# Bitcoin Miner Implementation Using Nvidia CUDA

Eleftherios Amperiadis

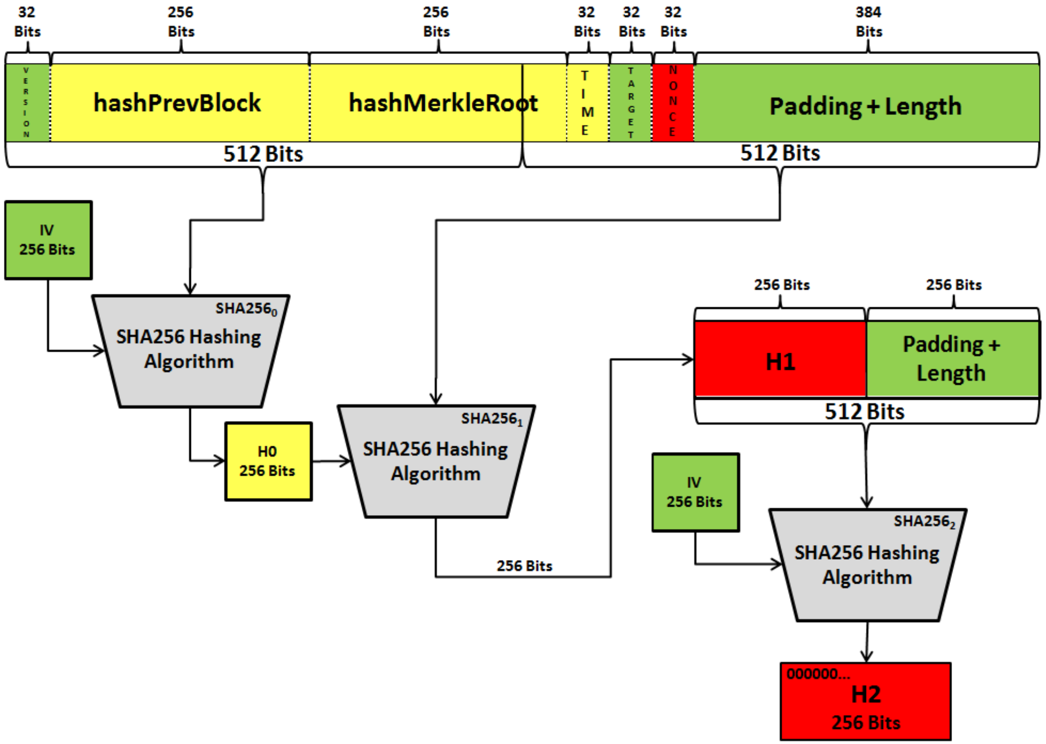
City College of New York

Senior Design II

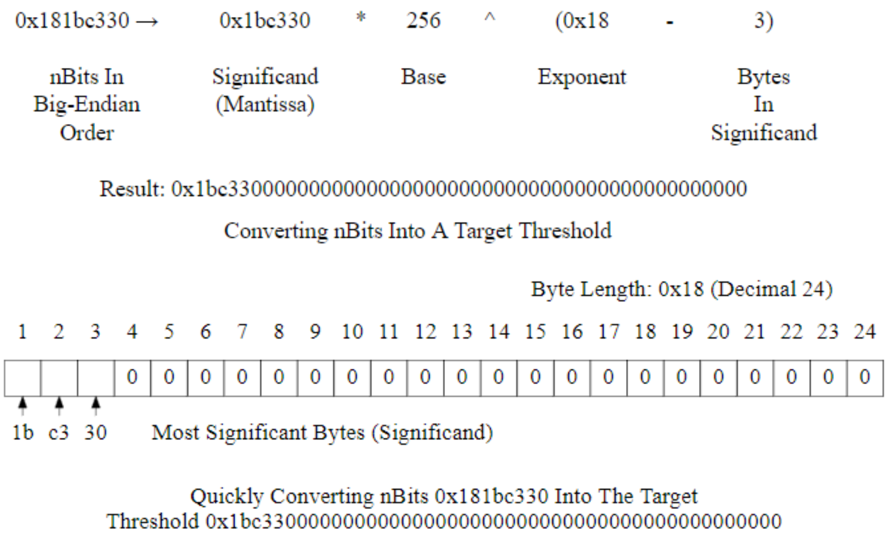
Spring 2019

# What is Bitcoin Mining?

- It is the process of applying three rounds of the SHA256 hashing algorithm to a 640-bit long block header to get a 256-bit output H2.
- If H2 is less than or equal to the difficulty vector, the mining is a success.



