GPU LAB-2

Name: Riddhi R. Thakker
ID: 201601124
Name: Purva Mhasakar
ID: 201601082

CONTEXT

1. **Description of the problem**
   Vector Addition: Add two vectors are added into one.
   Square Set Numbers: Square a set of numbers.

2. **Complexity of Serial Code:** $\mathcal{O}(n)$ [for both]

3. **Possible Speedup (theoretical):**
   Theoretical Speedup can be defined using Amdahl's Law:

$$S_{Latency} = \frac{1}{(1-p) + \frac{p}{s}}$$

   where:
   $S_{Latency}$ = theoretical speedup
   s = speedup of the part of the task that is parallelized
   p = proportion of execution time that the part benefiting from parallelism

4. **Profiling information:**
   Profiling Information for Vector Addition:

data.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 61.84      0.41     0.41                             main
 39.21      0.68     0.26        1   262.74   262.74  vector_addition

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.
 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.

Copyright C 2012-2014 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.

                     Call graph explanation follows


granularity: each sample hit covers 2 bytes for 1.48% of 0.68 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.41    0.26                 main [1]
                0.26    0.00       1/1           vector_addition [2]
-----------------------------------------------
                0.26    0.00       1/1           main [1]
[2]     38.8    0.26    0.00       1         vector_addition [2]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

Index by function name

   [1] main                    [2] vector_addition
```

Profiling Information for Square of Set of numbers:

data.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 60.67      0.03     0.03        1    30.34    30.34  square_set_no
 40.45      0.05     0.02                             main
 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.

Copyright C 2012-2014 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.

                     Call graph explanation follows


granularity: each sample hit covers 2 bytes for 19.78% of 0.05 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.02    0.03                 main [1]
                0.03    0.00       1/1           square_set_no [2]
-----------------------------------------------
                0.03    0.00       1/1           main [1]
[2]     60.0    0.03    0.00       1         square_set_no [2]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

Index by function name

   [1] main                    [2] square_set_no
```

In profiling of both the programs, it is noticed that the maximum time for execution is required by the vector addition and square set function. Hence, these are functions which needs to be parallelized.

5. **Optimization Strategy:**
   Following factors are to be considered while paralleling a code:

   (a) Dependency:
   There is NO inter dependency between the computations, hence the vector addition as well as square can be parallelized easily.

   (b) Locality of Reference (or Re-usability):
   Here in both the programs, a particular memory location is accessed only once and hence, there is NO re-usability i.e. temporal locality is not present. However, there is considerable spatial locality present in both these programs.

   (c) Optimization (Scope of parallelization):
   Only one function is present in both the programs and that too are simple addition and multiplication programs, hence, apart from parallelizing memory accesses much cannot be done in the part of computation. So, there is limited scope of parallelization present in these programs.

   (d) CGMA (Compute to Global Memory Access):
   As we are not parallelizing the computation part, the number of computations remains same as they were in serial code. Only the net time to memory access decreases, and not the number of memory accesses required. Hence, throughput will increase but CGMA will remain unchanged.

   (e) Synchronization:
   The programs are free from all kind of dependencies which lead to asynchronization (such as: flow, anti, input, output, loop carried). Hence, synchronization between the threads is not required in this case.

   (f) Control Divergence:
   The threads are organized along single dimension only and the number of threads that are launched are also multiples of 32 (warp size). Thus, control divergence is not present here.

   (g) Thread Organization:
   How many threads to launch in a block and how many blocks to launch in grid will be determined by the hardware properties of the device. (Discussed in later section)

6. **Problems in parallelization and possible solutions:**
   The main important problem while parallelizing is the identify a pattern and then, verifying the dependencies among them. Afterwards, the thread organization is another problem, which has to be solved considering the hardware properties of the device. It is necessary to ensure that there is as less as possible control divergence presence.

