Embedded Computer Architecture

Assignment 2 – Optimisation of Mining Application using GPU

Sindhuja Chandrasekaran- 0926903

Esakki Raja - 0927494

Contents

Introduction 2

Application to be Optimized 2

CUDA Framework 2

Porting of Application in CUDA 3

Optimization of Application 4

Conclusion 8

Introduction

The purpose of this assignment is to optimize the performance of the given Mining application. This is achieved by utilising the Graphical Processing Unit(GPU) to perform computations that are initially being handled by the CPU. Though the GPU is typically used for image processing, in this case it is used for General Purpose computations, since it offers high level of parallelism and thus improving the efficiency of the program.

The mining application in this assignment is to earn coinporaal coins by solving the hashes. This report explains about mining application & how it is being optimised using CUDA and the analysis of the obtained results.

In section 1, we have discussed about the mining application and its function. The source code of the mining application was provided in C version. Section 2 describes the CUDA framework which helps to port the given C code to CUDA C. This porting helps the host(CPU) to interact with the device(GPU). The porting of the given source code to CUDA has been explained in Section 4.

In the final two sections, we have provided the optimization techniques such as Reduce kernel management overhead, Reduce Memory management overhead, Reduce memory transfer overhead and other bottlenecks that are implemented and the results achieved in the benchmark of the server.

Application to be Optimized

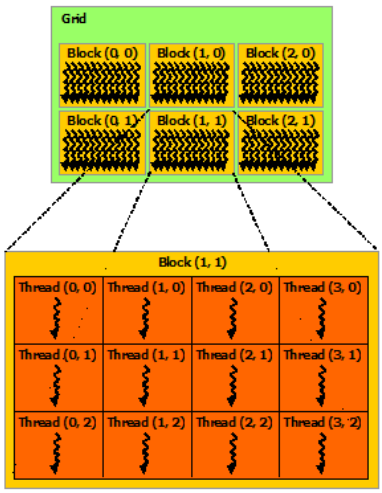
The application requests for a block from the competition server which consists of base string and a multiplier. A nonce string ('a'…'z','A'...'Z','0'…'9') is then appended to the input block. The hash function then produces an output of 64 characters for this input block. It is considered to have found a valid coin if this output starts with '0000'. Hence for one input block, 62 nonces are possible and thus the hash function is being performed 62 times for one input block from the server.

When the hash function is being implemented on GPU, the GPU can implement this function for 62 different nonces simultaneously. This is expected to increase the performance of the application.

CUDA Framework

CUDA is a parallel programming model that can be implemented on GPU. Since it is created by NVIDIA, such programs can only be implemented on CUDA enabled GPU devices.

CUDA has two important components: Kernels and Threads. In CUDA, the parallel portions of an applications are executed on the device as kernels and one kernel is executed at a time. In kernel, the basic unit of execution is thread & they are organized blocks. These blocks are organized in grids. The values of threads, blocks and grids are passed at the launch of the Kernel.



In order to port the provided application into CUDA C , the device memory allocation and the data transfer from host to device has to be taken care of. This is done by calling Cudamalloc() and cudaMemcpy() functions.

Porting of Application in CUDA

The porting has been done for the part of the code that appears as a bottleneck with higher computational complexity. We observed that the hash function is the major computational bottleneck and thus it has been ported to CUDA to implement it in GPU. The hash function is launched as a Hash kernel by the host code. On further analysing, since there are scope for parallelism in stringToHex function, it is also included as a device function and is called by the Hash kernel. Also, the function called by hash function are made as device functions. The rest of the functions(benchmark(), requestInput, ValidateHash) are implemented by the host code. The Hash kernel is being implemented in such a way that the output of the Hash kernel is nothing but the nonce of a found valid coin. The kernel is initially launched with 62 blocks and 1 thread and the number of threads was later increased to 32.

In the following tabulation, we see the code snippets of C-version and ported code of the hash kernel call. The memory is allocated for the input and output in the global memory space which is accessed by both.(CPU & GPU). The input is then copied from host to device before making the kernel call and the output is copied from device to host after the execution of the kernel.

