Fredrik Robertsén

Implementing a DES brute force cracker on an NVidia Fermi GPU

Turku Centre *for* Computer Science

TUCS Technical Report
No 1057, August 2012

Implementing a DES brute force cracker on an NVidia Fermi GPU

Fredrik Robertsén

Åbo Akademi University, Department of Information Technologies

TUCS Technical Report
No 1057, August 2012

Abstract

This report describes an optimized way of implementing a brute force known plaintext DES cracker. Various techniques for optimizing the basic DES algorithm are discussed. And implementation techniques are given for both CPU-based and Nvidia GPU-based implementations.

Keywords: Data encryption standard, CUDA, Fermi.

1. Introduction

The program described in this report was created as a part of a course on cryptography and network security lectured in 2012 at Åbo Akademi University, Department of Information Technologies (http://users.abo.fi/ipetre/crypto/). In this course we were set a number of challenges to break certain encryption schemes. Some of these challenges where based on the DES encryption scheme. The CPU implementation was mainly created for solving an alphanumeric key. The GPU implementation was an attempt to break a full 56-bit DES key. This did however prove to be impossible to achieve given the limited hardware and time to run the program. So instead a 49-bit key was broken.

The attack conducted was a known plaintext attack. That is when both the input and output are known and only the key needs to be found. The following implementation is optimized for a known plaintext attack, where we know that the input is a text message and we know the text. However with some modifications it can also be used without knowing the text of the input message.

Please note that the code presented in this report is not thoroughly tested and not guaranteed to work correctly. It has been tested in the two challenges and some other test cases but not more than that.

1.1. The DES algorithm: the key schedule

The material in this and the next section is based on the FIPS 46-3 (1) standard published in 1999 describing the DES standard.

The purpose of the key schedule is to preprocess the key before it is used to encrypt or decrypt the message. As an input the key schedule takes a 64-bit key. From this key it uses 56-bits to create 16 sub keys each with a length of 42-bits. Each of these sub keys are then used by the message schedule to do the actual encryption with.

The first step in the key schedule is to permutate the given 64-bit key, in this step the key is also shortened from 64-bits to 56-bits by ignoring the least significant bit in each byte. The values used for the key permutation is shown in Table 1: . The table works by taking the 57-bit of the input key and placing it in the first position of the permutated key.

Table 1: Permuted choice 1 (1)

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Next the key is split into two pieces, each piece being 28-bits long. Each of these pairs of pieces is then left shifted by varying amount 16 times the results of each shift are then stored and later become the sub keys. The total number of left shifts used for each shift is shown in Table 2: Number of left shifts.

Table 2: Number of left shifts (1)

Sub key	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Left shift	1	2	4	6	8	10	12	14	15	17	19	21	23	25	27	28

The final step of the key schedule is to combine each the two pieces in each sub key into one 56-bit segment again. After this each sub key is again permutated as shown in Table 3: .

Table 3: Permuted choice 2 (1)

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

The entire key schedule process is shown in Figure 1: The key schedule.

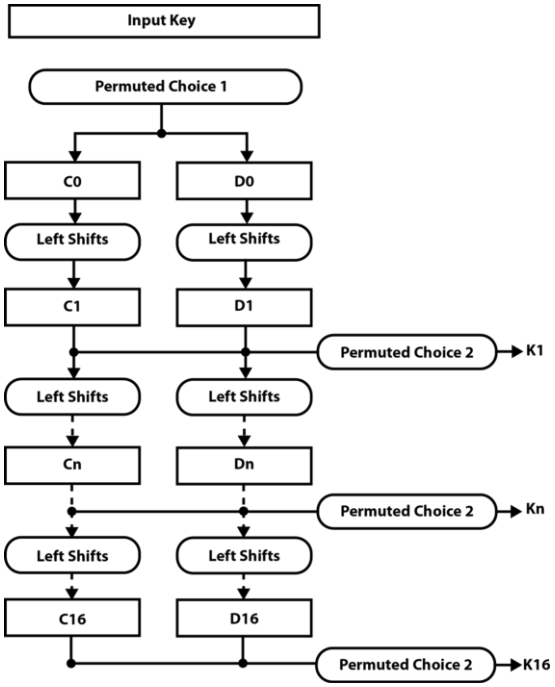


Figure 1: The key schedule (1)

1.2. The DES algorithm: the message schedule

The message schedule is where the actual encryption or decryption takes place. The DES cipher is a block cipher, meaning it takes one block of the input and does the encryption on that block and then moves on to the next block. It takes a 64-bit block from the input message and preforms the encryption on that block, once it is finished it moves on to the next block.

The message schedule begins by a permutation, taking a 64-bit part of the message and rearranging the bits according to the values in Table 4: Initial permutation.

Table 4: Initial permutation (1)

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

After this the message is split into two 32-bit pieces one left part and one right part.

Next comes the actual message encoding part of the algorithm. This part consists of 16 iterations, one iteration for each of the sub keys. For message encryption the keys are used in ascending order starting at key 1 whereas for decryption the keys are used in descending order starting from key 16.

Each of the iterations can be described by the following equations

$$Left_n = Right_{n-1},$$
$$Right_n = Left_{n-1} \oplus f(Right_{n-1}, Key_n).$$

So the left value is assigned the previous right value, and the right value is assigned the previous left value combined with the result from the function f in an exclusive or operation. The f function takes as input the current sub key and the previous right value and produces a 32-bit result.

Table 5: Expanding permutation (1)

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

The first step in the f function is an expanding permutation according to Table 5: Expanding permutation, this expands the 32-bit input to a 48-bit segment. Next this segment is combined with the sub key in an XOR operation, after this the substitution with the values from the S-box corresponding to the segment number are performed.

Table 6: S-box 1 (1)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

There are eight S-boxes, each one contains 64 segments with a length of 4-bits, the first S-box, S1 is shown in Table 6: S-box 1. These segments are arranged in four rows each with a length of 16.

The 48-bit output segment from the expanding permutation is split into eight segments, each being 6-bits long. The first and last bit is then used to determine what row of an S-box and the middle 4-bits are used to

determine the column. The result of the S-box substitution is a 32-bit segment. The final step in the f function is another permutation according to Table 7: P permutation.

Table 7: P permutation (1)

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

After all the 16 iterations the left and right parts are joined together to form a 64-bit segment again. After this the final permutation is done according to Table 8: Inverse initial permutation. Now the message is done.

Table 8: Inverse initial permutation (1)

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

The block diagram for the f function is shown in Figure 2: The f function.

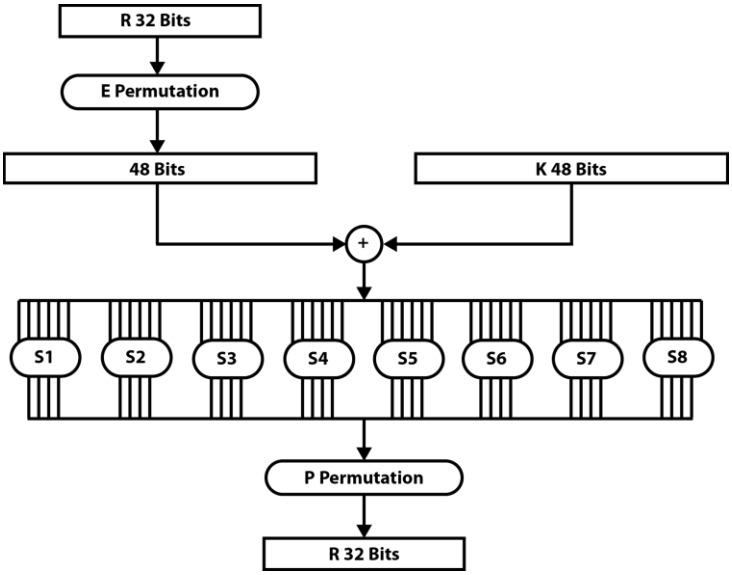


Figure 2: The f function (1)

The entire message schedule process is shown in Figure 3: The message .