HARDWARE DETAILS

For generating the hardware features, do the following:
There is file name **hw_properties.cu** which is written in cuda, to get the CUDA device properties. Following command should be used to compile this file:

```
nvcc hw_properties -o hw_properties
./hw_properties > hw_properties.txt
```

These commands are includes in run_script.sh shell script.
The hardware features of the CUDA Device:

1. Number of CUDA devices: 1

CUDA Device #0

1. GPU card's name: GeForce GTX 680
2. Total Global Memory: 2147287040 Bytes
3. Maximum Threads per Block: 1024
4. Maximum Threads Dimension in X-axis: 1024
5. Maximum Threads Dimension in Y-axis: 1024
6. Maximum Threads Dimension in Z-axis: 64
7. Maximum Grid Size in X-axis: 2147483647
8. Maximum Grid Size in Y-axis: 65535
9. Maximum Grid Size in Z-axis: 65535
10. Warp Size: 32 Threads
11. Clock Rate: 1058500 kHz
12. Shared Memory Per Block: 49152 Bytes
13. 32-bit Registers Per Block: 65536

The CPU features:

1. Architecture: x86_64
2. CPU op-mode(s): 32-bit, 64-bit
3. Byte Order: Litte Endian
4. CPU(s): 24
5. On-line CPU(s) list: 0-23

6. Thread(s) per core: 2
7. Core(s) per socket: 6
8. Socket(s): 2
9. NUMA node(s): 2
10. Vendor ID: GenuineIntel
11. CPU family: 6
12. Model: 63
13. Model name: Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz
14. Stepping: 2
15. CPU MHz: 1416.375
16. BogoMIPS: 4804.69
17. Virtualization: VT-x
18. L1d cache: 32K
19. L1i cache: 32K
20. L2 cache: 256K
21. L3 cache: 15360K
22. NUMA node0 CPU(s): 0-5,12-17
23. NUMA node1 CPU(s): 6-11,18-23

The broad differences between the CPU and GPU are:
there is NO cache levels present in the GPU unlike CPU, though GPU have shared memory to facilitate the quick memory accesses
the number of threads in CPU as compare to that in GPU are drastically less

INPUT PARAMETERS & OUTPUT

OBSERVATIONS

1. Comparison of Serial & Parallel Code

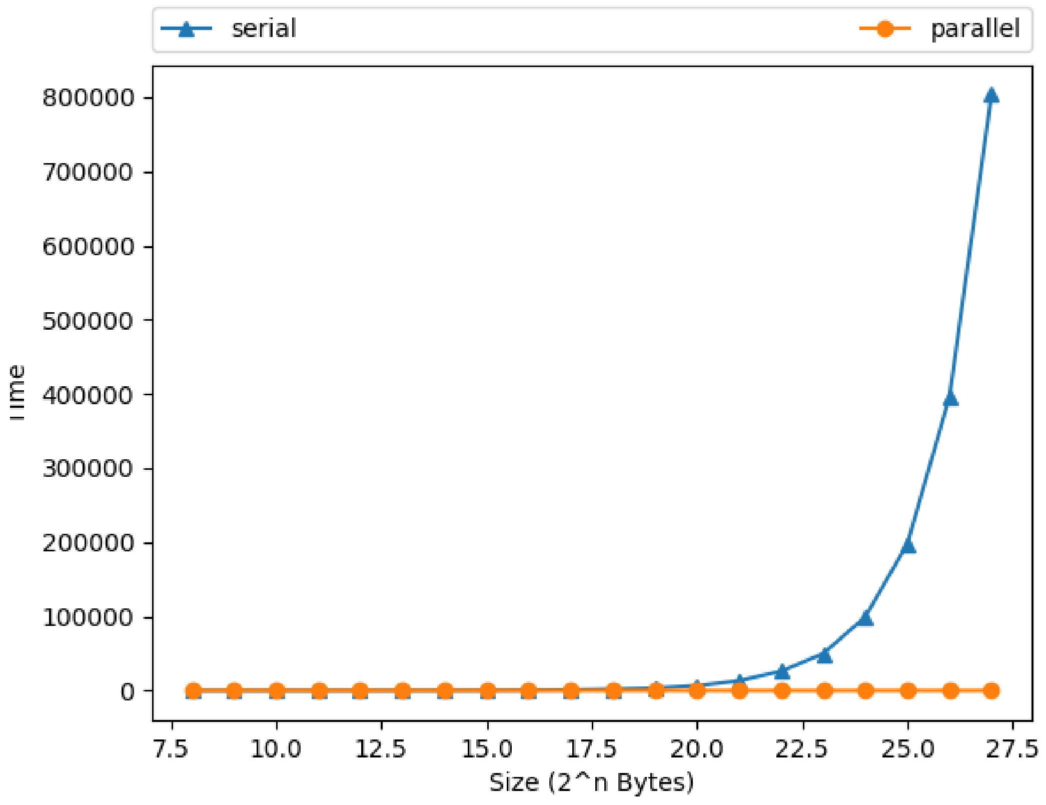

Figure 1: Time VS Problem Size for Vector Addition

The time taken by serial code increases exponentially after 2^{20} where as time taken by parallel code remains almost same throughout the experiment.

