# Embedded Computer Architecture

## Assignment 2

- Mining application on GPU

Submitted by:

Praveen Pujari: 0924737(p.pujari@student.tue.nl)

Sethuraman Jayaraman : 0926183 (s.jayaraman@student.tue.nl)

Date: 18-12-2014

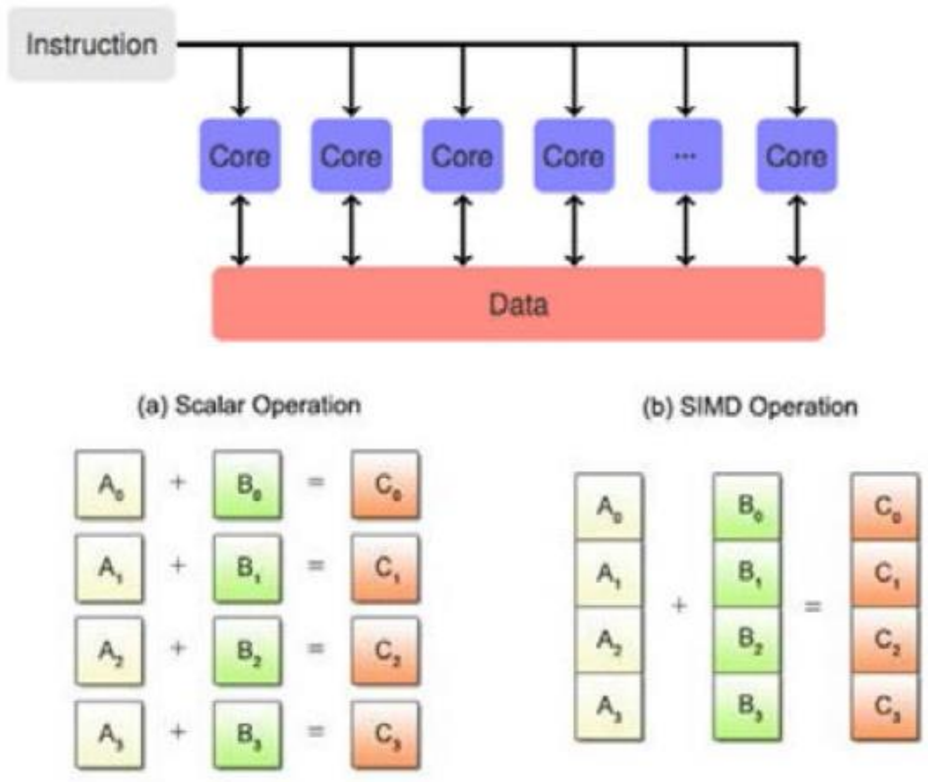
## Introduction

This assignment is based on GPGPU(General Purpose GPU). We explore the parallelism and computational capabilities of the GPU and port Coinporaal Mining application which involves collecting Coinporaal coins. They are obtained as result of solving hashes. In this assignment, we will try to optimize the hashing algorithm on GPUs. This enables us to join the mining competition as an extracurricular to mine as many coins as possible.

## CUDA – an Introduction

GPU is a hardware specially designed for highly parallel applications. GPUs are massively multithreaded many-core chips with hundreds of cores, thousands of concurrent threads, huge economies of scale and have an aggressive performance growth. CUDA is a scalable parallel programming model where a program runs on any number of processors without recompiling. The GPU core is the stream processor and Stream processors are grouped in Stream Multiprocessors. SM is basically a SIMD processor (Single Instruction Multiple Data). CUDA Architecture Expose GPU computing for general purpose. In CUDA, a kernel is executed by many threads, a thread is a sequence of executions and many threads will be running at the same time. CUDA C/C++ is based on industry-standard C/C++ with a small set of extensions to enable heterogeneous programming and many straightforward APIs to manage devices, memory etc.

We have used NSIGHT/Cuda 6.5 on Visual Studio 2013 development environment for analysing the performance.