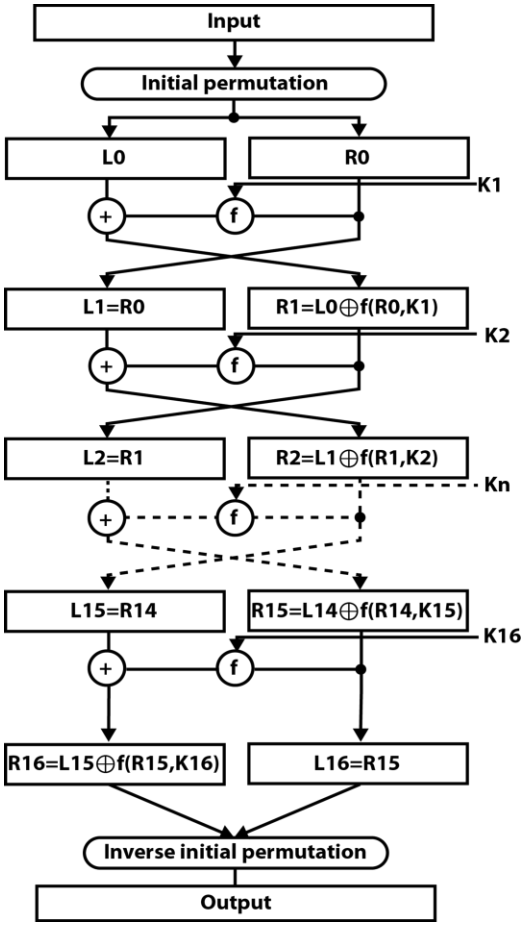
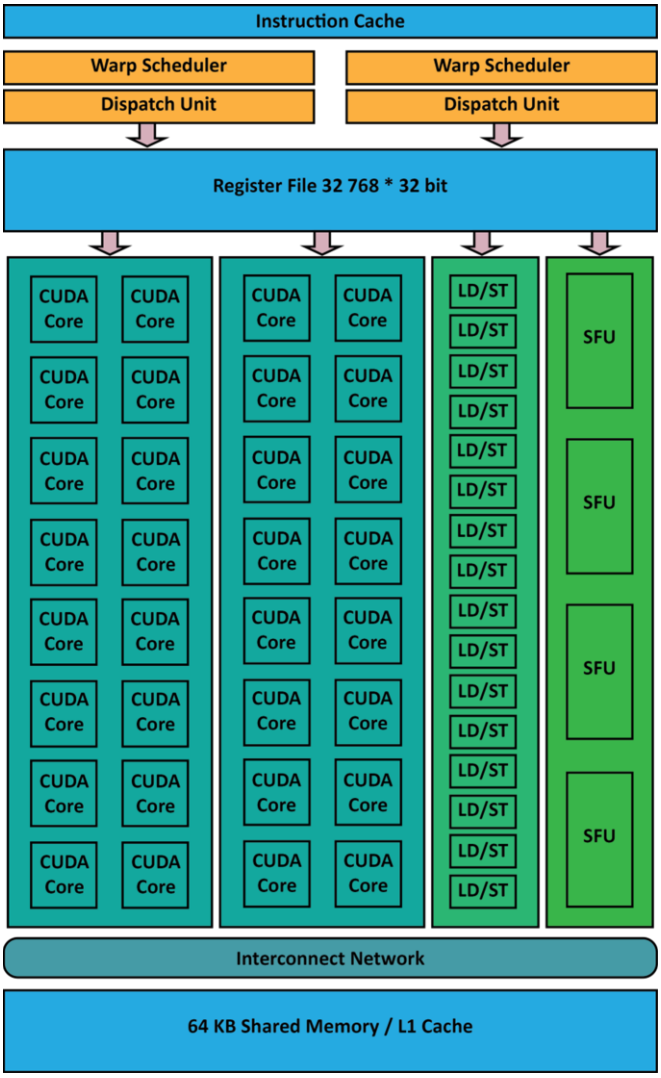


Figure 3: The message schedule (1)

2. The NVidia Fermi GPU architecture

The GPU in an NVidia graphics or compute card based on the Fermi architecture is based on the SIMT architecture where many cores execute the same instruction on many data cores concurrently. The instructions have to be the same but the data can be different. The Fermi core consists of streaming multiprocessors each of which contain 32 CUDA cores as shown in **Error! Reference source not found.** (2).



Error! Reference source not found. **(2)**

Each streaming multiprocessor has a 16 KB or 48 KB L1 cache. The size of the cache can be altered at kernel run time. All multiprocessors share the same 768 KB L2 cache (2). In addition to the L1 cache each streaming multiprocessor also has a private shared memory that only it can access. The size of this memory varies based on the size of the L1 cache, with a

small L1 cache the shared memory is 48 KB and with a large L1 cache 16 K (3).

When threads are scheduled to run on the GPU they are grouped into thread blocks, each thread block is a three dimensional construct. The maximum size of a thread block is 1536 treads. These thread blocks are then grouped into a grid again being a three dimensional construct. The maximum size of a grid is 64535 (4). When threads are run a whole thread block is allocated to one of the streaming multiprocessors. The thread block is split into warps of 32 threads, each of these warps are then run in parallel (4).

For maximum performance all threads within one warp should take the same instruction path. If the warp contains divergent branches these instructions are scheduled for serial execution and all the cores that do not take the branch are essentially idle.

The shared memory in each streaming multiprocessor is shared among all the threads in the same thread block. The read latency from the shared memory is about the same as from the L1 cache whereas the read latency from global memory is 400-600 cycles (5). The only limitation is that only threads on the GPU can read and write from the shared memory. The host system has no control over the data in the shared memory.

3. The CPU implementation

For the CPU implementation Intel Cilk+ (6) was used to make the program parallel. The general idea of the CPU implementation was to have a number of threads each looping through a set of keys. It would take this key and the input message, encrypt the input message with the key and then check if the cipher text is the same as the input cipher text. The program would run on the first 64-bits of the input message. If it found a key so that the cipher texts matched it would try the same key for the next 64-bits of the input. If it found that these cipher texts matched as well the key was probably found. If the cipher texts did not match it would simply move on the next key in the set.

3.1. The key schedule and key generation

3.1.1. Theory

In a naïve implementation the key schedule needs to be run once for each new key. The key schedule is however easy to optimize given a sufficient amount of fast memory.

The main idea behind optimizing the key schedule is that a certain bit of the input key always gets copied to the same bit in a given sub key. For instance the most significant bit gets copied to bit 28 in sub key one and bit 38 in sub key two. This means that a completely optimized key schedule could be just copy bit X to position Y in sub key N for all the bits in all the sub keys. However there is an even faster solution available if there is a sufficient amount of fast memory available, such as the L2 and L1 caches on the processor.