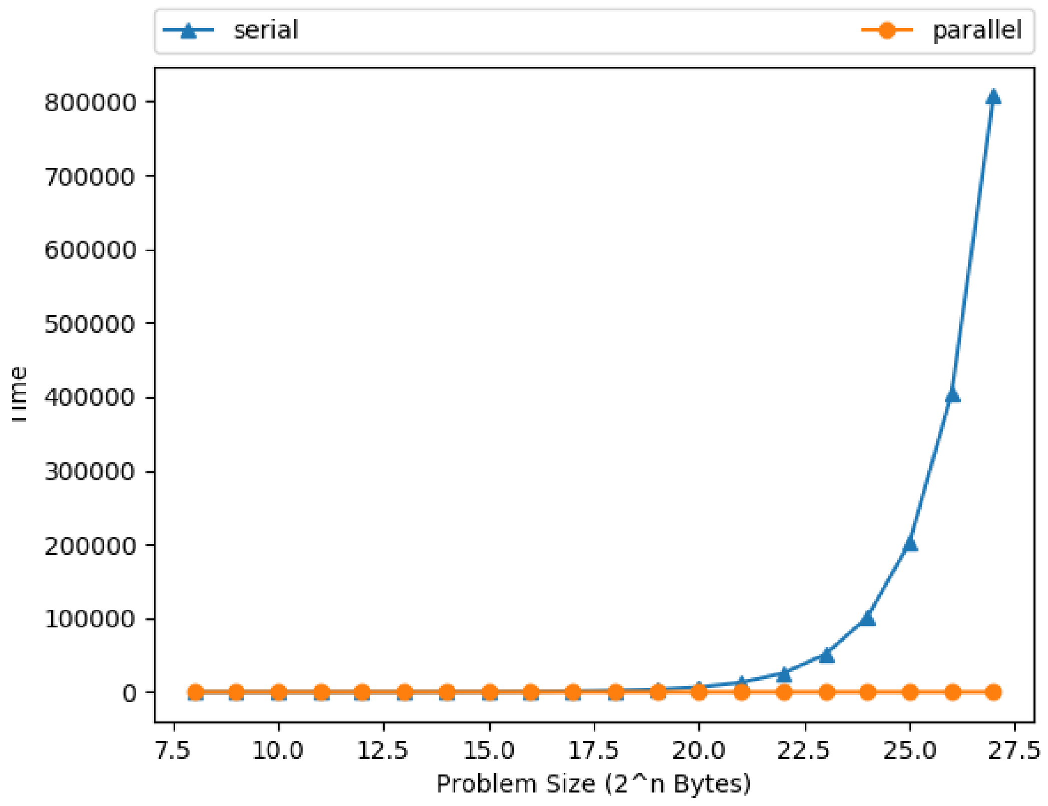

Figure 2: Time VS Problem Size for Square of Vector

2. Speedup Curve

   $Speedup = SerialTime/CUDATime$

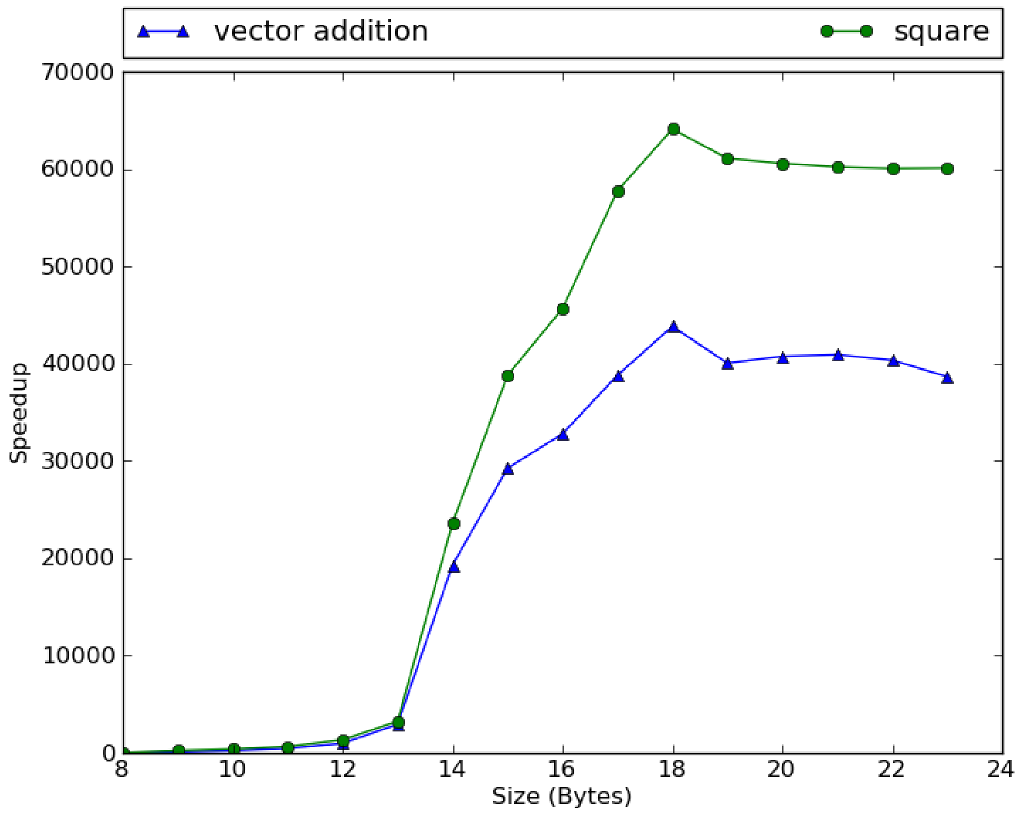

Figure 3: Speedup VS Problem Size

The speedup of square of set of numbers is more than that of vector addition because the serial time for square is comparatively more than that of vector addition.