## Mining Application

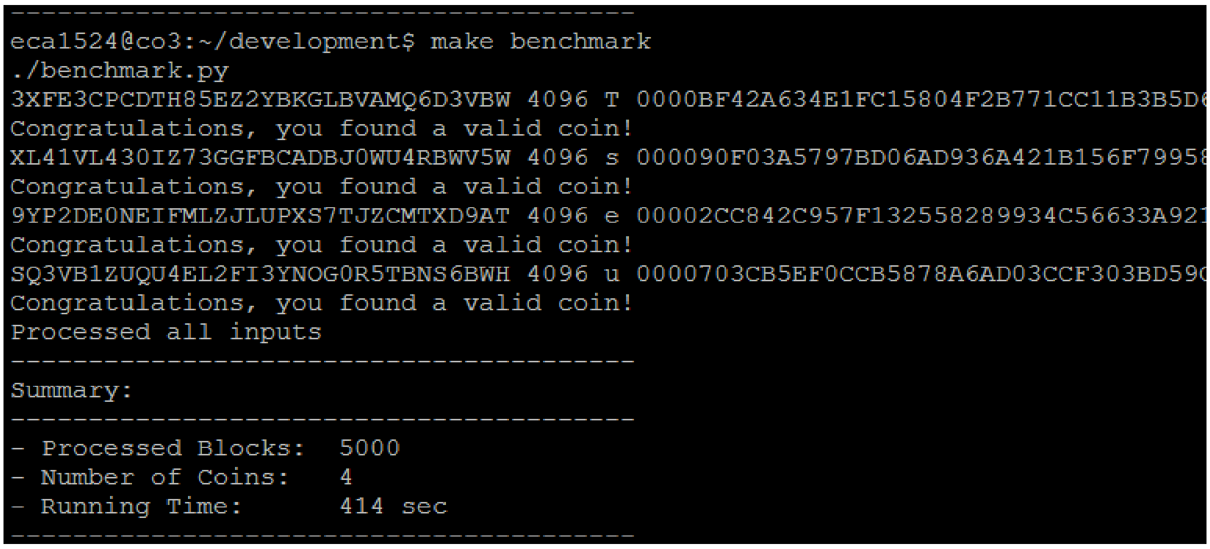
### Source

The application generates hash for given input string, nonce and multiplier.

The competition server gives 32 character input string and it is processed with 62 single character nonces ‘a’…’z’,’A’…’Z’,’0’…’9’ with a multiplier of 4096. Then the result is verified by competition server for Valid Coins i.e, four leading zeroes.

### Results:

The output of the benchmark for the source code is shown below:



```
----------------------------------------
eca1524@co3:~/development$ make benchmark
./benchmark.py
3XFE3CPCDTH85EZ2YBKGLBVAMQ6D3VBW 4096 T 0000BF42A634E1FC15804F2B771CC11B3B5D
Congratulations, you found a valid coin!
XL41VL430IZ73GGFBCADBJ0WU4RBWV5W 4096 s 000090F03A5797BD06AD936A421B156F7995
Congratulations, you found a valid coin!
9YP2DE0NEIFMLZJLUPXS7TJZCMTXD9AT 4096 e 00002CC842C957F132558289934C56633A92
Congratulations, you found a valid coin!
SQ3VB1ZUQU4EL2FI3YNOG0R5TBNS6BWH 4096 u 0000703CB5EF0CCB5878A6AD03CCF303BD59
Congratulations, you found a valid coin!
Processed all inputs
----------------------------------------
Summary:
----------------------------------------
- Processed Blocks:   5000
- Number of Coins:    4
- Running Time:       414 sec
----------------------------------------
```

The crux of the computation lies in multiplier times iterations done in *requestInput(base)* and the **Hash** function. The hash function is ported on to GPU for parallel processing to improve the performance.