# Difficulty Vector




```
__device__
void computeTarget(uint32_t nbits)
{
    uint32_t mantissa = nbits & 0x00FFFFFF;
    uint32_t exponent = (nbits & 0xFF000000) >> 24;
    uint32_t offset = 0x20 - (exponent - 0x03);
    uint8_t inter[32]={0};

    inter[offset-1]   = (mantissa & 0x000000FF);

    inter[offset-2] = (mantissa & 0x0000FF00) >> 8;

    inter[offset-3] = (mantissa & 0x00FF0000) >> 16;

    target[0] = inter[0]<<24  | inter[1]<<16  | inter[2]<<8  | inter[3];
    target[1] = inter[4]<<24  | inter[5]<<16  | inter[6]<<8  | inter[7];
    target[2] = inter[8]<<24  | inter[9]<<16  | inter[10]<<8 | inter[11];
    target[3] = inter[12]<<24 | inter[13]<<16 | inter[14]<<8 | inter[15];
    target[4] = inter[16]<<24 | inter[17]<<16 | inter[18]<<8 | inter[19];
    target[5] = inter[20]<<24 | inter[21]<<16 | inter[22]<<8 | inter[23];
    target[6] = inter[24]<<24 | inter[25]<<16 | inter[26]<<8 | inter[27];
    target[7] = inter[28]<<24 | inter[29]<<16 | inter[30]<<8 | inter[31];
}
```

- The 32-bit long target in the block header is expanded into a 256-bit difficulty vector.
- On the left is the specification of this expansion as shown in the Bitcoin developers manual.
- On the right is my implementation of this operation. It is similar to the way floating point numbers are encoded.

# The Bitcoin Block Header in Detail

| Field | Size | Description |
|---|---|---|
| Version | 32 bits | Block version information that is based on the Bitcoin software version creating this block |
| hashPrevBlock | 256 bits | The hash of the previous block accepted by the Bitcoin network |
| hashMerkleRoot | 256 bits | Bitcoin transactions are hashed indirectly through the Merkle Root |
| Timestamp | 32 bits | The current timestamp in seconds since 1970-01-01 T00:00 UTC |
| Target | 32 bits | The current Target represented in a 32 bit compact format |
| Nonce | 32 bits | Goes from 0x00000000 to 0xFFFFFFFF and is incremented after a hash has been tried |
| Padding + Length | 384 bits | Standard SHA256 padding that is appended to the data above |

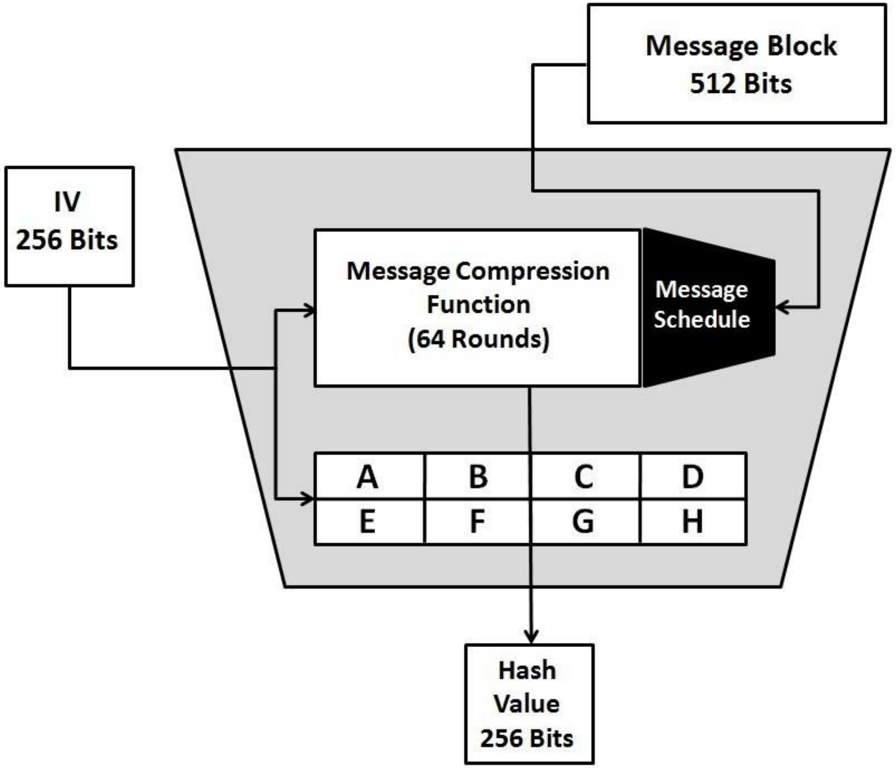
- The fields in Green rarely change.
- The fields in yellow change, but not frequently.
- The fields in red change after every attempted hash.
- This analysis is important for optimizing the miner in later sections. Some rounds of SHA256 do not have to be repeated, which saves computations.