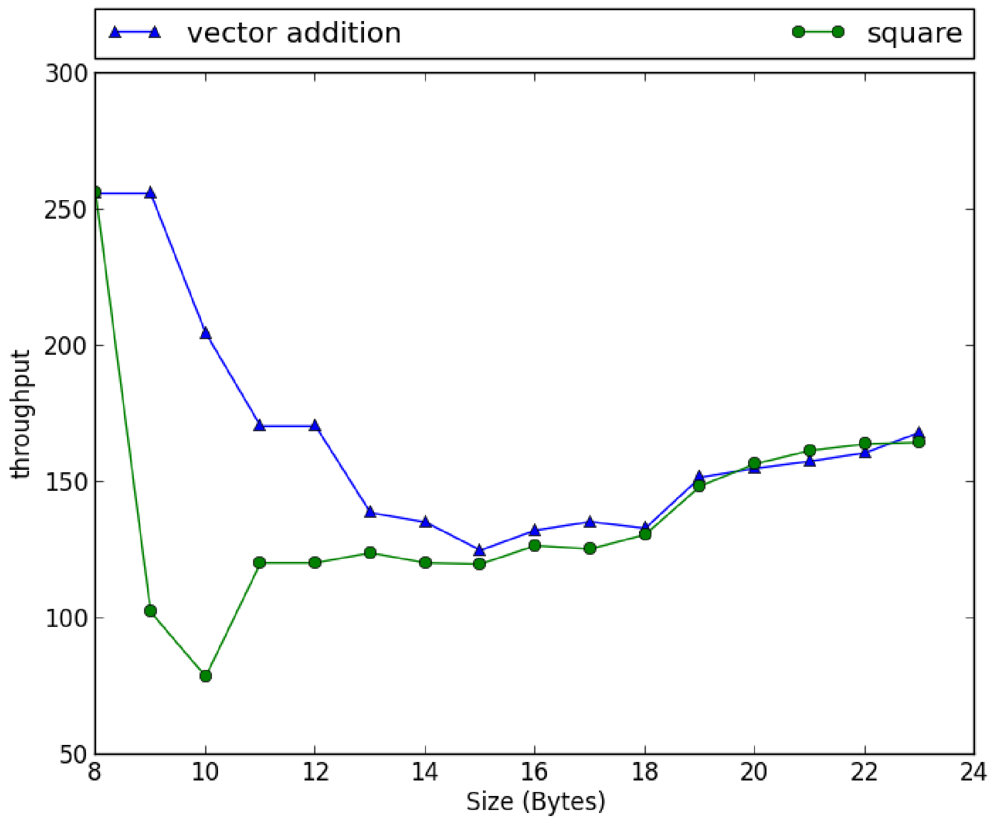

Figure 4: Throughput VS Problem Size

3. Effect of Block Size on Speedup of Vector Addition
   The maximum threads per block are 1024. This means at any given point of time, we cannot launch more than 1024 threads simultaneously. Hence, we have changed the block size in the range of 128 to 1024 (in the multiples of 2). On launching lesser than 1024 threads, the number of blocks to be launched together in the grid increases and even there will be equivalent amount of inactive threads in the blocks, which is not an efficient design. The maximum utilization of the resources will be done when we launch 1024 threads together in a block, this will also decrease the number of blocks in the grid. Though, synchronization is not needed in this code as there is no dependency between the results, yet it is necessary to synchronize all the threads across all the blocks to