1. Использование GPU для различных криптографических приложений

Достаточно часто, решая прикладные криптографические задачи, приходится перебирать некоторое конечное количество значений на одинаковой схеме (подбор пароля по его хэшу, подбор пароля по развертке ключа и по проведению процедуры расшифровывания симметричного алгоритма и т.д.).

Задачи такого плана очень хорошо ложатся на архитектуру SIMD. Кратко опишем основные архитектуры, которые понадобятся нам далее:

SISD - (Single Instruction Single Data) - архитектура, при которой один поток инструкций выполняется над одним потоком данных. Данная архитектура больше всего распространена, так как она является архитектурой одноядерного процессора CPU.

SIMD - (Single Instruction Multiple Data) - архитектура, при которой один поток инструкций выполняется над несколькими потоками данных. То есть код выполняемой программы одинаковый для всех потоков данных, а сами значения в этих данных различны. Типичным представителем данной архитектуры является GPU (Graphics Processing Unit).

Почему же именно SIMD так хорош для проведения подбора паролей? Потому что большинство криптографических алгоритмов являются не сильно ветвистыми (то есть в них нет большой вариативности действий в зависимости от входных данных), и поэтому в один момент времени будут выполняться одинаковые шаги алгоритма для разных потоков, если их запустить параллельно.

1.1. Сравнение CPU и GPU

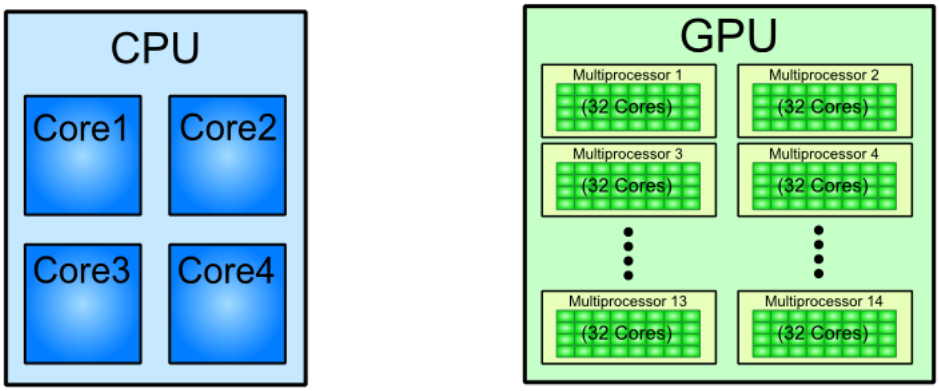
Первый вопрос, который должен задать каждый перед применением GPU для решения своих задач — а для каких целей хорош GPU, когда стоит его применять? Для ответа нужно определить 2 понятия:

- *Задержка* (latency) — время, затрачиваемое на выполнение одной инструкции/операции.
- *Пропускная способность* — количество инструкций/операций, выполняемых за единицу времени.

Для того, чтобы разобраться в этих терминах, приведем простой пример: допустим, мы имеем легковой автомобиль со скоростью 90 км/ч и вместимостью 4 человека, и автобус со скоростью 60 км/ч и вместимостью 20 человек. Если за операцию принять перемещение 1 человека на 1 километр, то задержка легкового автомобиля — 3600/90=40с — за столько секунд 1 человек преодолеет расстояние в 1 километр, пропускная способность автомобиля — 4/40=0.1 операций/секунду; задержка автобуса — 3600/60=60с, пропускная способность автобуса — 20/60=0.3(3) операций/секунду.

Так вот, CPU в данной ситуации можно сравнить с автомобием, а GPU — с автобусом: он имеет большую задержку но также и большую пропускную способность. Если для вашей задачи задержка каждой конкретной операции не настолько важна как количество этих операций в секунду — стоит рассмотреть применение GPU.

Сравнение архитектур CPU и GPU



Наши однотипные вычисления при их массовости отлично выписываются в класс таких задач - попробуем организовать их расчет на GPU.

1.2. Базовые понятия и термины CUDA

Итак, разберемся с терминологией CUDA:

- Устройство (device) — GPU. Выполняет роль «подчиненного» — делает только то, что ему говорит CPU.
- Хост (host) — CPU. Выполняет управляющую роль — запускает задачи на устройстве, выделяет память на устройстве, перемещает память на/с устройства. И да, использование CUDA предполагает, что как устройство так и хост имеют свою отдельную память.
- Ядро (kernel) — задача, запускаемая хостом на устройстве.