# Why is Bitcoin mining needed?

- The process of mining is to solve a proof-of-work problem.
- The collective of miners attempting to solve this problem are all in a competition to solve the problem first.
- Throughout this process, miners are confirming, recording and validating transactions of Bitcoin which is a valuable service to the Bitcoin network.
- The miner who solves the proof-of-work problem is rewarded with Bitcoin as an incentive.

# The SHA256 Hashing Algorithm

- SHA256 is part of a family of cryptographic hash functions developed by the National Security Agency.
- The SHA256 algorithm can accept strings up to $2^{64}$-1 bits long and will compress these strings down to 256-bit output.



# SHA256 Constants

```
Initialize hash values:
(first 32 bits of the fractional parts of the square roots of the first 8 primes 2..19):
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19
```

```
Initialize array of round constants:
(first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311):
k[0..63] :=
   0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
   0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
   0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
   0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
   0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
   0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
   0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
   0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
```

# SHA256 Padding and Schedule Array Generation

```
Pre-processing (Padding):
begin with the original message of length L bits
append a single '1' bit
append K '0' bits, where K is the minimum number >= 0 such that L + 1 + K + 64 is a multiple of 512
append L as a 64-bit big-endian integer, making the total post-processed length a multiple of 512 bits
```

```
Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    create a 64-entry message schedule array w[0..63] of 32-bit words
    (The initial values in w[0..63] don't matter, so many implementations zero them here)
    copy chunk into first 16 words w[0..15] of the message schedule array

    Extend the first 16 words into the remaining 48 words w[16..63] of the message schedule array:
    for i from 16 to 63
        s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
        s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightshift 10)
        w[i] := w[i-16] + s0 + w[i-7] + s1
```

# SHA256 Compression Function

```
Initialize working variables to current hash value:
a := h0
b := h1
c := h2
d := h3
e := h4
f := h5
g := h6
h := h7

Compression function main loop:
for i from 0 to 63
    S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
    ch := (e and f) xor ((not e) and g)
    temp1 := h + S1 + ch + k[i] + w[i]
    S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
    maj := (a and b) xor (a and c) xor (b and c)
    temp2 := S0 + maj

    h := g
    g := f
    f := e
    e := d + temp1
    d := c
    c := b
    b := a
    a := temp1 + temp2
```

```
Add the compressed chunk to the current hash value:
h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d
h4 := h4 + e
h5 := h5 + f
h6 := h6 + g
h7 := h7 + h

Produce the final hash value (big-endian):
digest := hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
```