The input key can be seen as eight segments, each containing 7-bits. If all permutations of bits in the first segment are cycled through, leaving all the bits in the other segments as 0. After this the key schedule would be run with the partial key as input the result will be all sub keys that can be generated by the first segment. These values are then stored. Now if this process is repeated for all segments the results are all the pieces needed to construct all the sub keys for any input key. This is done by taking the piece corresponding to character one at position one and combining it with the one for character two at position two.

By doing the key schedule in pieces and then combining the pieces to create the final sub keys results in a fewer number of key schedule functions that need to be run. A normal approach, for a 56-bit key, that runs the entire key schedule for each key results in the key schedule being run 2^{56} times. Using this optimization the key schedule is only run $56 * 8$ times. There is however some computation needed to combine the sub keys together and also some to read them from memory.

There are further optimizations that can be done to improve the reading from memory and combining of the keys. If the keys are cycled through with nested *for* loops only one part of the key will be changed. So there is no need to regenerate the whole key each time. There are two ways for solving this: (i) clear the bits associated with the current segment (ii) store the segment with certain bits cleared outside the loop.

3.1.2. Implementation

Code 1: the key space generator shows a simple implementation of the key schedule. A simple implementation is sufficient when the pieces of the key are pre-generated.

Code 1: the key space generator

```
void keySpaceGeneratorOpt(u64 ***keySpacePre, char *charSet, int size)
{
    for (int place = 0; place < 8; place++)
        for(int ch = 0; ch < size; ch++)
        {
            u64 inputKey = 0;
            u64 premutedKey = 0;
            u32 low = 0;
            u32 high = 0;
            if (place != 7)
            {
                inputKey = (((u64)charSet[ch]) << (8*place))&
~(0xffffffffffffffff<<(8*(place+1)));
            }
            else
            {
                inputKey = (((u64)charSet[ch]) << (8*place));
            }
            for(int i = 0; i < 56; i++)
            {
                copyBit(inputKey, 63-pc1[i], premutedKey, 55-i);
            }
            low =premutedKey;
            low = low << 4;
            low = low >> 4;
            high = premutedKey>>28;
            for (int i =  0; i < 16 ; i++)
            {
                u64 mergedKey = 0;
                low = (((low << keyRotate[i])<<4)>>4) | low >> 28-keyRotate[i];
                high = (((high << keyRotate[i])<<4)>>4) | high >> 28-keyRotate[i];
                mergedKey = (((u64)high << 28) | low);
                keySpacePre[ch][place][i] = 0;
                for (int j = 0; j < 48 ; j++)
                {
                    copyBit(mergedKey, 55-pc2[j], keySpacePre[ch][place][i], 47-j);
                }
            }
        }
}
```

In the code the *charSet* input array is all the possible characters used in the key and *pc1* is an integer array containing the indexes for the K permutation for the input key. Likewise *pc2* contains the bits that should be read from the sub key. Key rotate contains the steps the each part of the sub key should be shifted. And *keySpacePre* is a three dimensional array where the pieces of the key is stored, the first dimension is the current permutation, the second is the piece of the key and the last is the current sub key. The bits are transferred in the key using the *copyBit* macro. It is a simple macro that takes as input the value to read bits from, the place to read from, the value to store the bits to, and the place where the value should be stored.

To parallelize the program the key space was divided into buckets and each thread got one bucket to work with, when it was done with the current bucket it got another one. For simplicity the key space was divided into $4 * 128$ buckets, each representing a unique four 7-bit segment part of the key. In theory all these buckets could be run in parallel. However due to the limited number of cores available the Cilk runtime will further group together these buckets and schedule them on the cores.

The function that loops trough the contents of each bucket in presented in Code 2: how the key is generated for each message.

Code 2: how the key is generated for each message

```
void parallelDESBruteforceRangeWorker(char *charSet, u64 ***keySpacePre, int size,u64
R1, u64 L1,u64 plainR1, u64 plainL1, long long int start, long long int* factors)
{
    long long int current = start;
    int decoded;
    int l3 = ((start%(factors[4]))/(factors[3]));
    int l2 = ((start%(factors[5]))/(factors[4]));
    int l1 = ((start%(factors[6]))/(factors[5]));
    int l0 = ((start%(factors[7]))/(factors[6]));
    u64 *keySet;
    keySet = new u64[16];
    keySet[0] = keySpacePre[13][4][0]|
        keySpacePre[12][5][0]|
        keySpacePre[11][6][0]|
        keySpacePre[10][7][0];
    ...
        keySet[15] = keySpacePre[13][4][15]|
        keySpacePre[12][5][15]|
        keySpacePre[11][6][15]|
        keySpacePre[10][7][15];
    for (int l4 = 0; l4 < size; l4++)
    {
        keySet[0] = keySpacePre[14][3][0]|
            (keySet[0] & 0xffffddfbf7d7fdff );
        ...
            keySet[15] = keySpacePre[14][3][15]|
            (keySet[15] & 0xfffffefcfbfeeffe );
        for (int l5 = 0; l5 < size; l5++)
        {
            keySet[0] = keySpacePre[15][2][0]|
```

```
                (keySet[0] & 0xfffffef7efdffdf );
            ...
                keySet[15] = keySpacePre[15][2][15]|
                (keySet[15] & 0xffffddfbf7d7fdff );
            for (int l6 = 0; l6 < size; l6++)
            {
                keySet[0] = keySpacePre[l6][1][0]|
                    (keySet[0] & 0xffffb7ffffefbffd );
                ...
                    keySet[15] = keySpacePre[l6][1][15]|
                    (keySet[15] & 0xfffffef7efdffdf );

                for (int l7 = 0; l7 < size; l7++)
                {
                    keySet[0] = keySpacePre[l7][0][0]|
                        (keySet[0] & 0xffffeeff9ffff6fb );
                    ...
                        keySet[15] = keySpacePre[l7][0][15]|
                        (keySet[15] & 0xffffb7ffffefbffd );
                    decoded = messageSheduleOpt(keySet, R1, L1, plainR1, plainL1);
                    if (decoded)
                    {
                        //possible key found
                    }
                }
            }
        }
    }
    free(keySet);
}
```

The code begins by calculating which characters the start of the key is based on the bucket number. The *factors* array contains pre calculated factors based on the number of different character possibilities used in the key. Next it creates the key set for the start of the key; this set is the same for the entire bucket. After that it begins looping trough the remaining characters in the key. It first clears the bits associated with the current piece; this is all the bits the current part of the key can change. Once all this is done it fetches the new piece from memory and using bitwise OR to combine it with the rest of the sub key. Exactly how the *messageSheduleOpt* function works and how the parameters for it are generated will be presented later.

3.2. The message schedule

3.2.1. Theory

Since the attack that was conducted was a known plaintext attack, there are a lot of optimizations that can be done to the message schedule. The first step in the message schedule is to permute the input, in this case the cipher text. This permutation will be the same for all keys, so since the input does not change, it is a calculation that can be performed once and then stored to memory. Then when a thread needs the data it can simply read the stored value. The second step is to split the cipher text into two

32-bit parts, this can also be performed once and then read from the memory.

The first time the pair of cipher text pieces is used in the algorithm it is when they are expanded to 48-bits by an expanding permutation. Since the input is always the same this expansion can also be done once and then reused for all keys. A similar thing can be done with the plain text. To generate the plain text the two 48-bit parts at the end of the permutations are contracted to 32-bits each and then joined together to form a 64-bit message, this message is permuted one last time before it is returned as the final plain text. Since the plain text is known however the reverse operations can be performed for the known plain text. It is first permuted in reverse order to what the normal permutation is done, it is then split into two 32-bit parts. Each of these parts are then expanded to 48-bits. Now these two parts are stored and can be used each time that a check for a valid message is decrypted from the cipher text.