При использовании CUDA вы просто пишете код на языке программирования C или C++, после чего компилятор CUDA сгенерирует код отдельно для хоста и отдельно для устройства.

Небольшая оговорка: код для устройства должен быть написан только на языке C с некоторыми 'CUDA-расширениями'.

1.2.1. Основные этапы CUDA-программы

Рассмотрим общую схему работы любой программы, написанной для CUDA:

1) Хост выделяет нужное количество памяти на устройстве.

2) Хост копирует данные из своей памяти в память устройства.

3) Хост стартует выполнение определенных ядер на устройстве.

4) Устройство выполняет ядра.

5) Хост копирует результаты из памяти устройства в свою память.

Естественно, для наибольшей эффективности использования GPU нужно чтобы соотношение времени, потраченного на работу ядер, к времени, потраченному на выделение памяти и перемещение данных, было как можно больше, а также желательно, чтобы из памяти видеокарты в память хоста и наоборот за один раз копировалось как можно больше информации - это повысит скорость копирования.

1.2.2. Ядра

Рассмотрим более детально процесс написания кода для ядер и их запуска. Важный принцип — ядра пишутся как (практически) обычные последовательные программы — то-есть вы не увидите создания и запуска потоков в коде самих ядер. Вместо этого, для организации параллельных вычислений GPU запустит большое количество копий одного и того же ядра в разных потоках — а точнее, вы сами говорите сколько потоков запустить. И да, возвращаясь к вопросу эффективности использования GPU — чем больше потоков вы запускаете (при условии что все они будут выполнять полезную работу) — тем лучше. Код для ядер отличается от обычного последовательного кода в таких моментах:

Внутри ядер вы имеете возможность узнать «идентификатор» или, проще говоря, позицию потока, который сейчас выполняется — используя эту позицию мы добиваемся того, что одно и то же ядро будет работать с разными данными в зависимости от потока, в котором оно запущено. В некоторых случаях в коде ядра необходимо использовать различные способы синхронизации, но конкретно наша задача не будет требовать таких ухищрений.

Каким же образом мы задаем количество потоков, в которых будет запущено ядро? Поскольку GPU это все таки Graphics Processing Unit, то это, естественно, повлияло на модель CUDA, а именно на способ задания количества потоков:

Сначала задаются размеры так называемой сетки (grid) в 3D координатах: grid_x, grid_y, grid_z. В результате, сетка будет состоять из grid_x*grid_y*grid_z блоков. Потом задаются размеры блока в 3D координатах: block_x, block_y, bloc_z. В результате, блок будет состоять из block_x*block_y*block_z потоков. Итого, имеем grid_x*grid_y*grid_z*block_x*block_y*block_z потоков. Важное замечание — максимальное количество потоков в одном блоке ограничено и зависит от модели GPU — типичны значения 512 (более старые модели) и 1024 (более новые модели, как на машине автора статьи). Внутри ядра доступны переменные threadIdx и blockIdx с полями x, y, z — они содержат 3D координаты потока в блоке и блока в сетке соответственно. Также доступны переменные blockDim и gridDim с теми же полями — размеры блока и сетки соответственно.

Источник информации:
https://habr.com/ru/company/epam_systems/blog/245503/

1.3. Установка CUDA development toolkit на ОС ubuntu 20.04

Для работы с видеокартой нам нужно получить к ней прямой доступ, поэтому нам нужно иметь ОС с доступом к ней; при этом нужно учитывать, что это железо очень капризное и очень специфичное в работе.

В общем случае можно работать с CUDA и в Windows, но так как мне привычнее работать в линукс-образных системах, то я установил ubuntu 20.04 второй системой (обойтись виртуальной машиной не получится, так как из решений под Windows нам подойдет только Hyper-V (которая не входит в обычную версию ОС), потому что только она умеет прокидывать видеокарту напрямую в BM, а для qemu в Windows не хватает некоторых драйверов, как выяснилось опытным путем. Vmware и Virtualbox тоже не умеют прокидывать видеокарту напрямую). Поэтому ниже я последовательно опишу действия, которые позволили мне правильно установить CUDA development toolkit.