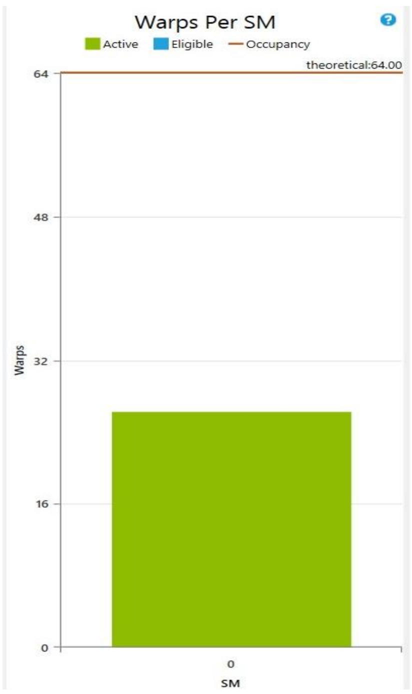
## Porting of Code to GPU

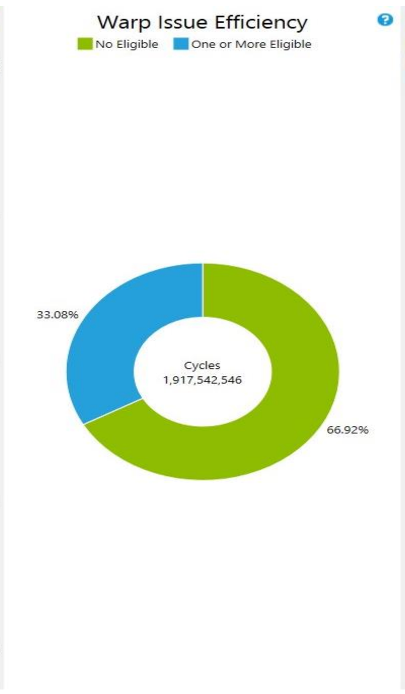
The hash function is ported on to the device using single block and single thread. The performance was obviously slower than the CPU. The first intuition was to use 62 blocks, one for each nonce processed in parallel having 256 threads each. Since 256 parallel operations were observed in the code.

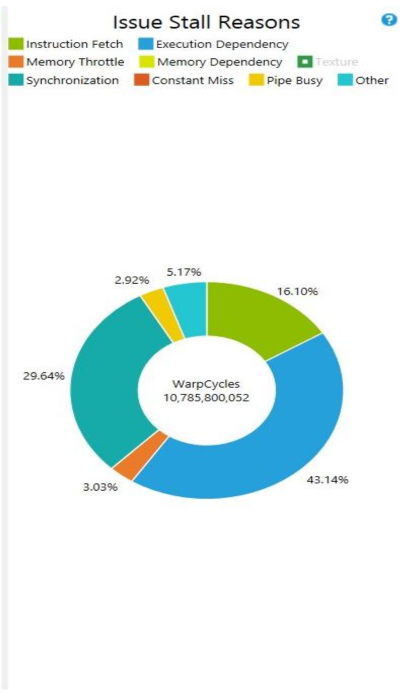
### Implementation I – (62 blocks 256 threads):

The performance analyzed using NSIGHT Visual Profiler for this implementation is shown below. The performance bottlenecks observed are:

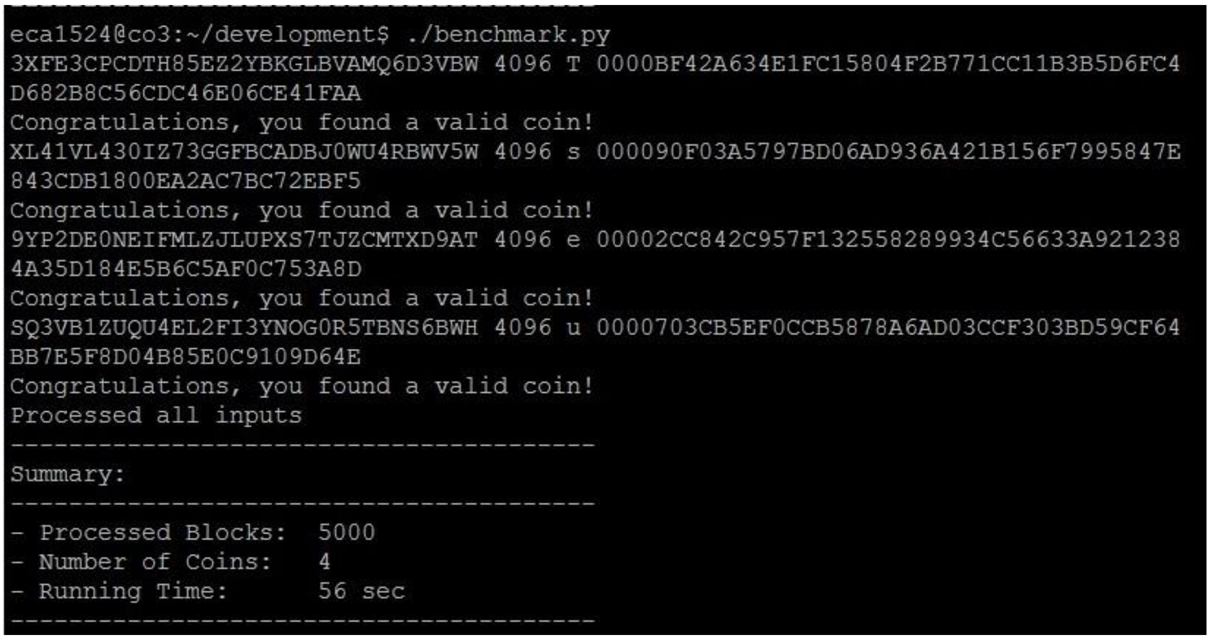
- Less Utilization of the threads as the no. of operations requiring all 256 were less.
- Since lesser no. of threads are needed for most operations, the use of *if-statements* increased, which gave way to problems due to branching.
- Synchronization of large no. of threads consumed more GPU cycles.