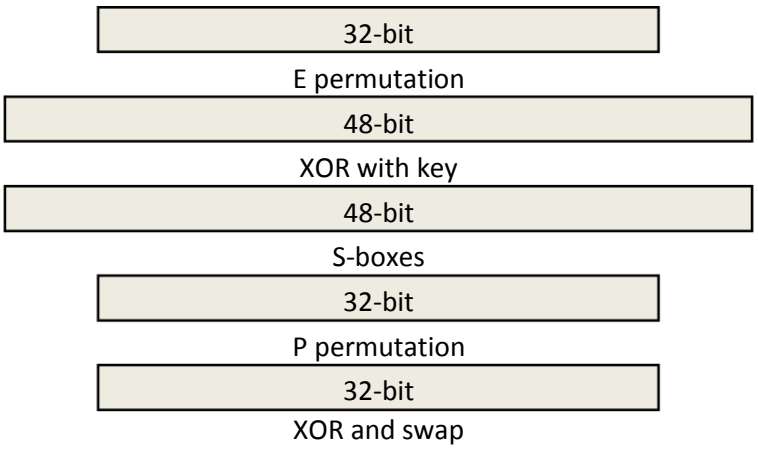


Figure 4: normal iterations

The iteration part of the message schedule has a lot of room for improvements. The original implementation, shown in Figure 4: normal iterations, takes a 32-bit part and expands and permutes it to 48-bits then does an exclusive or operation with a key. This 48-bit segment is then fed into the S-boxes where it is once again contracted to 32-bit. The new 32-bit segment is first permuted once again and then combined with the previous result through an exclusive or operation. The expansion and contractions are completely unnecessary, if the S-boxes are modified, as seen in Figure 5: non contracting iterations, to take a 48-bit input and also output a 48-bit value the whole iteration process can be rewritten to work on 48-bit values.

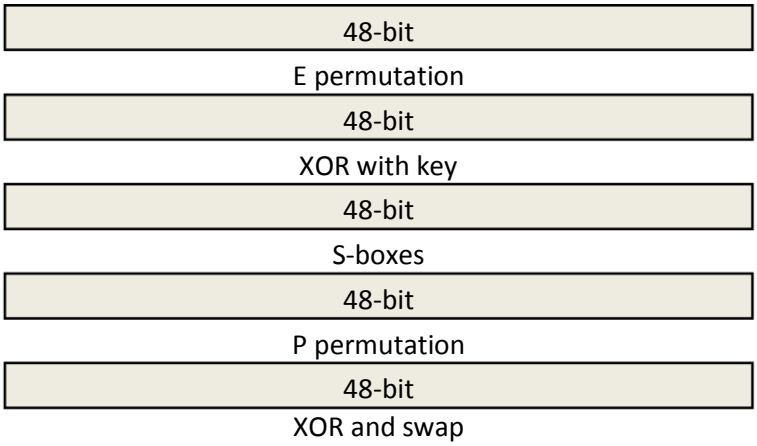


Figure 5: non contracting iterations

With this modified 4-bit version of the iterations a lot of operations can be combined into one. The input messages are permutated they are fed into the message schedule so the first two permutations can easily be done once and stored. With the expanded 48-bit version of the S-boxes the operations inside each one of the iteration can be rearranged. This means that both the E permutation and the P permutation can be included in the S-boxes. The result is far fewer operations, as shown in Figure 6: combined S-boxes, and a significant performance increase.

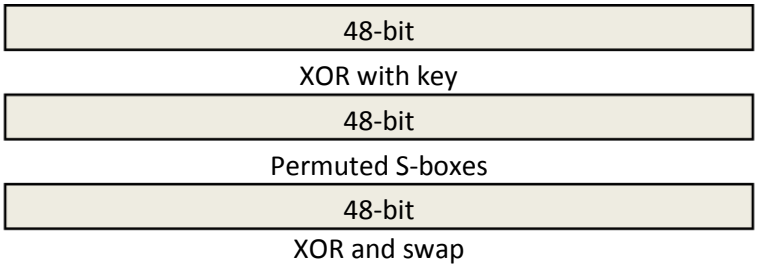


Figure 6: combined S-boxes

How the data in the S-boxes are accessed can be further improved by rearranging the data in the S-boxes. The original specifications have two dimensional S-boxes, and the way the indexes are calculated involve taking the first and last bits of a 6-bit segment and using them as one index and the middle 4-bits as another index. This introduces unnecessary bit shift operations. The s-boxes can be rearranged into a one dimensional array. And then an unaltered 6-bit segment can be used as the index for the array, thus saving a lot of extra calculations. Code describing how to generate the improved S-boxes and further explanation how they are done is presented in the next section.

In most cases it is possible to skip the final step of the iteration. After the second to last iteration is finished half of the decrypted message is already

done, the only thing done to it after this is a final permutation. Since the plain text was already inversely permutated and stored, it can easily be compared to the result of the second to last permutation. If both messages match the last iteration is performed. If they do not match the program simply moves on to the next key. If both parts of the message match it is highly likely that a key has been found, to be sure a second 64-bit block from the cipher text should be checked using the same key.

3.2.2. Implementation

Before the message schedule can be called the parameters for it needs to be generated. To run it needs the pre permutated cipher text and the inversely permutated plain text, both split into two 48-bit segments. This is done once before the program starts cycling trough the keys. A simple implementation for both is thus sufficient since it will only be done once.

Code 3: the plain text permutation

```
u64 plaintext;
... get the plain text
u64 permutedPlain = 0;
for (int i = 0; i < 64; i++)
{
    copyBit(plaintext, i, permutedPlain, plainPerm[i]);
}
u32 splitR1 = permutedPlain;
u64 plainR1  = 0;
plainR1 = ((u64)splitR1&0xf)<<1 |
          ((u64)splitR1&0xf0)<<3 |
          ((u64)splitR1&0xf00)<<5 |
          ((u64)splitR1&0xf000)<<7 |
          ((u64)splitR1&0xf0000)<<9 |
          ((u64)splitR1&0xf00000)<<11 |
          ((u64)splitR1&0xf000000)<<13 |
          ((u64)splitR1&0xf0000000)<<15;
u32 splitL1 = permutedPlain >> 32;
u64 plainL1 = 0;
plainL1 = ((u64)splitL1&0xf)<<1 |
          ((u64)splitL1&0xf0)<<3 |
          ((u64)splitL1&0xf00)<<5 |
          ((u64)splitL1&0xf000)<<7 |
          ((u64)splitL1&0xf0000)<<9 |
          ((u64)splitL1&0xf00000)<<11 |
          ((u64)splitL1&0xf000000)<<13 |
          ((u64)splitL1&0xf0000000)<<15;
```

The plain text permutation shown in Code 3: the plain text permutation takes as input the plaintext as a 64-bit unsigned integer. Since the implementation written for this task takes the cipher text as an input and then decrypts it, the plain text is what is fed into the last of the iterations. For that the plain text has to be permuted in reverse to how the last permutation of the message is done in the original DES specifications. The result should be what the plain text looks like at the end of the permutations. Since the output from the permutations is two 48-bit segments the inversely permuted plain text is split into two segments and

then expanded to 48-bits. The bits used as padding for the expansion can be ignored in the comparison so their exact values are irrelevant, however in this implementation zeroes are used. The array *plainPerm* includes the indexes for what bits should be swapped where. *PlainL1* and *plainR1* are then passed on to the message schedule.