Для начала я установил ОС ubuntu 20.04, не обновляя её при установке, после чего перешел с ядра 5.8 на более стабильное ядро 5.4 следующими командами:

1) *sudo apt install –install-recommends linux-generic* - скачивание некоторых зависимостей и заголовков для ядра 5.4

2) *sudo update-grub* - обновление grub, чтобы в меню загрузки можно было выбирать нужное ядро в расширенных настройках запуска ubuntu

3) *sudo reboot now* - перезапуск системы из консоли

4) *uname -r* - вывод в консоль версии ядра

Далее рассмотрим последовательность действий для установки самого тулкита CUDA:

1) *sudo apt install gcc python3-dev python3-pip libxml2-dev libxslt1-dev zlib1g-dev g++* - установка некоторых компонентов, которые могут пригодится нам

2) *sudo apt-get install nvidia-cuda-toolkit* - усановка самого тулкита из репозиториев ubuntu (самый лаконичный способ установки, на мой взгляд).

3) *sudo add-apt-repository ppa:graphics-drivers/ppa* - добавление репозиториев драйверов в базу репозиториев инсталятора

4) *sudo ubuntu-drivers devices* - данная команда покажет нам рекомендуемую версию драйверов для нашей видеокарты - у меня это версия 460

5) *sudo apt install nvidia-driver-460* - установка самих драйверов для видеокарты

6) *sudo apt install nvidia-utils-460* - установка некоторого дополнительного окружения для драйверов нашей видеокарты

7) *nvidia-smi* - проверка драйверов (установились ли они)

Информация о данной последовательности действий была собрана из большого количества различных мануалов и собственного опыта, поэтому первоисточники привести достаточно проблематично.

1.4. Логика работы моей лаборатории

Я написал лабораторный стенд, который позволяет количественно оценить возможности вашей машины при использовании CPU и GPU на максимум по перебору хэшей SHA1 для всех возможных 6-ти буквенных ключей (используется только английский строчный алфавит без цифер), что составляет около 300 млн различных паролей.

Приятной особенностью данного стенда является тот факт, что на GPU и на CPU запускается одинаковая реализация SHA1, то есть мы проверяем одинаковый код на разных архитектурах процессора, что делает результаты эксперимента более интересными.

1.5. Логика работы компонентов лаборатории

Пароли генерируются с помощью python-скрипта следующим образом:

В каждый созданный файл в папочку *./keys/* кладется количество ключей, которое не может превышать указанного в скрипте значения. Для примера я решил генерировать все возможные комбинации английского строчного алфавита длины 6.

После запуска bash-скрипта *./run.sh* из корневого каталога (который генерирует ключи), мы можем переместиться в каталог CPU или GPU, чтобы соответсвоенно запустить процесс восстановления парольной фразы по известному хэшу (запуская скрипт *./CPU/run.sh* или *./GPU/run.sh* соответственно) с помощью CPU или GPU.

Рассмотрим работу программы восстановления пароля на GPU:

Основными её компонентами являются скрипты *./GPU/main.cu* и *./GPU/info_about_GPU.cu*, а также подпапка *./sha1/*, в которой хранятся скрипты, отвечающие за расчет хэш-функции SHA1.

В файле *./GPU/info_about_GPU.cu* содержится функция, которая выводит информацию об имеющейся в ПК видеокарте и выдает на выходе значение, которое равно количеству нитей в сетке (в моём случае это число равно 1024).

Листинг программы *./GPU/main.cu* будет приведен ниже.

Кратко опишем логику его работы: он принимает на вход хэш в верхнем регистре (или любое другое значение, которое мы хотим получить, выполняя над данными нам ключами преобразование, которое мы запрограммировали - в нашем случае

это хэш - функция SHA1). Далее программа сама открывает по очереди все файлы из папки *./keys/*, ммапит их в оперативную память (для более быстрого доступа) и отдает их видеокарте, которая в свою очередь рассчитывает хэш для каждого ключа и сравнивает с переданным через командную строку значением. Если найдется нужный хэш, то видеокарта ставит в соответствующую ячейку выходного массива '1', если совпадения не произошло - то '0'. Если произошла ошибка, то элемент массива примет значение '2'.

Программа автоматически определяет количество файлов с ключами и количество ключей в каждом из них, а также размер этих ключей - поэтому все ключи в одном файле всегда должны иметь одинаковую длину и заканчиваться символом перевода строки.

Данную лабораторию можно использовать и для паролей разной длины - но при этом пароли с разными длинами кладутся в разные файлы. Так что можно генерировать пароли, подчиняясь этому правилу и правилу о том, что после каждого пароля должен стоять $'\backslash n'$.

Теперь обратимся к файлу *./GPU/sha1/sha1.cu*.

В нем находится код программы, который работает в каждом ядре видеокарты и рассчитывает хэш для определенного ключа, который вычисляется по смещению z. Для вычисления хэша я использовал реализацию из интернета, ссылку на которую я оставлю в листинге программы.