C - Version	CUDA C

//do hash
Hash(input, output_hash);
	char* dev_inputmain;
	char* dev_outputmain;
	char outputmain[2]={'\0', '\0'};
	cudaStatus = cudaMalloc((void**)&dev_inputmain, inputSize * sizeof(char));
	.
	.
	.
	.
	cudaStatus = cudaMemcpy(dev_inputmain, input, inputSize * sizeof(char), cudaMemcpyHostToDevice);
	.....
	HashKernel << < 62, 32 >>

	….. cudaStatus = cudaMemcpy(output, dev_output, 2 * sizeof(char), cudaMemcpyDeviceToHost);

C- Version	CUDA C
while(p+sizeof(uint32_t)*BW <=inputSize) { for(unsigned int q=0; q<BW; q++) { in[q] = 0; for(unsigned int w=0; w<sizeof(uint32_t); w++) in[q] |= (uint32_t)((unsigned char)(input[p+q*sizeof(uint32_t)+w])) << (8*w); } p += sizeof(uint32_t)*BW; InputFunction(in); RoundFunction(); }	while (p + sizeof(uint32_t)*BW <= inputSize) { __syncthreads(); if (thd < 32){ tempIn[thd] = input[thd + p]; } __syncthreads(); if (thd < 8) { in[thd] = 0; unsigned int temp = thd * sizeof(uint32_t); for (unsigned int w = 0; w < sizeof(uint32_t); w++){ unsigned int inputIndex = temp + w; if (inputIndex == 0 && p ==0){ in[thd] |= (uint32_t)((unsigned char)(hashnonce[0])) << (8 * w); } else{ int test = (uint32_t)((unsigned char)(tempIn[inputIndex])) << (8 * w); in[thd] |= (uint32_t)((unsigned char)(tempIn[inputIndex])) << (8 * w); }

Optimization of Application

The code was run initially in the host (CPU) and we checked the output of the benchmark in the server. The output of the benchmark result is shown below. The expectation now is that the ported code should reduce the time taken to find the valid coins for the same number of inputs(5000).

```
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:      445 sec
----------------------------------------
eca1541@co3:~/mining$
```

Figure: Benchmark Results

Implementation with 62 blocks and 1 thread

After porting the code to CUDA C, the code was run in the visual studio, "NSIGHT- Visual Profiler" that provides statistical analysis of the functions and threading involved in the CUDA C. Initially, we have ported our code with **62 blocks and 1 thread** and analysed the result to optimize further.

Kernel call is deployed with 62 blocks with each block processing one particular nonce for the given base and multiplier. Since all of them get processed in parallel, the running time is reduced considerably. The output of the NSIGHT visual profiler and bench mark is shown below.

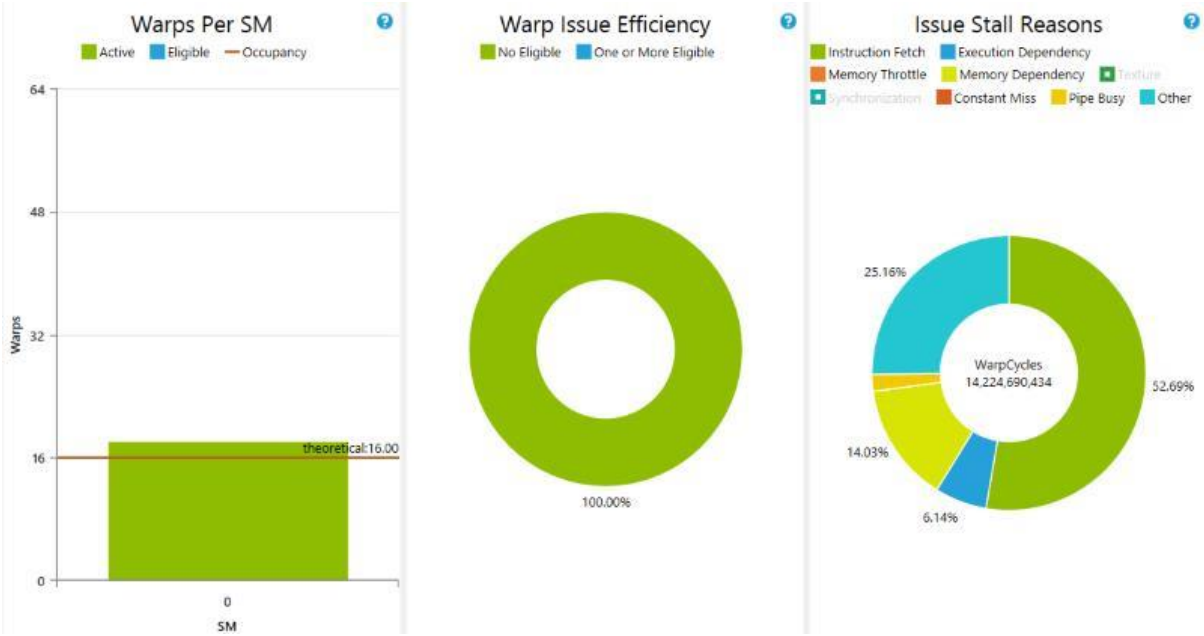


Figure : NSIGHT Visual profiler- 62 Blocks & 1 Thread

Implementation with 62 blocks and 32 thread

By analysing the profiler data of the previous implementation, we took following optimization techniques into motivation and we have ported the code with 62 blocks and 32 threads. This avoids the usage of synchronization threads that reduces the consumption of the GPU cycle. The following methods are the optimization techniques and bottlenecks that are implemented.