As stated earlier the input to what remains of the DES message schedule should be the cipher text permuted, split into two and expanded. How this was accomplished is shown in Code 4: the cipher text permutation.

Code 4: the cipher text permutation

```
u64 cipherText;
... get the cipher text
for (int i = 0; i < 64; i++)
{
    copyBit(cipherText, i, permutedCipher1, cipherPerm[i]);
}
u32 R1 = permutedCipher1;
u64 R1exp = 0;
u32 L1 = permutedCipher1 >> 32;
u64 L1exp = 0;
R1exp =(((((uint64_t)0x1)&R1))<<47)|
       (((((uint64_t)0xF8000000)&R1))<<15)|
       (((((uint64_t)0x1F800000)&R1))<<13)|
       (((((uint64_t)0x1F80000)&R1))<<11)|
       (((((uint64_t)0x1F8000)&R1))<<9)|
       (((((uint64_t)0x1F800)&R1))<<7)|
       (((((uint64_t)0x1F80)&R1))<<5)|
       (((((uint64_t)0x1F8)&R1))<<3)|
       (((((uint64_t)0x1F)&R1))<<1)|
       (((((uint64_t)0x80000000)&R1))>>31);

L1exp =(((((uint64_t)0x1)&L1))<<47)|
       (((((uint64_t)0xF8000000)&L1))<<15)|
       ... same as above
       (((((uint64_t)0x1F)&L1))<<1)|
       (((((uint64_t)0x80000000)&R1))>>31);
```

The input is again stored a 64-bit unsigned integer. The cipher text is then permuted, the *cipherPerm* array contains the indexes of the bits that needs to be swapped. After that the cipher text is split in the same way as the plain text was split. Then the cipher text is expanded, this time the padding is done with the cipher text itself, in the same way that the DES implementation describes it. The output, stored in *R1exp* and *L1exp* is then the values passed on to the message schedule.

The message schedule shown in Code 5: the CPU message schedule gets called once for each key, this is done after the key has been generated. The basic operation of the message schedule is to decrypt the provided cipher text, then compare it to the provided plain text, the reason why the message schedule is doing decrypting instead of encrypting is to make it easier to change to an unknown plaintext attack.

Code 5: the CPU message schedule

```
inline int messageSheduleOpt(u64 *keySet, u64 R, u64 L, u64 plainR1, u64 plainL1){
    u64 KxorER;
    u64 temp;
    u64 T;
    // iteration 15
    KxorER = keySet[15] ^ R;
    temp = SB1[(0x3F)&(KxorER>>42)];
    temp |= SB2[(0x3F)&(KxorER>>36)];
    temp |= SB3[(0x3F)&(KxorER>>30)];
    temp |= SB4[(0x3F)&(KxorER>>24)];
    temp |= SB5[(0x3F)&(KxorER>>18)];
    temp |= SB6[(0x3F)&(KxorER>>12)];
    temp |= SB7[(0x3F)&(KxorER>>6)];
    temp |= SB8[(0x3F)&(KxorER)];
    T = L^temp;
    // iteration 14
    ...
    // iteration 1
    ...
    if (plainR1 != (R&(0x000079e79e79e79e))){
        return 0;
    }
    // iteration 0
    KxorER = keySet[0] ^ R;
    temp = SB1[(0x3F)&(KxorER>>42)];
    temp |= SB2[(0x3F)&(KxorER>>36)];
    temp |= SB3[(0x3F)&(KxorER>>30)];
    temp |= SB4[(0x3F)&(KxorER>>24)];
    temp |= SB5[(0x3F)&(KxorER>>18)];
    temp |= SB6[(0x3F)&(KxorER>>12)];
    temp |= SB7[(0x3F)&(KxorER>>6)];
    temp |= SB8[(0x3F)&(KxorER)];
    T = L^temp;
    if (plainL1 == (T&(0x000079e79e79e79e))){
        return 1;
    }
    return 0;
}
```

The input to the message schedule is the current set of 16 sub keys and the preprocessed plain and cipher text. Both in the form of two 48-bit parts, however they are stored in 64-bit unsigned integers with the extra bits set to zero. As the implementation is doing decrypting it will start with sub key number 15 and cycle down to sub key 0. The current sub key is fetched from the *keySet* array, this is then bitwise xored with the right part of the message and this result is stored into the *KxorER* value.

Next is the s-box pass, thanks to the rearranged s-boxes the value in *KxorER* only needs to be split into eight 6-bit pieces. Each 6-bit sequence is masked off and shifted right so that it will form an integer between 0 and 63. This value is then used as an index for each s-box. The return values from the s-boxes are gathered together and stored in the *temp* value.

The *L, R* and *T* variables are reused for all the iterations. The first iteration outputs to *T* and reads from *R* and *L*, in the next iteration stores it result to *L* and reads from *T* and *R* and so on.

After the second to last iteration the value currently in *R* is compared to the *plainR1* value, of they do not match the key was not the correct one and it is unnecessary to do the last iteration. The hex value in that *R* is combined with is just in order to clear certain bits whose values do not matter. If *R* is the same as *plainR1* the last iteration is performed. After the last iteration the value of *T* is compared to *plainL1*, if they match the function returns a 1 and a possible key has been found otherwise a 0 and it moves on the next key.

4. Generating the modified S-boxes

For the implementation to work there is a need to modify the original S-boxes. The main idea is to incorporate the P permutation and the E permutation of the message schedule into the S-boxes. In the end the S-boxes will work as a lookup table, each giving a bit sequence that needs to be combined with the others to create the message.

The values for the permutations are stored in the e and p arrays, they are essentially the same permutations defined in the DES algorithm. The code presented in Code 6: Preprocessing of the first S-box shows the steps required to generate the first S-box, the same code can be used for the rest of them as well with another shift value.

Code 6: Preprocessing of the first S-box

```
static byte e[48] =
{
    0, 31, 30, 29, 28, 27, 28, 27, 26, 25, 24, 23, 24, 23, 22, 21,
    20, 19, 20, 19, 18, 17, 16, 15, 16, 15, 14, 13, 12, 11, 12, 11,
    10, 9, 8, 7, 8, 7, 6, 5, 4, 3, 4, 3, 2, 1, 0, 31
};
static byte p[32]=
{
    16, 25, 12, 11, 3, 20, 4, 15, 31, 17, 9, 6, 27, 14, 1, 22,
    30, 24, 8, 18, 0, 5,29, 23, 13, 19, 2, 26, 10, 21, 28, 7
};
static uint64_t sb1[4][16] =
{
    {14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7},
    {0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8},
    {4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0},
    {15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13},
};
uint64_t sb1Shifted[4][16];
uint64_t sb1Permutated[4][16];
uint64_t sb1Expanded[4][16];
uint64_t sb1Shuffeled[64];
uint64_t sb11D[64];

void SBoxFix()
{

    int shifts = 7;
    int j = 0;
    // shift the bits to the correct location
    for (int x = 0; x < 4; x++)
    {for (int y = 0; y < 16; y++)
        {
            sb1Shifted[x][y] = sb1[x][y]<<(4*shifts);
        }
    }
    // do the P permutation
    for (int x = 0; x < 4; x++)
    {for (int y = 0; y < 16; y++)
        {for (int i = 0; i <31;i++)
            {
```

```
                copyBit(sb1Shifted[x][y], p[i], sb1Permutated[x][y], 31-i);
            }
        }
    }
    // do the E permuatation
    for (int x = 0; x < 4; x++)
    {for (int y = 0; y < 16; y++)
        {for (int i  = 0; i < 48; i ++)
            {
                copyBit(sb1Permutated[x][y], e[i], sb1Expanded[x][y], 47-i);
            }
        }
    }
    // flatten to a 1d array
    for (int x = 0; x < 4; x++)
    {for (int y = 0; y < 16; y++)
        {
            sb11D[(16*x)+y] = sb1Expanded[x][y];
        }
    }
    // rearrange to allow reads with 6-bit values instead of one 2-bit and one 4-bit
    for (int x = 0; x < 16; x++)
    {
        sb1Shuffeled[j++] = sb11D[x];
        sb1Shuffeled[j++] = sb11D[x+16];
    }
    for (int x = 0; x < 16; x++)
    {
        sb1Shuffeled[j++] = sb11D[32+x];
        sb1Shuffeled[j++] = sb11D[32+x+16];
    }
}
```

