ta sẽ tính 2 phép tính đơn giản m lần. Ta sẽ so sánh thời gian thực hiện khi sử dụng tính toán song song và tính toán tuần tự.

Với n=10, m = 1000 ta thu được T1 = 0.32, T2 = 0.05

Với n=10, m = 10000 ta thu được T1 = 0.34, T2=0.48

Sau khi thử với một vài ví dụ đơn giản, nhận thấy thời gian được cải thiện khá nhiều nếu vòng lặp trong mỗi thread tốn một khoảng thời gian nhất định. Tuy nhiên khi ta giảm thời gian tính toán của mỗi thread. Kết quả nhận được khi sử dụng MultiThread sẽ kém hơn nhiều so với việc sử dụng tính toán một cách tuần tự. Điều này có thể được giải thích bằng việc thời gian ta tiết kiệm được bằng việc sử dụng MultiThread là nhỏ hơn nhiều so với thời gian để khởi tạo vào kích hoạt threads.

Như vậy có thể kết luận được tính toán song song chỉ có hiệu quả khi thời gian thực hiện các phép toán lặp là lớn. Đối với phép toán tốn ít thời gian, việc chuyển từ tuần tự sang song song sẽ mất thời gian hơn.

Thử nghiệm đối với bài toán ta đang xét đến, ta thấy kết quả thu được chậm hơn khá nhiều so với việc không sử dụng MultiThread

**2.3.2 Sử dụng pallarell programing with CUDA**

Ngày nay, khi việc tính toán song song trở nên phổ biến hơn, ta có thể sử dụng các GPU chuyên dụng để cải thiện thời gian tính toán. Để làm được điều đó, ta có thể sử dụng CUDA Toolkit của Nvidia được thiết kế để lập trình song song với GPU.

1. Giới thiệu về CUDA

CUDA (Compute Unified Device Architecture) là ngôn ngữ C/C++ được mở rộng, cho phép viết chương trình chạy trên CPU (những phần tính toán tuần tự) và GPU (những phần tính toán song song).

CPU có một vài core, mỗi core mạnh và phức tạp. Tập trung tối ưu hóa độ trễ (latency)

Độ trễ = thời gian hoàn thành một công việc

GPU có rất rất nhiều core, mỗi core yếu và đơn giản. Tập trung tối ưu hóa băng thông (throughput)

Băng thông = số lượng công việc hoàn thành trong một đơn vị thời gian

Cấu trúc của một chương trình CUDA

Một số khái niệm:

– Host: CPU và bộ nhớ của nó

– Device: GPU và bộ nhớ của nó

– Hàm kernel <<<…>>>: cấu hình thực thi, được gọi từ host

\_\_global\_\_: hàm sẽ chạy trên device và được gọi từ host

\_\_host\_\_: hàm sẽ chạy trên host và được gọi từ host

\_\_device\_\_: hàm sẽ chạy trên device và được gọi từ device

Những phần tính toán tuần tự chạy ở host (CPU), những phần tính toán song song (mức độ lớn) chạy ở device (GPU)

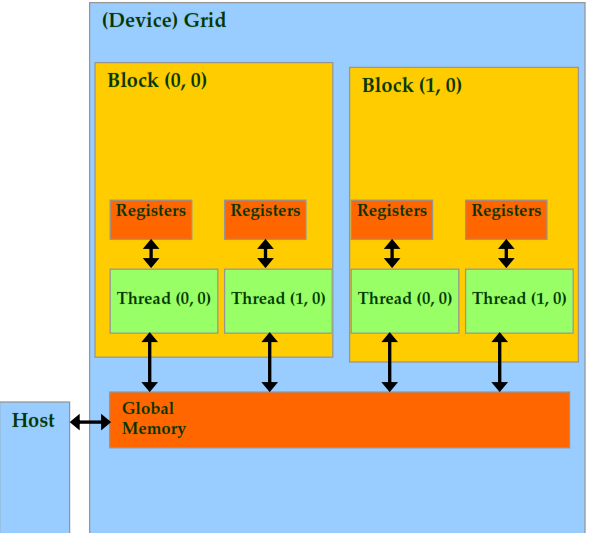
Từ host, để nhờ device tính toán song song cần theo các bước sau

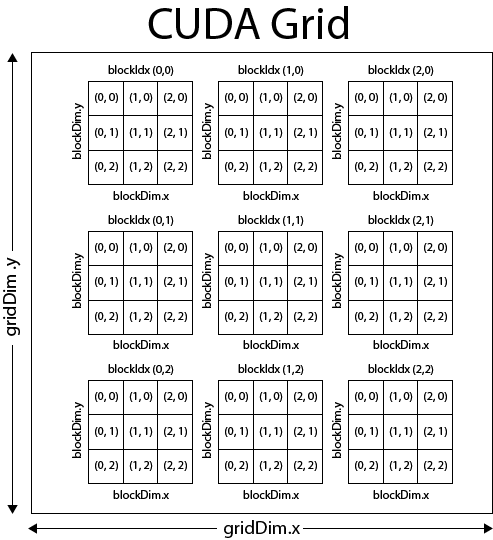
* Host cấp phát các vùng nhớ ở device bằng hàm cudaMalloc
* Host chép các dữ liệu cần thiết sang các vùng nhớ ở device bằng hàm cudaMemcpy
* Host gọi hàm kernel
* Host chép kết quả từ device về bằng hàm cudaMemcpy
* Host giải phóng các vùng nhớ ở device bằng hàm cudaFree
* Hàm kernel dược thực thi song song ở device bởi rất nhiều thread

Các thread này được tổ chức thành 2 cấp: grid gồm các block có cùng kích thước, block gồm các thread.

Khi host gọi hàm kernel cần cho biết grid gồm bao nhiêu block, mỗi block có bao nhiêu thread (blockSize, gridSize).

Trong hàm kernel, mỗi thread có thể sử dụng các biến hệ thống blockIdx, threadIdx, blockDim, gridDim để xác định phần dữ liệu mà mình sẽ phụ trách tính toáncuda\_memory





Một tính chất của hàm kernel là “không đồng bộ” (asynchronous): sau khi host gọi hàm kernel ở device, host sẽ được tự do tiếp tục làm các công việc của mình mà không phải chờ hàm kernel ở device thực hiện xong.

1. Áp dụng vào thuật toán đang sử dụng

…

Định hướng tìm hiểu tiếp theo

* Lập trình một bài toán nhỏ liên quan đến việc tính toán song song sử dụng CUDA và so sánh với việc tính toán một cách tuần tự
* Áp dụng vào bài toán đang thực hiện với hy vọng cải thiện tốc độ của thuật toán