# Development Process

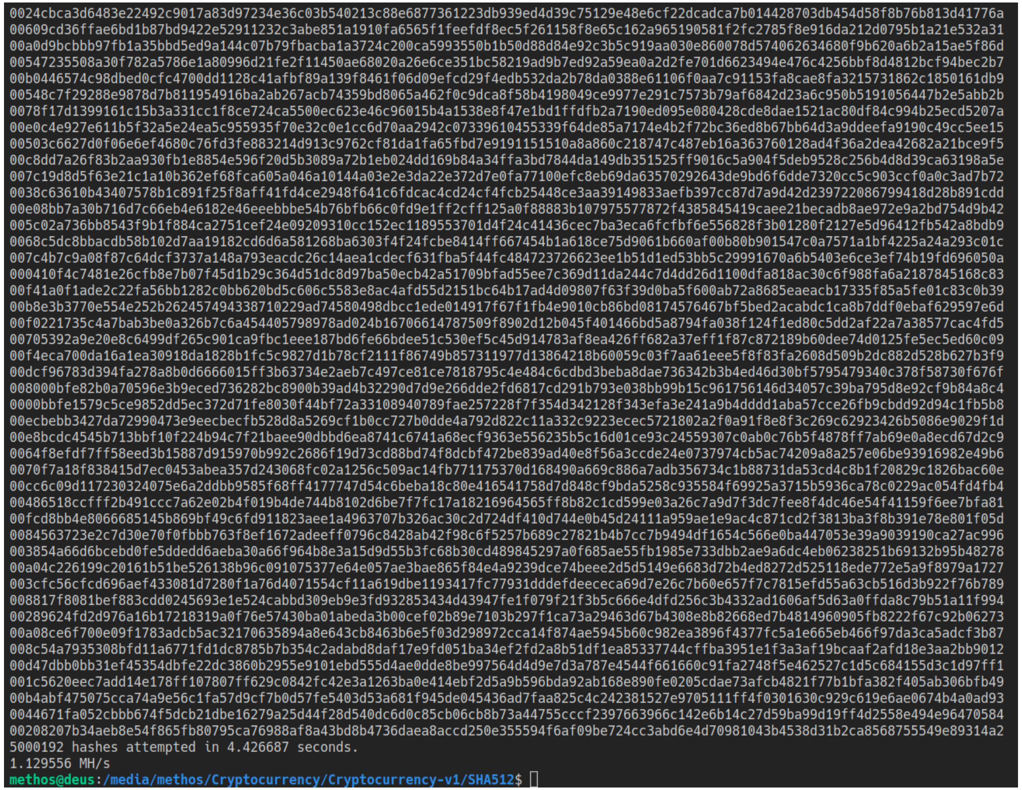
- First, an SHA256 Nvidia CUDA kernel was created.
- Over the course of several months, this kernel was continuously optimized.
- Seven major versions of the kernel exist, each better than the last.
- Version 8 has the complete Bitcoin miner using version 7’s SHA256 kernel as an engine.

# Why use Nvidia CUDA?

- Because of the computationally intensive nature of generating a hash of a string using SHA256, a parallel computation devices such as a Graphics Processing Unit (GPU) is advantageous.
- A single SHA256 hash can not be computed in parallel, however multiple computational cores such as those present in a GPU can solve multiple hashes per clock cycle iteration.

# SHA256 Implementation Version 1

- The first version of the SHA256 CUDA kernel achieves about 1 MH/s.
- In this context, MH/s refers to 1,000,000 hashes per second.



# SHA256 Implementation Version 2

```
cudaMallocManaged(&message_array, (cuda_blocks*array_len*string_len*sizeof(unsigned char)));
cudaMallocManaged(&hashed_array, (array_len*cuda_blocks*64*sizeof(unsigned char)));
cudaMallocManaged(&padded_array, (array_len*cuda_blocks*128*sizeof(unsigned char)));
cudaMallocManaged(&hashed_winner, (array_len*cuda_blocks*sizeof(bool)));

temp_array = generateArray(array_len*cuda_blocks, string_len);
for(int i=0; i<array_len*string_len;i++)
{
  message_array[i] = temp_array[i];
}
```

```
__global__
void generateArray(unsigned char* message_array, int string_len)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int upper_bound = idx * string_len;
  curandState state;
  curand_init(clock64(), idx, 0, &state);

  for (int i = 0; i < string_len; i++)
  {
    uint32_t index = curand_uniform(&state);
    message_array[upper_bound+i] = index;
  }
}
```

- The function that generates random strings that feeds the SHA256 Kernel was moved from the host(CPU-top) to the device(GPU-bottom).
- Curand CUDA library was used to generate random uint32_t values.
- Performance improved up to 8.5 MH/s.

```
methos@deus:/media/methos/Cryptocurrency/Cryptocurrency-v2/SHA512$ ./cuda512
50069504 hashes attempted in 5.834028 seconds.
8.582322 MH/s
```