The code begins by shifting the values in the S-box to their proper location, the first box representing the leftmost bits. Depending on the S-box the size of the output varies, for the first S-box the output is 32-bits with all bits after bit 28 being set to zero. After this the P permutation is done to the output, now the size of the output is 32-bit values. Now the output needs to be expanded to 48-bits, this is done with the E permutation.

The original S-box was defined as a two dimensional array however to improve read speed and reduce the processing needed to figure out which element to read the S-box should be defined as a one dimensional array. To further improve the read speed a full 6-bit sequence should be used as an index for the array, instead of one 2-bit and one 4-bit sequence. In order for this to work the elements in the S-boxes needs to be rearranged.

5. The GPU implementation

The GPU implementation is an evolution of the CPU implementation. A lot of the same ideas are used again however not all of them are translated to the GPU. The key generation and key schedule had to be remade but the message schedule works in much the same way. Some modification was done to utilize some of the special caches and memory that is available on the GPU.

5.1. The key schedule and key generation

5.1.1. Theory

The same idea with each sub key merged together from eight segments is still used for the GPU; however this requires a lot of cache memory that is not available on the GPU. A naïve implementation with the entire key generation done by the GPU will because of this not yield more than a 50% speedup. The way around this is to generate the sub keys on the host with the CPU and then transfer the keys to the GPU. However this also comes with a limitation, the PCI-express bus can only transfer a certain amount of data each second, 8GB/s for 16x PCI-express 2.0 (7), and this will cap the performance at about 200 million keys/second. So the ideal situation is to generate the key set for the first part of characters on the host with the CPU and then transfer this to the GPU. There the GPU will fill in the last characters. This limit the amount of data that has to be transferred and at the same time it limits the amount of work the GPU has to do for each key. After some testing we determined that 6 characters on the CPU and two on the GPU was the most optimal split for this implementation.

The key schedule for the last two characters that gets run on the GPU will not use the same idea that the CPU uses with combining pre generated segments to the received key. The CPU implementation is memory intensive, requiring a lot of memory reads that are not suited for the GPU. (5) The GPU key schedule needs to be an optimized version of the original implementation. The generation of each of the 16 sub keys can be seen as a permutation of the original input key, the partial table for witch is shown in Table 9: key permutations for the first sub key. This permutation would be a combination of the initial key permutation, the shifts and the final key permutation. For the GPU part the permutation for the last two characters of each sub key are only needed.

Table 9: key permutations for the first sub key

Input bit	1	2	3	4	5	6	7	9	10	…	54	55	59	60	61	62	63
Becomes	8	2	11	44	29	30	40	1	20	…	47	38	15	18	35	39	28

Since each memory transfer to the GPU from the host machine has an overhead (4), it is faster to group together many partial sub keys and

transfer them as a bigger block to the GPU. An easy way to keep track of witch sub keys what threads are working with would be to group together 128 partial sub keys in each block. However to further reduce the number of memory transfers between the GPU and host this implementation uses blocks of 1024 partial sub keys for each transfer.

An important factor to consider with all CUDA programs is the thread block dimensions and the grid dimensions. An easy solution is to use one dimensional thread blocks with 128 threads in each. Each of the treads would then represent one character of the key. Then the grid would need to at least two dimensional, one dimension representing the second to last character of the key, and the other what partial sub key collection that should be used. To use 1024 partial sub keys the grid would need to be three dimensional with the last dimension being used to represent the eight combined sub key groups.

Since the GPU and CPU can work independently of each other the CPU can be generating the next set of partial sub keys while the GPU does the decryption. On newer GPU:s the GPU can do calculations while data is being copied to and from it as well (2). In order to accomplish this, the data transfers and kernel launches need to be started in streams. A stream can be seen in the same way a thread is viewed on the CPU, it is a list of commands the GPU needs to do in a specific sequence. To multithread the CPU part of the program OpenMP (8) was used. With each OpenMP thread taking a block of 268435456 keys and independently of each other generates the partial sub keys for each of its keys and then in their own stream transfers and decrypts with them on the GPU.

5.1.2. Implementation

Because the key is generated using the same scheme as the CPU version the same kind of simple functions for generating the pieces of the sub key can be used. After the pieces are generated they are combined and transferred to the GPU. Code 7: cycling trough the keys shown how the keys are cycled through. Nested *for* loops are used to cycle trough the keys in the same way it was done on the CPU.

Code 7: cycling trough the keys

```
for (int l5 = 0; l5 < 128; l5++)
{// 0xXX_______
    //for (int l4 = 70; l4 >= 0; l4--)
    for (int l4 = 0; l4 < 128; l4++)
    {// 0x__XX_____
        for (int l3 = 0; l3 < 128; l3++)
        {// 0x____XX____
#pragma omp parallel for
            for (int l2 = 0; l2 < 128; l2++)
            {// 0x______XX__
                testKey(l5, l4, l3, l2,size, preKeySpace,
                    keyNotFound, charSet, grid, sBoxPtr, devKs);
            }
```

```
        }
    }
}
```

The first three *for* loops simply loop trough each character, the fourth runs in parallel and executes the test key function where the actual work is done.