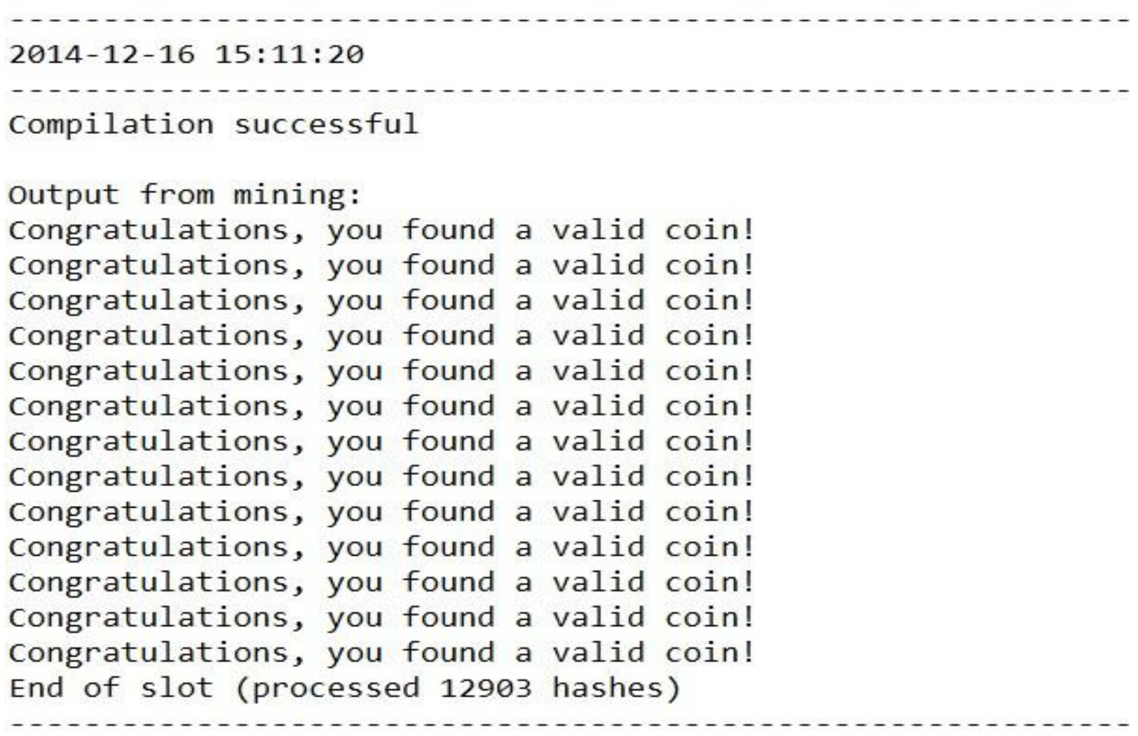
The output of benchmark of 256- threads implementation:



```
eca1524@co3:~/development$ ./benchmark.py
3XFE3CPCDTH85EZ2YBKGLBVAMQ6D3VBW 4096 T 0000BF42A634E1FC15804F2B771CC11B3B5D6FC4
D682B8C56CDC46E06CE41FAA
Congratulations, you found a valid coin!
XL41VL430IZ73GGFBCADBJ0WU4RBWV5W 4096 s 000090F03A5797BD06AD936A421B156F7995847E
843CDB1800EA2AC7BC72EBF5
Congratulations, you found a valid coin!
9YP2DE0NEIFMLZJLUPXS7TJZCMTXD9AT 4096 e 00002CC842C957F132558289934C56633A921238
4A35D184E5B6C5AF0C753A8D
Congratulations, you found a valid coin!
SQ3VB1ZUQU4EL2FI3YNOG0R5TBNS6BWH 4096 u 0000703CB5EF0CCB5878A6AD03CCF303BD59CF64
BB7E5F8D04B85E0C9109D64E
Congratulations, you found a valid coin!
Processed all inputs
----------------------------------------
Summary:
----------------------------------------
- Processed Blocks:  5000
- Number of Coins:   4
- Running Time:      56 sec
----------------------------------------
```

The running time is reduced to 56 seconds by use of 256 threads. Which gives an improvement by a factor of 8.

