MP4: Brute-Force Key Search

Submission Instructions. For this assignment, you will parallelize the same code using **both** CUDA **and** OpenACC. This assignment is worth 40 points total. Submission instructions are at the end of the assignment.

WebCode – Online Editor – http://gpu.cse.eng.auburn.edu:8000/

Again, **please use WebCode to write (and run) your code for this assignment**. If WebCode does not work correctly, or if you have problems/complaints/suggestions, e-mail both William Hester <weh0008@auburn.edu> and Jeff Overbey <joverbey@auburn.edu> so we can address them.

Remember: you must be on Auburn's network or on the VPN to access WebCode.

Background

Encryption

The goal of **encryption** is to encode data so that it can be read by an intended recipient but not by an eavesdropper. Often, the sender and the intended recipient share a (secret) **key**, a value that is used to encrypt and decrypt the message. Many algorithms encrypt 32 to 128 bits of data using 128- or 256-bit keys.

Three of the most widely used encryption algorithms are AES (the Advanced Encryption Standard, which became a U.S. government standard in 2001), DES (the Data Encryption Standard, its 1979 predecessor, which is now considered insecure), and RSA (named after its inventors: Rivest, Shamir, and Adleman).

The Tiny Encryption Algorithm (TEA) was designed by David Wheeler and Roger Needham and first published in 1994. It it not as secure as AES or RSA; it has several known weaknesses, including one that led to a method for hacking the Xbox (which used it inappropriately). However, TEA requires very little code to implement.

The TEA Encryption Algorithm

The following C code implements the TEA encryption algorithm. You do **not** need to understand exactly how or why it works; you only need to be able to **call** this function correctly. So, pay attention to the function signature (the first line), and ignore most of the rest of it:

```
void encrypt(uint32_t *data, uint32_t *key) {
    uint32_t v0=data[0], v1=data[1], sum=0, i;           /* set up */
    uint32_t delta=0x9e3779b9;                           /* a key schedule constant */
    uint32_t k0=key[0], k1=key[1], k2=key[2], k3=key[3];   /* cache key */
    for (i=0; i < 32; i++) {                             /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                                    /* end cycle */
    data[0]=v0; data[1]=v1;
}
```

Notice that the `encrypt` function takes two arguments: the data to encrypt and the key. Both are passed as arrays of `uint32_t`'s (or, more precisely, pointers to `uint32_t`'s).

```
    void encrypt(uint32_t *data, uint32_t *key) {
        ...
    }
```

There is also a `decrypt` function, which is similar.

```
    void decrypt(uint32_t *data, uint32_t *key) {
        ...
    }
```

The TEA encryption algorithm encrypts **64 bits** of data at a time and uses a **128-bit** key. When it finishes encrypting, the encrypted data will also be 64 bits, just like the input data.

In the C implementation above, the 64 bits of data and the key are stored as arrays of `uint32_t`'s.

```
uint32_t data[2] = { 0xDEADBEEF, 0x0BADF00D };               // 64 bits of data
uint32_t key[4]  = { 12345678, 13572468, 3444555, 1222333 };  // 128-bit key
```

The `encrypt` and `decrypt` functions operate *in-place*; that is, they modify the values in the data array. E.g.:

```
printf("The original values are %u %u\n", data[0], data[1]);
encrypt(data, key);     // Replace data[0] and data[1] with the encrypted values
printf("The encrypted values are %u %u\n", data[0], data[1]);
decrypt(data, key);     // Replace data[0] and data[1] with the decrypted values
printf("Now we should get the original values again: %u %u\n", data[0], data[1]);
```

The above code produces the following output:

```
The original values are 111222333 444555666
The encrypted values are 12309543 3426093923
Now we should get the original values again: 111222333 444555666
```

If you want to try it yourself, the above example is available in a template called **TEA Example** in WebCode.

A Brute-Force Key Search

Suppose you are evil, and you want to figure out someone's key.

Let's assume you have somehow discovered that, if the data { 0xDEADBEEF, 0x0BADF00D } is encrypted with their key, it will encrypt to { 0xFF305F9B, 0xB9BDCECE }. Then, there is a simple, extremely inefficient, not-very-clever way to figure out their key: Try encrypting the data { 0xDEADBEEF, 0x0BADF00D } with every possible key, and see which key gives you { 0xFF305F9B, 0xB9BDCECE }.

Since TEA uses a 128-bit key, this is a bit impractical, since there are 2^{128} possible keys (about 3.4×10^{38}).

What if the key were only 28 bits? Then there would be only 2^{28} = 268,435,456 possible keys. A CPU can search through this many keys in about 30 seconds.

To explore this idea, let's use the same TEA code as before, but assume the key has the form { *k*, *k*, *k*, *k* }, where *k* is some 28-bit value (i.e., a value between 0 and 0x0FFFFFFF, inclusive).

The **MP4** template in WebCode does exactly this. It searches through every possible 28-bit key in this form, reporting the one that produced the correct result. Your goal is to parallelize it to run on the GPU.

Assignment

For this assignment, you will parallelize the brute-force key search using both CUDA and OpenACC.

- OpenACC has a directive called `#pragma acc routine`. You will need to add this to the `encrypt` function to be able to invoke it from an OpenACC parallel region. (Read about it in the OpenACC specification if you want.)

- Log into WebCode. Starting from the **MP4** template, create **two** new projects—one OpenACC, one CUDA.

Which one you do first will be the **opposite** of what you did for the previous MP:

- If your last name (family name) starts with A–L, please parallelize the code using **OpenACC first**. Then, parallelize it using **CUDA afterward**, i.e., after you have completed the OpenACC code.

  OpenACC First: Alkofahi, Almohaishi, Bordelon, Brooks, Calvert, Caufield, Farmer, Feist, Gong, Hancock, Hester, Hoover, Jennings, Kilgore, Li

- If your last name (family name) starts with M–Y, please parallelize the code using **CUDA first**. Then, parallelize it using **OpenACC** after you have completed the CUDA code.

  CUDA First: Mukhopadhyay, O'Rourke, Perreault, Pierce, Price, Ravipati, Rowley, Sawyer, Scott, Sprunger, Stewart, Tang, W. Wang, X. Wang, Y. Wang, Wei, Yan

Hints

- Focus on parallelizing the loop that tries all 268,435,456 possible keys. You probably won't get any useful speedup trying to parallelize the `encrypt` or `decrypt` functions (if you look at their code, can you see why?). Feel free to rewrite the main function (in particular, the loop that tries every possible key) however you need to; just make sure you're checking for (and getting) the correct result at the end!

- The *for* contains a *break* statement to terminate the loop when it finds the correct key. You don't need to maintain this behavior on the GPU; as long as it finds the correct key, you're fine.

- Try to spend a bit of time trying to optimize your code. How fast can you make it?

Submission Instructions

After you have finished writing your code, you will need to download it from WebCode and upload your code to Canvas. To download a file, click the "download" icon next to its filename in WebCode.

- **Submit three files in Canvas:**
  - Your **CUDA** code (mp4.cu)
  - Your **OpenACC** code (mp4.c)
  - A **text file** (mp4.txt) with *brief* answers to the following questions:
    1. Which version did you write first: CUDA or OpenACC?
    2. Which version was *easier* to write, OpenACC or CUDA? Why?
    3. If you needed to parallelize code like this in the future, would you prefer to use CUDA or OpenACC?
    4. Did you try to optimize your CUDA code? Your OpenACC code? If so, what did you try? What worked, and what didn't?