Code 8: the grouping of the keys and launching the kernels on the GPU

```
void testKey(int l5, int l4, int l3, int l2, int size, u64 ***preKeySpace,int
*keyNotFound, char *charSet, dim3 grid, sBoxPointers sBoxPtr, int *devKs)
{
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    u64 *keySet;
    cudaMallocHost((void**)&keySet,sizeof(u64)*128*16*8);
    int *foundKey;
    cudaMallocHost((void**)&foundKey, sizeof(int)*4);
    int *devFoundKey;
    cudaMalloc((void**)&devFoundKey, sizeof(int)*4);
    cudaMemcpyAsync(devFoundKey, keyNotFound, sizeof(int)*4,
        cudaMemcpyHostToDevice, stream);
    u64 *devKeySet;
    cudaMalloc((void**) &devKeySet, sizeof(u64)*16*128*8);
    int segmentOfset;
    for (int l1 = 0; l1 < 128;)
    {// 0x_______XX__
        for (int segment = 0; segment < 8; segment++){
            segmentOfset = (16*128*segment);
            for (int bucket = 0; bucket < 128; bucket++)
            {// 0x_________XX
                keySet[(bucket*16)+0+segmentOfset]  =
                    preKeySpace[bucket][2][0] |
                    preKeySpace[l1][3][0] |
                    preKeySpace[l2][4][0] |
                    preKeySpace[l3][5][0] |
                    preKeySpace[l4][6][0] |
                    preKeySpace[l5][7][0];
                ...
                    keySet[(bucket*16)+15+segmentOfset] =
                    preKeySpace[bucket][2][15]|
                    preKeySpace[l1][3][15]|
                    preKeySpace[l2][4][15]|
                    preKeySpace[l3][5][15]|
                    preKeySpace[l4][6][15]|
                    preKeySpace[l5][7][15];
            }
            l1++;
        }
        cudaMemcpyAsync(devKeySet, keySet, sizeof(u64)*128*16*8,
            cudaMemcpyHostToDevice,stream);
        kernel1<<<grid,128,0,stream>>>(sBoxPtr, devFoundKey, devKeySet, devKs);
        cudaMemcpyAsync(foundKey, devFoundKey, sizeof(int)*4,
            cudaMemcpyDeviceToHost,stream);
        cudaStreamSynchronize(stream);
        if(foundKey[0] != -1){
            // handle the found key
        }
    }
    cudaStreamSynchronize(stream);
```

```
    cudaFreeHost(keySet);
    cudaStreamDestroy(stream);
    cudaFree(devFoundKey);
    cudaFree(devKeySet);
    cudaFreeHost(foundKey);
}
```

In Code 8: the grouping of the keys and launching the kernels on the GPU the actual kernel launches and all the copying of memory to the device occurs. Each thread creates a separate stream, this will ensure the GPU will always be busy doing computation even when one thread is occupied with generating keys and another one is transferring them to the GPU. Each thread also allocate space for its own key set in the *keySet* array, the size of this array is set to $16 * 8 * 128$ elements. An array to store the key if found is also created, this array is then filled with -1 values to indicate that a key has not been found.

This function also loops trough two characters, after the *l1* loop there is a another loop that will create the segments of keys to be transferred to the GPU, in this case segments of size 128*8 are used. After the key segment is generated it is transferred to the GPU using asynchronous transfer so that the GPU can keep working while doing the transfer.

The final two characters in each sub key are generated on the GPU, this is shown in Code 9: the key generation on the GPU. This is done in two different places in the GPU code. The first place generates the second to last character form the grid id, the other generates the last form the thread id.

Code 9: the key generation on the GPU

```
int tid = threadIdx.x;
int gid = blockIdx.x;
...
    __shared__ u64 preKey[16];
    ...
    if (tid < 16)
    {
        preKey[tid] =
            devKeySet[(blockIdx.y*16)+tid+(16*128*blockIdx.z)]|
            (((u64)((1<<6)&gid)>>6)<<devKs[tid])|
            (((u64)((1<<5)&gid)>>5)<<devKs[tid+(16)])|
            (((u64)((1<<4)&gid)>>4)<<devKs[tid+(32)])|
            (((u64)((1<<3)&gid)>>3)<<devKs[tid+(48)])|
            (((u64)((1<<2)&gid)>>2)<<devKs[tid+(64)])|
            (((u64)((1<<1)&gid)>>1)<<devKs[tid+(80)])|
            (((u64)((1   )&gid)   )<<devKs[tid+(96)]);
    }
    __syncthreads();
    ...
    u8 c0 = (1<<0)&tid;
    u8 c1 = (1<<1)&tid;
    u8 c2 = (1<<2)&tid;
    u8 c3 = (1<<3)&tid;
    u8 c4 = (1<<4)&tid;
    u8 c5 = (1<<5)&tid;
    u8 c6 = (1<<6)&tid;
```

```
...
key = preKey[15]|
    (((u64)c4>>4)<<46)|
    (((u64)c6>>6)<<43)|
    (((u64)c1>>1)<<20)|
    (((u64)c2>>2)<<14)|
    (((u64)c0>>0)<<1);
...
```

The *tid* and *gid* parameters are used for storing the thread id and block id. Next comes the generation of the second to last character. Because the x dimension of the block is 128 the grid id can be interpreted as the 7-bit representation of the second to last character. The same is true for the thread id. Since the same 7 characters are used by each thread in the thread block these can be shared between all the threads in the thread block trough shared memory. Using shared memory will reduce the number of reads from global memory that needs to be done. Since there are 16 different sub keys only 16 threads need to read from global memory. Each of the 16 threads reads one sub key, after this it generates the bits corresponding to the second to last character of the key. Before the computation can continue all the threads in the current thread block needs to be synced.

How the block id needs to be permutated for each sub key is stored in the *devKs* register arranged for best coalescing. What sub key set is read is determined by the block id, the z id is for the eight key blocks, the y is used for the third to last key. In the cases where certain bits are not used in the sub keys the bit is simply shifted 65 places resulting in no bits being added to the sub key.

The last character is determined by the thread id, each bit is first read into temporary storage containers. This is mainly to limit the amount of bit shifts that needs to be done. Before each key iteration the last character is added to the current sub key, the extra bit shifts are optimized away by the compiler.

5.2. The message schedule

5.2.1. Theory

The theory behind the GPU implementation of the message schedule is very similar to the implementation for the CPU. The only difference is that the last character of the key is generated between each iteration in the key schedule. There is also some local caching of the S-boxes in the shared memory.

Each of the S-boxes are used 15 or in the worst case 16 times for each key, once for each iteration in the key schedule. However the distribution of which element of an S-box is read is seemingly random. In a warp of 32 threads the reads from the S-boxes are rarely from the same element so

the values from the S-boxes are perfect for caching in the shared memory. Reads from the shared memory in each streaming multiprocessor is much faster that reads from the global memory, 1,6TB/s from shared memory compared to 177 GB/s from global memory. (9) The downside is that only the kernels can write to the shared memory, so in the start of each thread block the values have to be read from global memory and written to shared memory.

Each S-box contains 64 elements that need to be transferred to the shared memory. The thread blocks are grouped into warps of 32 threads so it is fairly trivial to have the first two warps do the transfer. The simplest way is to have each thread with and id below 64 read the element at the place corresponding to its own id. And then write the element it to shared memory. This would then have to be repeated for each S-box. This may seem to waste a lot of resources but the speedup of each memory read far outweighs it.

5.2.2. Implementation

Code 10: the S-boxes being cached to shared memory

```
int tid = threadIdx.x;
int gid = blockIdx.x;
...
__shared__ u64 SB1[64];
__shared__ u64 SB2[64];
__shared__ u64 SB3[64];
__shared__ u64 SB4[64];
__shared__ u64 SB5[64];
__shared__ u64 SB6[64];
__shared__ u64 SB7[64];
__shared__ u64 SB8[64];
...
if (tid < 64)
{
    SB1[tid] = devSboxPtr.devSB1[tid];
    SB1[tid] = devSboxPtr.devSB1[tid];
    SB2[tid] = devSboxPtr.devSB2[tid];
    SB3[tid] = devSboxPtr.devSB3[tid];
    SB4[tid] = devSboxPtr.devSB4[tid];
    SB5[tid] = devSboxPtr.devSB5[tid];
    SB6[tid] = devSboxPtr.devSB6[tid];
    SB7[tid] = devSboxPtr.devSB7[tid];
    SB8[tid] = devSboxPtr.devSB8[tid];
}
...
__syncthreads();
```

First the S-boxes are cached in shared memory as shown in Code 10: the S-boxes being cached to shared memory. Each threads *tid* is used as an access index. The *devSboxPtr* is a struct containing pointers to each of the S-boxes. The *if* statement is only taken by the first 64 threads the others rest will have to wait for all the threads to sync up after the read is done. The thread sync done here can be the same one done for the key generation.