# SHA256 Implementation Version 3

```
__global__
void padding(unsigned char *message, int size, unsigned char *hashed_array,
  unsigned char *padded_array, bool* hashed_winner)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;

  int thread_offset = idx*128;
  int message_offset = idx*size;
  int pad_offset = 112;



  for(int i=0;i<size;i++)
    padded_array[i+thread_offset]=message[i+message_offset];

  padded_array[size+thread_offset] = 0x80;

  for(int i=size+1+thread_offset;i<pad_offset+thread_offset;i++)
    padded_array[i] = 0x00;

  uint64_t val = size*8;

  for(int i=pad_offset+thread_offset;i<pad_offset+thread_offset+8;i++)
    padded_array[i] = 0x00;

  padded_array[pad_offset+thread_offset+8]  =  val >> 56;
  padded_array[pad_offset+thread_offset+9]  =  val >> 48;
  padded_array[pad_offset+thread_offset+10] =  val >> 40;
  padded_array[pad_offset+thread_offset+11] =  val >> 32;
  padded_array[pad_offset+thread_offset+12] =  val >> 24;
    padded_array[pad_offset+thread_offset+13] =  val >> 16;
    padded_array[pad_offset+thread_offset+14] =  val >>  8;
    padded_array[pad_offset+thread_offset+15] =  val >>  0;

  computeHash((unsigned char*)padded_array, 128, (unsigned char*)hashed_array, idx, hashed_winner);
}
```

- Padding was optimized to be more streamline from v.2 (left) to v.3 (bottom).
- The #pragma unroll directive was used to unroll loops.
- Performance increased to 24 MH/s.

```
methos@deus:/media/methos/Cryptocurrency/Cryptocurrency-v3/algorithms$ ./cuda256 10 100000000
100007936 solutions in 4.111877 seconds at 10 difficulty.
24.321724 MH/s
```

```
__device__
void sha256Pad(uint8_t *input, int inlen, uint32_t *output, uint8_t *pad, int idx, int input_stride,
                uint8_t *state, int dif, int pad_stride)
{
    #pragma unroll 30
    for (int i=0;i< 30;i++)
        pad[i+pad_stride]  =  input[i+input_stride];

    pad[30+pad_stride] =  0x80;

    #pragma unroll 32
    for (int i=31;i<63;i++)
        pad[i+pad_stride] =  0x00;

    pad[63+pad_stride] =  0xf0;

}
```

# SHA256 Implementation Version 4

```
__device__
void sha256Compute(uint32_t *input, uint32_t *out, int idx, int input_stride, int out_stride)
{
    uint32_t pad[16];

    #pragma unroll 8
    for (int i=0;i< 8;i++)
        pad[i]  =  input[i+input_stride];

    pad[8] =  0x80000000;

    #pragma unroll 6
    for (int i=9;i<15;i++)
        pad[i] =  0x00000000;

    pad[15] =  0x00000100;

    uint32_t s[64];
    uint32_t values[8];

    #pragma unroll 16
    for(int i=0;i<16;i++)
        s[i]  = pad[i];

    #pragma unroll 48
    for(int i=16;i<64;i++)
        s[i] = s[i-16]+(s0(s[i-15]))+s[i-7]+(s1(s[i-2]));

    A = h[0];
    B = h[1];
    C = h[2];
    D = h[3];
    E = h[4];
    F = h[5];
    G = h[6];
    H = h[7];

    uint32_t temp1;
    uint32_t temp2;

    #pragma unroll 64
    for (int i=0;i<64;i++)
    {
        temp1 = H+(S1(E))+(ch(E,F,G))+k[i]+s[i];
        temp2 = (S0(A))+(maj(A,B,C));

        H = G;
        G = F;
        F = E;
        E = D+temp1;
        D = C;
        C = B;
        B = A;
        A = temp1+temp2;
    }

    out[0+out_stride] = h[0]+A;
    out[1+out_stride] = h[1]+B;
    out[2+out_stride] = h[2]+C;
    out[3+out_stride] = h[3]+D;
    out[4+out_stride] = h[4]+E;
    out[5+out_stride] = h[5]+F;
    out[6+out_stride] = h[6]+G;
    out[7+out_stride] = h[7]+H;

}
```

- v.4 combines the padding and compression functions, thus reducing per thread function calls.
- It also uses the same word sizes, which allows for fewer read-writes from memory by transferring 32-bits at a time instead of 8-bits. Total looping goes from 64 down to 16 for padding.
- Performance went up to 28 MH/s.

```
methos@deus:/media/methos/Cryptocurrency/Cryptocurrency-v4/algorithms$ ./cuda256 10 100000000
100007936 solutions in 3.575889 seconds at 10 difficulty.
27.967293 MH/s
```