The output from the competition server is shown below:



```
------------------------------------------------------------
2014-12-16 15:11:20
------------------------------------------------------------
Compilation successful

Output from mining:
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
End of slot (processed 12903 hashes)
------------------------------------------------------------
```

The implementation is now able to process 12903 hashes in the given slot.

## Implementation II – (62 blocks 32 threads):

This implementation is the result of our observations made with the performance analyzer.

Each block in a grid of GPU is made to process the base string using a distinct nonce.

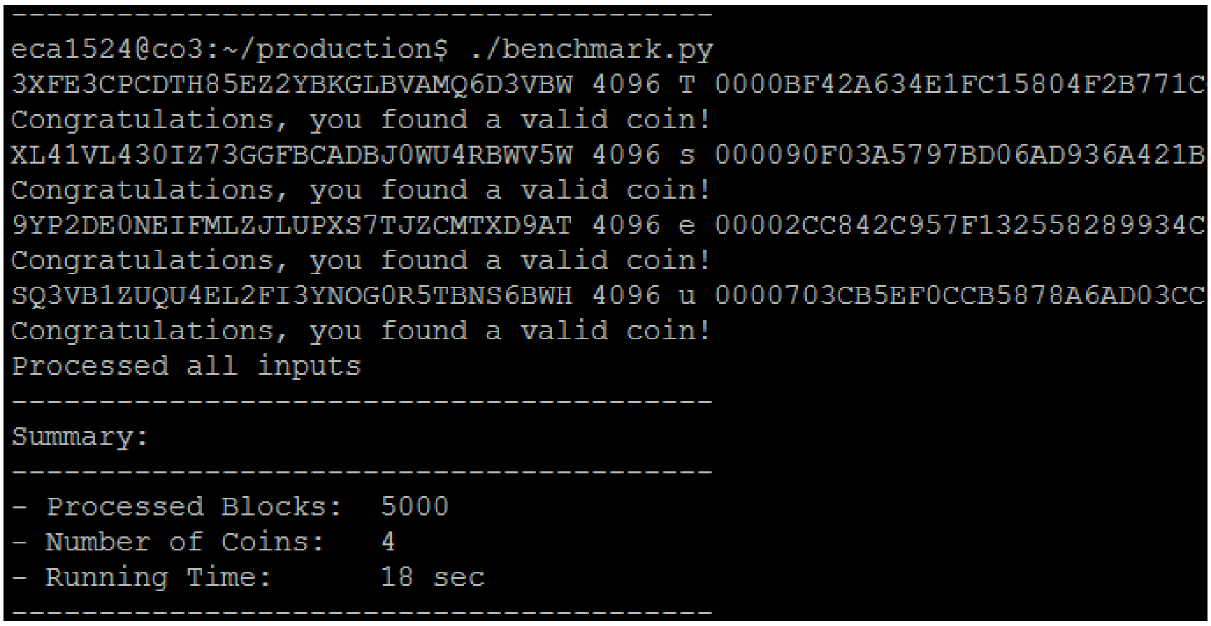
| **Native code** | **32 threads 62 blocks ported code** |
|---|---|
| Main.c | Main.cu |
| The device pointers for the input and output of Hash kernel are declared, allocated memory on device and copied to the global memory before making the kernel call. | |
| `//calculate hash`<br>`      Hash(input, output_hash);` | `char* d_input;`<br>`cudaMalloc((void**)&d_input, sizeof(char)*64);`<br>`char* d_output_hash;`<br>`cudaMalloc((void**)&d_output_hash, 62*33 * sizeof(char));`<br>`cudaMemcpy(d_input, input, sizeof(char)*64, cudaMemcpyHostToDevice);`<br><br>`      Hash<<<62,32>>>(d_input, d_output_hash,(INPUT_SIZE+1));`<br><br>`cudaMemcpy(output_hash, d_output_hash, 62*sizeof(char)*(33), cudaMemcpyDeviceToHost);`<br>`      cudaFree(d_input);`<br>`      cudaFree(d_output_hash);` |
| Interface.c | Interface.cu |
| Instead of passing the input with (**INPUT_MULT x UNIQUE_INPUT_SIZE**) size, it is enough to make sure that we send all the needed characters that is required for hashing. | |