Code 11: the GPU message schedule

```
// iteration 15
key = preKey[15]
    |  (((u64)c4>>4)<<46)
    |  (((u64)c6>>6)<<43)
    |  (((u64)c1>>1)<<20)
    |  (((u64)c2>>2)<<14)
    |  (((u64)c0>>0)<<1);
KxorER = key ^ R;
temp = SB1[(0x3F)&(KxorER>>42)];
temp |= SB2[(0x3F)&(KxorER>>36)];
temp |= SB3[(0x3F)&(KxorER>>30)];
temp |= SB4[(0x3F)&(KxorER>>24)];
temp |= SB5[(0x3F)&(KxorER>>18)];
temp |= SB6[(0x3F)&(KxorER>>12)];
temp |= SB7[(0x3F)&(KxorER>>6)];
temp |= SB8[(0x3F)&(KxorER)];
T = L^temp;
// iteration 14
...
// iteration 1
...
if (plainR1 != (R&(0x000079e79e79e79e))){
    return;
}
// iteration 0
key = preKey[0]
    |  (((u64)c3>>3)<<44)
    |  (((u64)c6>>6)<<40)
    |  (((u64)c5>>5)<<30)
    |  (((u64)c4>>4)<<29)
    |  (((u64)c2>>2)<<11)
    |  (((u64)c0>>0)<<8)
    |  (((u64)c1>>1)<<2);
KxorER = key ^ R;
temp = SB1[(0x3F)&(KxorER>>42)];
temp |= SB2[(0x3F)&(KxorER>>36)];
temp |= SB3[(0x3F)&(KxorER>>30)];
temp |= SB4[(0x3F)&(KxorER>>24)];
temp |= SB5[(0x3F)&(KxorER>>18)];
temp |= SB6[(0x3F)&(KxorER>>12)];
temp |= SB7[(0x3F)&(KxorER>>6)];
temp |= SB8[(0x3F)&(KxorER)];
T = L^temp;
if (plainL1 == (T&(0x000079e79e79e79e))){
    foudnKey[0] = threadIdx.x;
    foudnKey[1] = blockIdx.x;
    foudnKey[2] = blockIdx.y;
    foudnKey[3] = blockIdx.z;
    return;
}
return;
```

As mentioned earlier the actual message schedule for the GPU is very similar to the one that was used for the CPU implementation. Code 11: the GPU message schedule shows the modified key schedule for the GPU. The main difference is that instead of the entire key being read from memory for the key is partially generated between the iterations. Another difference

is that when the key has been found only the block id and grid id for the thread that found it is stored in the *foundKey* array. The *foundKey* array is then transferred to the CPU between the kernel launches. This might generate some problems when more than one key is found however in such situation it would be simple to implement an atomic counter of the numbers of keys found and then if more than one are found the key segment can be studied further with the CPU implementation.

6. Results

The CPU implementation was run on a computer containing two Intel Xeon E5420 clocked at 3 GHz and 8 GB of DDR2 memory. The same computer was used for the GPU implementation then equipped with an NVidia GTX570 graphics card. The GPU implementation was also run on an NVidia Tesla C2070 card. The number of keys per second each achieved is shown in Table 10: Million keys/sec.

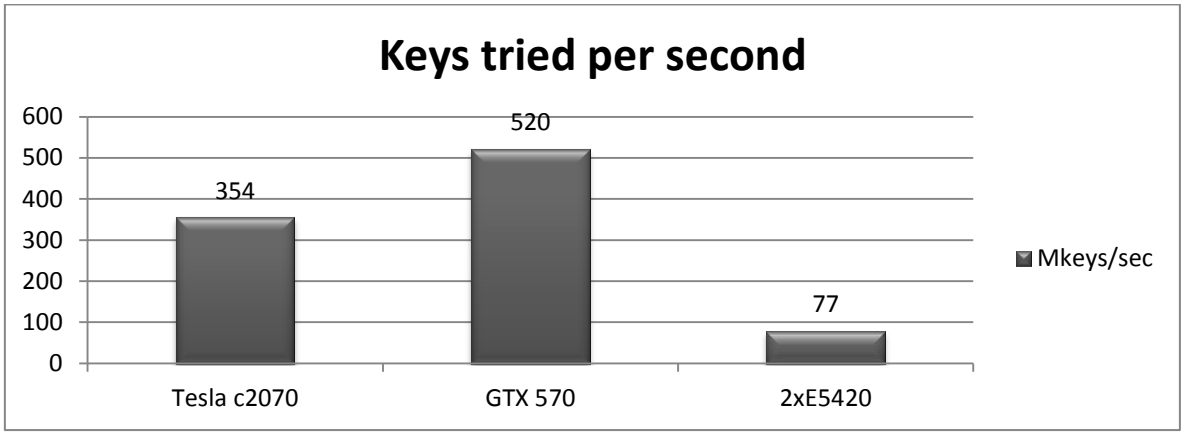


Table 10: Million keys/sec

7. Conclusions

We started the project by building a CPU version of a DES cracker, then it was optimized keeping in mind that it would eventually be ported to the GPU. When the CPU version delivered a decent speed it was ported to the GPU with some smaller modifications. These modifications had to do with the way the key was generated. The key generation was kept partially on the CPU. This was done in order to have the GPU focus on the massively parallel portion of the code, and not having to redo a lot of work.

Even though the GPU version does show a clear improvement form the CPU version, a speedup of 6,75 was measured. The speed is still however no match for purpose built hardware. It is however a cheaper alternative than an all CPU based solution. The benefit of using GPU as a basis instead of building custom hardware is that the GPU based solution can be adapted to any other algorithm without having to rebuild and repurchase hardware.

There are still some optimizations that could possibly be done to the code. The shared memory on the GPU is only 32-bits wide so when storing 64-bit values to it in the way we presented earlier there will be bank conflicts for all reads from shared memory. It is possible to eliminate the bank conflicts in the way described in *CUDA Application Design and Development* (9) so that both parts of the 64-bit value will reside on the same bank.

References

1. **NIST.** *FIPS 46-3.* 1999.

2. **Nvidia.** NVIDIA's Next Generation CUDA Compute Architecture:Fermi. 2009.

3. **Nvidia.** *TUNING CUDA APPLICATIONS FOR FERMI.* May 2011.

4. **Nvidia.** *NVIDIA CUDA C Programming Guide.* 16 4 2012.

5. **Nvidia.** *CUDA C BEST PRACTICES GUIDE.* January 2012.

6. Intel® Cilk™ Plus. [Online] Intel. [Cited: 2012 9 23.]

7. PCI Express Base specification. *PCI_SIG.* [Online] [Cited: 17 8 2012.]

8. OpenMP. *OpenMP Specifications.* [Online] OpenMP Architecture Review Board.

9. **Farber, Rob.** *CUDA Application Design and Development.* s.l. : Elsevier Inc., 2011.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku

- Department of Information Technology
- Department of Mathematics

Åbo Akademi University

- Department of Information Technologies

Turku School of Economics

- Institute of Information Systems Sciences

ISBN 978-952-12-2787-5
ISSN 1239-1891