# SHA256 Implementation Version 6

```
uint32_t *input;
uint32_t *output;
uint8_t *byte;

cudaMallocManaged(&input, 256);
cudaMallocManaged(&output, 256);
cudaMallocManaged(&byte, 1);
```

```
uint32_t *d_input;
uint32_t *d_output;
uint8_t *d_state;

cudaMallocManaged(&d_input, threads*256);
cudaMallocManaged(&d_output, threads*32);
cudaMallocManaged(&d_state, threads);
```

- Minimized memory transfer from host to device by statically allocating the memory on the device.
- We are only interested in a single input and output, not all inputs and outputs. This allows us to only transfer 512 bits total, down from 512 bits times thread count.
- Top left is v.6. Bottom left is all prior versions.

# Version 6 Continued

```
__device__
void sha256Check(uint32_t *out, int dif, int out_stride, uint8_t *state, int idx)
{
    if ((out[out_stride] >> (32-dif)) == 0)
    {
        state[idx] = 1;
    }
    else
        state[idx] = 0;
}
```

```
__device__
void verifyLeadingZeroes(unsigned char *hash, int leading_zero, int hash_offset, bool* hashed_winner, int idx)
{
  for(int i=0;i<64;i++)
  {
    for (int j=0;j<8;j++)
    {
      if(leading_zero == 0)
      {
        hashed_winner[idx] = true;
        i = 64;
        break;
      }
      if(((hash[i+hash_offset] >> j) & 0x01) != 0)
      {
        i = 64;
        break;
      }
      else
        leading_zero--;
    }
  }
}
```

- Function for checking which H2 is valid was simplified. Instead of iterating through each bit using loops, bit-wise shift operations were used to quickly compare values.
- Top left is v.6. Bottom left is all prior versions.

# Version 6 Continued

```
#pragma unroll 64
for (int i=0;i<64;i++)
{
    temp1 = H+(S1(E))+(ch(E,F,G))+k[i]+s[i];
    temp2 = (S0(A))+(maj(A,B,C));

    H = G;
    G = F;
    F = E;
    E = D+temp1;
    D = C;
    C = B;
    B = A;
    A = temp1+temp2;
}
```

```
#define P(a, b, c, d, e, f, g, h, x, K)       \
  {                                            \
    temp1 = h + S3(e) + F1(e, f, g) + K + x;  \
    temp2 = S2(a) + F0(a, b, c);              \
    d += temp1;                               \
    h = temp1 + temp2;                        \
  }
```

- Memory write and read operations heavily reduced by shifting the values in each subsequent call to P in the compression function.
- Performance went up to 49 MH/s.

```
P(A, B, C, D, E, F, G, H, w[0], K[0]);
P(H, A, B, C, D, E, F, G, w[1], K[1]);
P(G, H, A, B, C, D, E, F, w[2], K[2]);
P(F, G, H, A, B, C, D, E, w[3], K[3]);
P(E, F, G, H, A, B, C, D, w[4], K[4]);
P(D, E, F, G, H, A, B, C, w[5], K[5]);
P(C, D, E, F, G, H, A, B, w[6], K[6]);
P(B, C, D, E, F, G, H, A, w[7], K[7]);
```

```
methos@deus:/media/methos/Cryptocurrency/Cryptocurrency-v6/algorithms$ ./cuda256 30 512 32
7d390b9490da0c4fa8db561cb9b70bcd1ef21477cd7de8622b34fc8dffffffff
0000000295579ba2c262acec56a373bcd4ae0b48258f607157562a74e67fffd5
486457344 hashes attempted in 9.887338 seconds at 30 difficulty.
49.200032 MH/s
```

# SHA256 Implementation Version 7

```
while(1)
{
    uint32_t *input;
    uint32_t *output;
    uint8_t *byte;

    cudaMallocManaged(&input, 256);
    cudaMallocManaged(&output, 256);
    cudaMallocManaged(&byte, 1);

    sha256Compute<<<blocks, threads_per_block>>>(input,output,nonce,dif,byte);
    cudaDeviceSynchronize();


    counter += threads;

    if(byte[0]==1)
    {
        for(int j = 0; j < 8; j++)
            printf("%08x", input[j]);
        printf("\n");

        for(int j = 0; j < 8; j++)
            printf("%08x", output[j]);
        printf("\n");

        break;
    }

    cudaFree(input);
    cudaFree(output);
    cudaFree(byte);
}
```

```
cudaMallocManaged(&input, 128);
cudaMallocManaged(&counter,64);

start = clock();

sha256Compute<<<blocks, threads_per_block>>>(input,nonce,dif,counter,threads);
cudaDeviceSynchronize();

for(int j = 0; j < 4; j++)
    printf("%08x", input[j]);
printf("\n");
```