| | |
|---|---|
| for(int j=0;j<INPUT_MULT;j++)<br>for(int<br>i=0;i<UNIQUE_INPUT_SIZE;i++)<br>hash[j*UNIQUE_INPUT_SIZE+i]=hash[i]<br>;<br>hash[INPUT_SIZE]='\0'; | for(int i=0;i<UNIQUE_INPUT_SIZE;i++)<br>hash[UNIQUE_INPUT_SIZE+i]=hash[i];<br>    hash[65]='\0'; |
| **Hash.c** | **Hash.cu** |
| The ***in[]*** in ***while*** loop need not be iterated for **4096** times. It is enough to compute it once with the nonce and once without the nonce, the result of which can be reused. This reduces the no. of instructions. | |
| while(p+sizeof(uint32_t)*BW<br><=inputSize) {<br>        for(unsigned int q=0; q<BW;<br>q++) {<br>            in[q] = 0;<br>            for(unsigned int w=0;<br>w<sizeof(uint32_t); w++)<br>                in[q] \|=<br>(uint32_t)((unsigned<br>char)(input[p+q*sizeof(uint32_t)+w]<br>)) << (8*w);<br>        }<br>        p += sizeof(uint32_t)*BW;<br>        InputFunction(in);<br>        RoundFunction();<br>    } | temp[id] = (uint32_t)((unsigned<br>char)(tin)) << (8 * (id % 4));<br>            if (id < 8)<br>            {in[id] = 0;<br>             in[id] =<br>temp[id*4]\|temp[id*4+1]\|temp[id*4+2]\|<br>temp[id*4+3];}<br>        p += 4*BW;<br>        InputFunction(in);<br>        RoundFunction();<br>       temp[id] =<br>(uint32_t)((unsigned<br>char)(input[32+id])) << (8 * (id %<br>4));<br>            if (id < 8)<br>            {     in[id] = 0;<br>                  in[id] =<br>temp[id*4]\|temp[id*4+1]\|temp[id*4+2]\|<br>temp[id*4+3]; }<br>      while (p + 4*BW <= inputSize)<br>       {p += 4*BW;<br>        InputFunction(in);<br>        RoundFunction();} |
| A shared variable *ind* is used to manipulate the index of the *b[]* each time the InputFunction() and RoundFunction() are called. This eliminates the loops used for rotating *b[]* thus reducing the instruction count. | |
| uint32_t q[BW]; uint32_t A[MS];<br>    for(unsigned int j=0;j<BW;j++ )<br>        q[j] = b[index2(BL-1,j)];<br>    for(unsigned int i=BL-1; i>0;<br>i--)<br>    for(unsigned int j=0;j<BW; j++)<br>            b[index2(i,j)] =<br>b[index2(i-1,j)];<br>    for(unsigned int j=0; j<BW;<br>j++)<br>        b[index2(0,j)] = q[j];<br>for(unsigned int i=0; i<12; i++)<br>b[index2(i+1,i%BW)] ^= a[i+1];<br>.....<br>.....<br>   for(unsigned int j=0; j<BW; j++)<br>        a[j+13] ^= q[j]; | int id=threadIdx.x;<br>__shared__ uint32_t A[MS];<br>__syncthreads();<br>ind = (ind + 248) % 256;<br><br>if (id<12)<br>      {<br>            b[(index2(id + 1, id% BW)<br>+ ind) % 256] ^= a[id + 1];<br>      }<br><br><br>.....<br>.....<br><br>if (id<8){<br>a[id + 13] ^= b[(ind + id) % 256];} |
| All *for* loops can be eliminated since there are 32 threads that are running in parallel (SIMD). | |
| for(unsigned int i=0; i<MS; i++)<br>A[i] =<br>a[i]^(a[(i+1)%MS]\|(~a[(i+2)%MS]));<br>    for(unsigned int i=0; i<MS;<br>i++)<br>        a[i] = ROR(A[(7*i)%MS],<br>i*(i+1)/2); | <br>A[id]=<br>a[id]^(a[(id+1)%MS]\|(~a[(id+2)%MS]));<br>if(id==0)<br>a[id] = ROR32(A[(7 * id) % MS],<br>((((id*(id + 1) / 2) & 0x1F) + 32) &<br>0x1F)); |