Содержимое программы из папки *./CPU/* почти полностью повторяет ранее описанные программы, только из них убраны модификаторы функций и добавлена *#pragma* и библиотека для работы с OpenMP (также в bash-скрипте изменен компилятор и добавлен необходимый флаг).

1.6. Результаты работы

Тестирование программ дало нам следующие результаты:

Время выполнения расчетов на GPU:

```
borkdog@borkdog:~/hash_bruteforce/GPU$ ./run.sh
[*] info about GPU:

Number of CUDA devices 1.
There is 1 device supporting CUDA
For device #0
Device name:                GeForce MX250
Total Global Memory:        2099904512
Total shared mem per block: 49152
Total const mem size:       65536
Warp size:                  32
Maximum amounts of threads per block: 1024
Maximum amounts of blocks in grid:    2147483647
Number of muliprocessors:   3

[*] start brute passwords

ctr = 0
ctr = 2
[+] valid key: njdhsb

ctr = 4
ctr = 6

real    3m36,790s
user    3m36,323s
sys     0m0,396s
```

Время, затраченное для расчетов на CPU в однопоточном режиме:

```
borkdog@borkdog:~/hash_bruteforce/CPU$ ./run.sh

[*] start brute passwords

ctr = 0
[+] valid key: djnvqb

ctr = 2
ctr = 4
ctr = 6

real    5m13,032s
user    5m12,463s
sys     0m0,504s
```

Время, затраченное для расчетов на CPU с распараллеливанием на OpenMP:

```
borkdog@borkdog:~/hash_bruteforce/CPU$ ./run.sh

[*] start brute passwords

ctr = 0
[+] valid key: djnvqb

ctr = 2
ctr = 4
ctr = 6

real    1m29,669s
user    11m23,593s
sys     0m0,472s
```

Как мы и предполагали, видеокарта работает быстрее процессора, когда мы не используем многопоточность; но как только мы начинаем использовать её, то чувствуется заметная разница в скорости выполнения вычислений - скорее всего код,

который я написал, не так сильно оптимизирован под GPU, как требуется. Также это может быть связано с тем, что видеокарта в моём ноутбуке просто изначально слабее, чем 8-ми ядерный процессор intel core i7, поэтому мы можем наблюдать превосходство по скорости CPU над GPU.

1.7. Плюсы использования GPU совместно с CPU, основываясь на результатах проведенной работы

Используя результаты работы программ, можно выделить следующие плюсы работы с GPU:

Можно использовать много устройств GPU в одном компьютере под управлением одного CPU, что повысит производительность вычислений на стандартном ПК (подсоединять дополнительные процессоры CPU сложнее, чем устройства GPU).

Также можно комбинировать перебор на CPU и GPU, чтобы использовать мощности железа на максимум (в таком режиме работы желательно использовать дополнительное охлаждение ноутбука или ПК, при этом (если это ноутбук) - использовать питание от сети, чтобы не расходовать ресурс батарейки).

1.8. Листинги программ

Скрипт ./GPU/main.cu:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <inttypes.h>
#include <string.h>
#include <dirent.h> /* для рассчета количества файлов */

#include "info_about_GPU.cu"
#include "sha1/sha1.cu"

FILE* f_log;

int main(int argc, char **argv){

  if (argc != 2){
    printf("неправильное количество параметров\n");
  }

  int i, j;
  int fd;
    struct stat st;
    char *ptr, *d_ptr, *output, *d_output, *d_valid, *key;
  char name[128];
```

```
  int ctr;
  long long int amount_of_keys;
  long long int output_size;
  int key_len;

  /* рассчитываем количество файлов в директории "/keys/" */
  struct dirent **namelist;
  int NUMBER_OF_FILES = (scandir("../keys/", &namelist, NULL, alphasort))-2;

  printf("[*] информация о GPU:\n\n");
  int MAX_TREADS = print_info_about_GPU();
  printf("\n[*] начинаю перебор паролей\n\n");

/* создание файла логов */
  f_log = fopen("./log.txt", "w");

/* по порядку открываем все файлы с ключами */

  for (ctr=0; ctr<NUMBER_OF_FILES; ctr++){

/* собираем название файла (программа рассчитывает на то, что название
будет менее 128-ми символов) */
    strncpy(name, "../keys/file", 12);
    sprintf((name+12), "%d", ctr);
    strcat(name, ".txt");
    fprintf(f_log, "открываю файл: %s\n", name);

    fd = open(name,O_RDWR);
    if (fd == -1){
        fprintf(f_log, "[-] не могу открыть файл %s\n", name);
        fclose(f_log);
        return 1;
    }