- All looping is moved from host to device. Every time cudaMallocManaged or a kernel invocation is called, a large amount of overhead exist because memory transfers from host to device.
- Main function version 7 is on the bottom left, main function of older functions is on the top left.
- Performance increased to 1700 MH/s.

```
methos@deus:/media/methos/Cryptocurrency/Cryptocurrency-v7/algorithms$ ./cuda256 32 512 512
d1651f4c8134d05da2f9d24700040eac
2878685472 hashes attempted in 1.694152 seconds at 32 difficulty.
1699.189608 MH/s
```

# Bitcoin Miner

```
if (computeH0 == false)
{
    w[0]  = block_header[0];
    w[1]  = block_header[1];
    w[2]  = block_header[2];
    w[3]  = block_header[3];
    w[4]  = block_header[4];
    w[5]  = block_header[5];
    w[6]  = block_header[6];
    w[7]  = block_header[7];
    w[8]  = block_header[8];
    w[9]  = block_header[9];
    w[10] = block_header[10];
    w[11] = block_header[11];
    w[12] = block_header[12];
    w[13] = block_header[13];
    w[14] = block_header[14];
    w[15] = block_header[15];
    w[16] = w[0]+(s0(w[1]))+w[9]+(s1(w[14]));
    w[17] = w[1]+(s0(w[2]))+w[10]+(s1(w[15]));
    w[18] = w[2]+(s0(w[3]))+w[11]+(s1(w[16]));
    w[19] = w[3]+(s0(w[4]))+w[12]+(s1(w[17]));
    w[20] = w[4]+(s0(w[5]))+w[13]+(s1(w[18]));
    w[21] = w[5]+(s0(w[6]))+w[14]+(s1(w[19]));
    w[22] = w[6]+(s0(w[7]))+w[15]+(s1(w[20]));
    w[23] = w[7]+(s0(w[8]))+w[16]+(s1(w[21]));
    w[24] = w[8]+(s0(w[9]))+w[17]+(s1(w[22]));
    w[25] = w[9]+(s0(w[10]))+w[18]+(s1(w[23]));
    w[26] = w[10]+(s0(w[11]))+w[19]+(s1(w[24]));
    w[27] = w[11]+(s0(w[12]))+w[20]+(s1(w[25]));
    w[28] = w[12]+(s0(w[13]))+w[21]+(s1(w[26]));
    w[29] = w[13]+(s0(w[14]))+w[22]+(s1(w[27]));
    w[30] = w[14]+(s0(w[15]))+w[23]+(s1(w[28]));
    w[31] = w[15]+(s0(w[16]))+w[24]+(s1(w[29]));
    w[32] = w[16]+(s0(w[17]))+w[25]+(s1(w[30]));
    w[33] = w[17]+(s0(w[18]))+w[26]+(s1(w[31]));
```

- H0 from slide 2 only needs to be calculated once per block, the code on the left handles this optimization by using a flag.
- Final performance is 1630 MH/s.

```
methos@deus:/media/methos/Cryptocurrency/Cryptocurrency-v8/algorithms$ ./cuda256BTC 512 512 test.txt
1 cuda device(s)
GeForce RTX 2070 compute capability 7.5
Using GeForce RTX 2070

Block Header: 02000000000000000000000b6ff0b1b1680a2862a30ca44d346d9e8910d334beb48ca0c9d10aa52ee949386ca9385695f04ede270dda20810decd12bc9b048aaab31471381dec2f1dc31b3075240e73
First Round:  aa6610cbc03e38bd816f8932e89d6d0b262956d56593cc208000a62c2c98cc84
Solution:     0000007512dd1fa74bd086a297e633829416e17eb8172f8ded55e78376454909
Target:       000000c31b30000000000000000000000000000000000000000000000000000000
4939226368 hashes attempted in 3.029929 seconds.
1630.145910 MH/s
```