1. **Reducing Memory Management Overhead:** The CUDA memory management functions cudaMalloc and cudaFree are more than two orders of magnitude more expensive than the equivalent C standard library functions malloc and free. Hence, by allocating memory once at the beginning of an application and, that memory is reused in each kernel invocation.

2. Accessing shared memory is much faster than global memory. Therefore the input to the hash kernel is first copied as 32 blocks to shared memory, i.e according to the below code snippet each iteration requires 32 characters from the input block to perform computation. Hence instead of accessing input[] (which resides in global memory)for every computation, the required 32 characters are first copied to the shared memory i.e., tempIn[], for which the access rate is much faster.

```
while (p + sizeof(uint32_t)*BW <= inputSize) {
__syncthreads();
if (thd < 32){
tempIn[thd] = input[thd + p]; }
__syncthreads();
if (thd < 8)
{  in[thd] = 0;
unsigned int temp = thd * sizeof(uint32_t);
for (unsigned int w = 0; w < sizeof(uint32_t); w++){
unsigned int inputIndex = temp + w;
if (inputIndex == 0 && p ==0){
in[thd] |= (uint32_t)((unsigned char)(hashnonce[0])) << (8 * w);  }
else{
int test = (uint32_t)((unsigned char)(tempIn[inputIndex])) << (8 * w);
in[thd] |= (uint32_t)((unsigned char)(tempIn[inputIndex])) << (8 * w);
}
```

3. The device function 'RoundFunction' is implemented 32 threads. The rounding of the b array is implemented using threads thus increasing parallelism.

```
__device__ void RoundFunction(uint32_t r_thrd)
{
__shared__ uint32_t q[BW];
if(r_thrd < 8)
{
q[r_thrd] = b[248+r_thrd];
}
__syncthreads();
if(r_thrd<31)
{
unsigned int bIndex1= 248-(r_thrd*8);
unsigned int bIndex2= bIndex1-8;
for(unsigned int j=0; j<BW; j++)
b[bIndex1+j] = b[bIndex2+j];
}
if(r_thrd==31)
{
for(unsigned int j=0; j<BW; j++)
b[j] = q[j];
}
__syncthreads();
```

The output of this implementation in the benchmark is shown below.

```
eca1542@co3:~/production/development$ make benchmark
./benchmark.py
3XFE3CPCDTH85EZ2YBKGLBVAMQ6D3VBW 4096 T 0000BF42A634E1FC15804F2B771CC11B3B5D6FC4D682B8C56CDC46E06CE41FAA
Congratulations, you found a valid coin!
XL41VL430IZ73GGFBCADBJ0WU4RBWV5W 4096 s 000090F03A5797BD06AD936A421B156F7995847E843CDB1800EA2AC7BC72EBF5
Congratulations, you found a valid coin!
9YP2DE0NEIFMLZJLUPXS7TJZCMTXD9AT 4096 e 00002CC842C957F1325582899340C56633A9212384A35D184E5B6C5AF0C753A8D
Congratulations, you found a valid coin!
SQ3VB1ZUQU4EL2FI3YNOG0R5TBNS6BWH 4096 u 0000703CB5EF0CCB5878A6AD03CCF303BD59CF64BB7E5F8D04B85E0C9109D64E
Congratulations, you found a valid coin!
Processed all inputs
----------------------------------------
Summary:
----------------------------------------
- Processed Blocks:  5000
- Number of Coins:   4
- Running Time:      82 sec
----------------------------------------
```

Figure: Benchmark result of implementation with 62 blocks & 32 threads

Benchmark results of C and GPU implementation:

C implementation: 445 secs

GPU implementation: 82 secs

Figure: NSIGHT Visual Profiler – 62 blocks & 32 thread

The above result from NSIGHT visual profiler for 62 blocks & 32 thread implementation shows that instruction fetch is drastically reduced in comparison to the implementation with 1 thread. This implementation is achieved by sharing memory, the reason being the access to shared memory is faster than that of global memory.

Conclusion

However the GPU speeds up the process , the syncthreads function needs to be called before the part that involves sequential programming or the part that can be executed only after all the threads finishes the previous computation. This actually decreased the performance and decreases more when number of threads at the time of kernel is increased. The GPU is therefore efficient at parts of the program where there is much scope for parallelism. The loop unrolling pragma when given at various places did not reduce the time taken by the application. Hence smart utilisation of GPU is required of better optimisation of the application.

References:

[1] NVIDIA Programming Guide

http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3MKfw8Qcr