| | |
|---|---|
| `for(unsigned int i=0; i<MS; i++)`<br>`A[i]= a[i]^a[(i+1)%MS]^a[(i+4)%MS];`<br>`    A[0] ^= 1;`<br>`for(unsigned int i=0; i<MS; i++)`<br>`        a[i] = A[i];` | `A[id] =`<br>`a[id]^a[(id+1)%MS]^a[(id+4)%MS];`<br>`if(id==0) A[0] ^= 1;`<br>`a[id] = A[id];` |

## Results

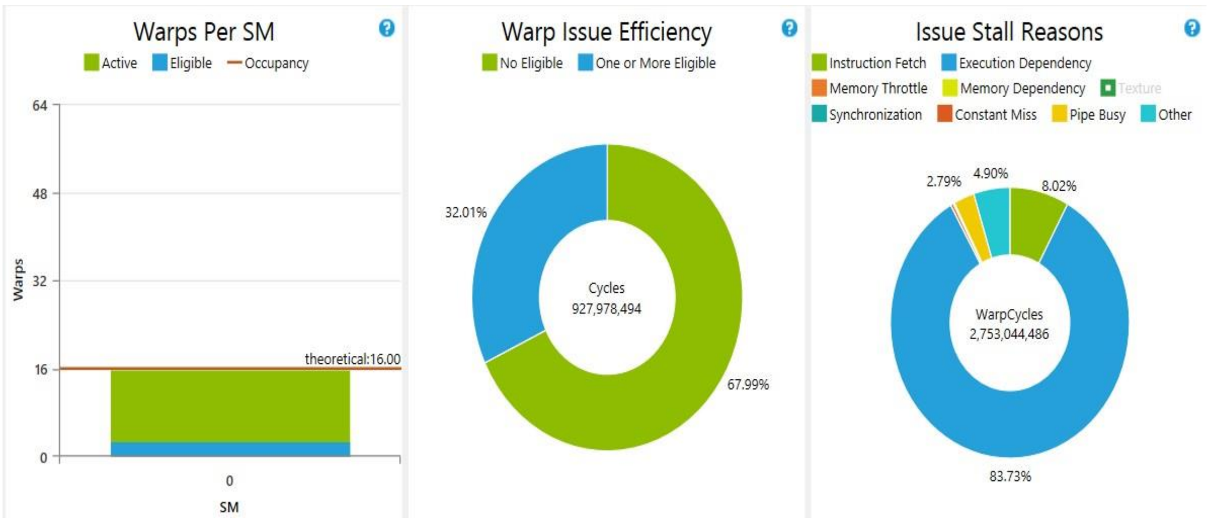
The output of the benchmark of this implementation is shown below:



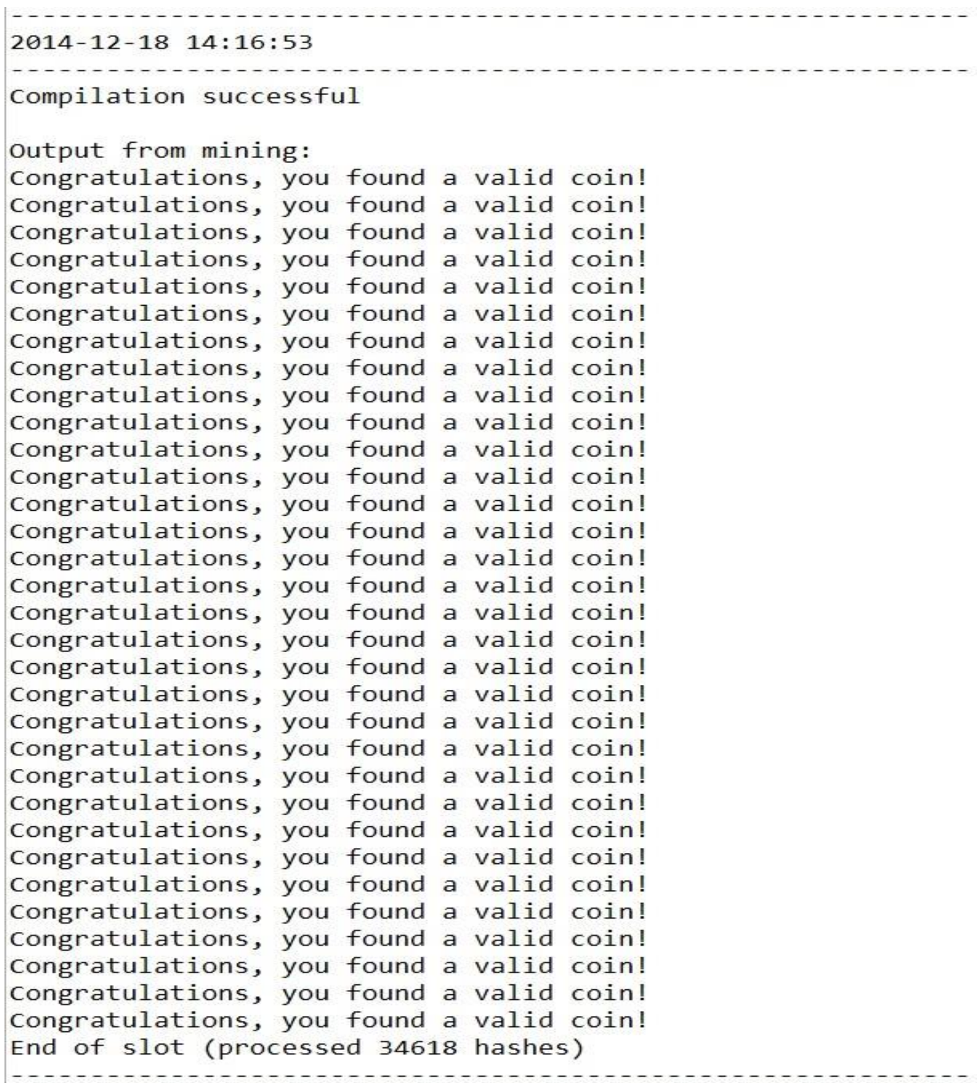
```
----------------------------------------
eca1524@co3:~/production$ ./benchmark.py
3XFE3CPCDTH85EZ2YBKGLBVAMQ6D3VBW 4096 T 0000BF42A634E1FC15804F2B771C
Congratulations, you found a valid coin!
XL41VL430IZ73GGFBCADBJ0WU4RBWV5W 4096 s 000090F03A5797BD06AD936A421B
Congratulations, you found a valid coin!
9YP2DE0NEIFMLZJLUPXS7TJZCMTXD9AT 4096 e 00002CC842C957F132558289934C
Congratulations, you found a valid coin!
SQ3VB1ZUQU4EL2FI3YNOG0R5TBNS6BWH 4096 u 0000703CB5EF0CCB5878A6AD03CC
Congratulations, you found a valid coin!
Processed all inputs
----------------------------------------
Summary:
----------------------------------------
- Processed Blocks:  5000
- Number of Coins:   4
- Running Time:      18 sec
----------------------------------------
```

The application takes **414** seconds with the Native code, while it takes **18** seconds with GPU implementation. **Hence the performance is improved by a factor of 23.**

The output from NSIGHT visual profiler for the above implementation is shown below. We observe that Instruction fetch and Synchronization dependency is significantly reduced in comparison to the implementation with 256 threads.



The output from the competition server is shown below:



```
------------------------------------------------------------
2014-12-18 14:16:53
------------------------------------------------------------
Compilation successful

Output from mining:
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
Congratulations, you found a valid coin!
End of slot (processed 34618 hashes)
------------------------------------------------------------
```

Now it is processing **34618** hashes in the given slot. The performance is improved by a factor of almost 3 in comparison to the 256-threads implementation.

## Conclusion:

1. Performance can be enhanced by
    - Using parallel execution on GPU with restricted memory accesses (__global__ , __device__ , __shared__ & local)
    - Reducing the instructions by optimizing the algorithm
    - Reducing the synchronization of threads ; __syncthreads();
2. Synchronization problems increase with data/execution dependency.
3. In this application use of 32 threads gives
   better occupancy and requires less synchronization.
4. Further improvements are possible if memory accesses can be reduced.
5. The Final implementation of the code is placed in ***/home/eca15/eca1524/production***
6. **The performance is enhanced by a factor of 23 using 62 blocks of 32 threads.**