/* ммапим текущий файл в оперативную память (для быстрой работы с ним) */

    fstat (fd, &st); /* рассчет размера файла */
    ptr = (char*)mmap(NULL, st.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if ((long int)ptr == -1){
        fprintf(f_log, "[-] невозможен ммапинг файла  %s\n", name);
        fclose(f_log);
        return 1;
    }

    /* рассчет длины ключей для текущего файла */
    char * slash_n = strstr(ptr, "\n");
    if (slash_n != NULL){
      key_len = slash_n - ptr;
    } else {
      printf("[-] невозможно посчитать длину ключа\n");
      fprintf(f_log, "[-] невозможно посчитать длину ключа\n");
      /* аварийное завершение программы */
```

```
  cudaFree(d_ptr);
  cudaFree(d_output);
  cudaFree(d_valid);
  free(output);
  munmap(ptr, st.st_size);
  close(fd);
  fclose(f_log);
  return 1;
}

amount_of_keys = (int)(st.st_size/(key_len+1));
output_size = amount_of_keys*sizeof(char);

output = (char*)malloc(st.st_size);
cudaMalloc(&d_ptr, st.st_size);
cudaMalloc(&d_output, output_size);
cudaMemcpy(d_ptr, ptr, st.st_size, cudaMemcpyHostToDevice);

cudaMalloc(&d_valid, strlen(argv[1])*sizeof(char));
cudaMemcpy(d_valid, argv[1], strlen(argv[1])*sizeof(char), cudaMemcpyHostToDevice);

/* Хендл event'а */
cudaEvent_t syncEvent1, syncEvent2;
float gpuTime;

cudaEventCreate(&syncEvent1);    /* Создаем event1 */
cudaEventCreate(&syncEvent2);    /* Создаем event2 */
cudaEventRecord(syncEvent1, 0);  /* Записываем event1 */

/* запуск SHA1 */
long long int val = (amount_of_keys+MAX_TREADS-1)/MAX_TREADS;
sha1<<<val, MAX_TREADS>>>(d_ptr, d_output, key_len, amount_of_keys, d_valid, strlen(argv[1]));

cudaEventRecord(syncEvent2, 0);    /* Записываем event2 */
cudaEventSynchronize(syncEvent2);  /* Синхронизируем event */
cudaEventElapsedTime ( &gpuTime, syncEvent1, syncEvent2);
fprintf(f_log, "[*] время, затраченное на рассчеты GPU: %.2f milliseconds\n", gpuTime );
cudaEventDestroy(syncEvent1);
cudaEventDestroy(syncEvent2);

cudaMemcpy(output, d_output, output_size*sizeof(char), cudaMemcpyDeviceToHost);

for (i=0; i < amount_of_keys; i++){
  key = ptr + i*(key_len+1)*sizeof(char);
  if (output[i] == '1'){
    printf("[+] подходящий ключ: ");
    for (j=0; j<key_len; j++){
      printf("%c", key[j]);
    }
    printf("\n\n");

    /* запись подходящего ключа в файл логов */
```

```
        fprintf(f_log, "[+] подходящий ключ: ");
        for (j=0; j<key_len; j++){
          fprintf(f_log, "%c", key[j]);
        }
        fprintf(f_log, "\n\n");
      }
      if (output[i] == '2'){
        printf("[-] ошибка на ключе:\n");
        for (j=0; j<key_len; j++){
          printf("%c", key[j]);
        }
        printf("\n\n");

        /* запись ключа, на котором произошла ошибка, в файл логов */
        fprintf(f_log, "[-] ошибка на ключе:\n");
        for (j=0; j<key_len; j++){
          fprintf(f_log, "%c", key[j]);
        }
        fprintf(f_log, "\n\n");

        /* аварийное завершение программы */
        cudaFree(d_ptr);
        cudaFree(d_output);
        cudaFree(d_valid);
        free(output);
        munmap(ptr, st.st_size);
        close(fd);
        fclose(f_log);
        return 1;
      }
    }

    /*освобождаем занятую память */
    cudaFree(d_ptr);
    cudaFree(d_output);
    cudaFree(d_valid);
    free(output);

    munmap(ptr, st.st_size);
    close(fd);
    fprintf(f_log, "[+] закончили с файлом %s\n", name);

    if ((ctr % 2) == 0){
        printf("ctr = %d\n", ctr);
    }
  }

  fclose(f_log);
  return 0;
}
```

Скрипт ./GPU/info_about_GPU.cu:

```
/* источник https://gist.github.com/stevendborrelli/4286842 */
/* источник информации о сетке и о потоках внутри неё:
https://www.youtube.com/watch?v=kzXjRFL-gjo */

#pragma once
#include <stdio.h>

int print_info_about_GPU() {
    int deviceCount;
    cudaDeviceProp deviceProp;

    cudaGetDeviceCount(&deviceCount);
    printf("Количество CUDA девайсов %d.\n", deviceCount);

    for (int dev = 0; dev < deviceCount; dev++) {

        cudaGetDeviceProperties(&deviceProp, dev);
        if (dev == 0) {
            if (deviceProp.major == 9999 && deviceProp.minor == 9999) {
                printf("CUDA GPU-девайсы не обнаружены\n");
                return -1;
            } else if (deviceCount == 1) {
                printf("Обнаружен один девайс с поддержкой CUDA\n");
            } else {
                printf("Обнаружено %d устройств, поддерживающих CUDA\n", deviceCount);
            }
        }

        printf("Для девайса #%d\n", dev);
        printf("Название девайса:                              %s\n", deviceProp.name);
        printf("Общее количество памяти:                       %ld\n", deviceProp.totalGlobalMem);
        printf("Обзее количество разделяеммой памяти на блок: %ld\n",
                                                                 deviceProp.sharedMemPerBlock);
        printf("Количество статической памяти:                 %ld\n", deviceProp.totalConstMem);
        printf("Размер варпа:                                  %d\n", deviceProp.warpSize);
        printf("Максимальное количество потоков на блок:       %d\n", deviceProp.maxThreadsDim[0]);
        printf("Максимальное количество блоков в сетке:        %d\n", deviceProp.maxGridSize[0]);
        printf("Количество мультипроцессоров:                  %d\n",
                                                                 deviceProp.multiProcessorCount);
    }
    return deviceProp.maxThreadsDim[0];
}
```

Скрипт ./GPU/sha1/sha1.cu:

```
 #include "func_sha1.cu"

__global__ void sha1(char *text, char *output, uint32_t length,
                     long long int N, char *valid_hash, int hashlen){
  int z = blockIdx.x*blockDim.x + threadIdx.x;
  char *buf;
  int i, j;
  SHA1Context sha;
```

```
  char he[] = "0123456789ABCDEF";

  buf = (char *)malloc((hashlen+1)*sizeof(char));
  /* данная проверка здесь нужна для того, чтобы потоки с индексами,
  выходящими за границы файла с ключами, не выполняли работу */
  if (z < N){
    /* здесь мы высчитываем указатели на нужный пароль и считаем от него хэш */
    char *tmp_text = text + z*(length+1)*sizeof(char);
    char *tmp_output = output + z*sizeof(char);

    SHA1Reset(&sha);
    SHA1Input(&sha, tmp_text, length);

    /* переносим хэш из структуры в строковый массив-буфер */
    if (!SHA1Result(&sha))
    {
      tmp_output[0] = '2';
    }
    else
    {
      for(i=0; i<5; i++){
        for(j=0; j<8; j++){
          buf[i*8+7-j] = he[(sha.Message_Digest[i]&(0xf<<4*j))>>(4*j)];
        }
      }
      buf[40] = '\0';
    }
    /* сравниваем получаенный хэш с нужным нам */
     int flag = 1;
     for (i=0; i<hashlen; i++){
       if (buf[i] != valid_hash[i]){
         flag = 0;
       }
     }
     tmp_output[0] = '0'+flag;
  }
  free(buf);
}
```

Скрипт ./run.sh:

```
#! /usr/bin/env bash

./gen_key.py
```

Скрипт ./GPU/run.sh:

```
#! /usr/bin/env bash

rm ./a.out
nvcc ./main.cu
time ./a.out 2446F0A91A1E6F9995E64D782D51709075B82F54 # хэш от ключа njdhsb
```

Скрипт ./CPU/run.sh:

```
#! /usr/bin/env bash

rm ./a.out
gcc ./main.c -fopenmp
time ./a.out 85F734A549BCA139E4ECEF71AFE94C5C58AAD03B # хэш от ключа djnvqb
```

Полный набор исходных кодов программ для лаборатории можно найти на моём github:

https://github.com/Respman/CUDA_for_crypto