finish the program. As the number of blocks increases, there will be more number of threads across many blocks to synchronize which will be overhead in time analysis. Moreover, there is upper bound over the number of blocks that can be launched in single SM, which will lead the program not to

execute parallel rather some serialization will be imposed into the execution. This serialization will add to the total time. Thus, the optimum solution is to utilize the resources in such a way that it does not lead to much of serialization.

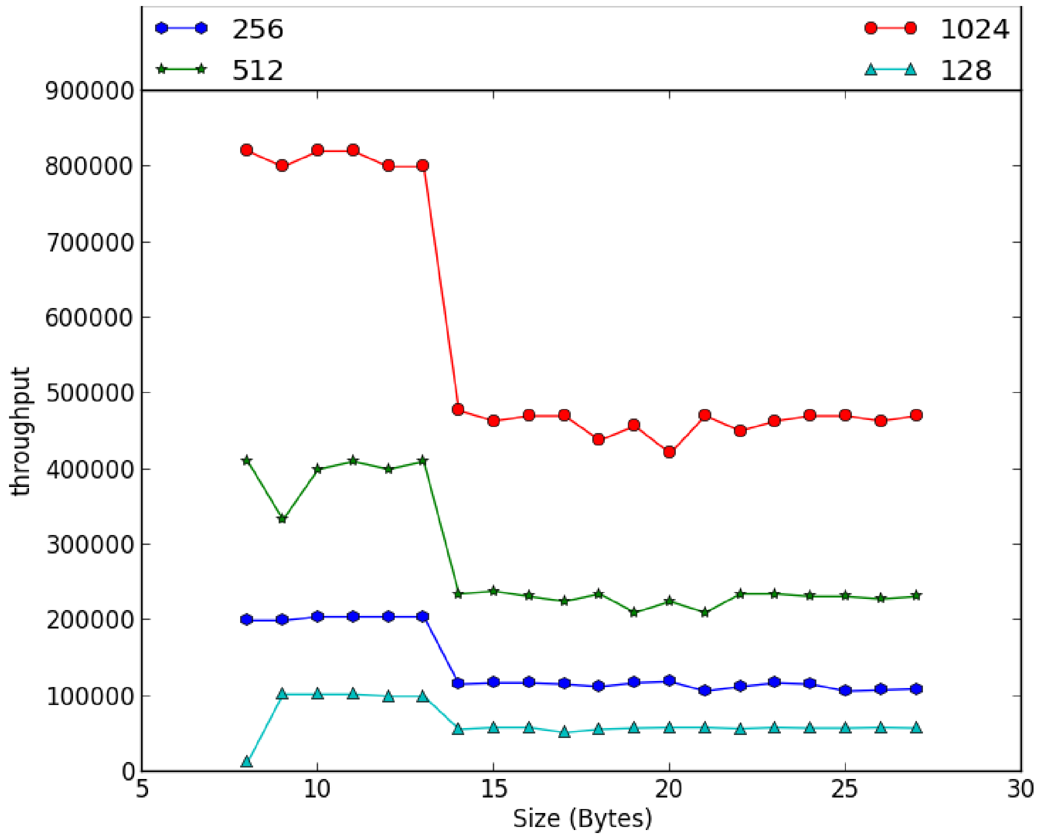

Figure 5: Throughput VS Problem Size for different block size

In the above graph, there is a major drop at problem size 2_{14}. This is because 8 blocks can be launched at a time in one SM, and the number of block with 1024 (threads) block size will be 16, twice the number allowed. Hence, we have to execute the program into 2 phases.

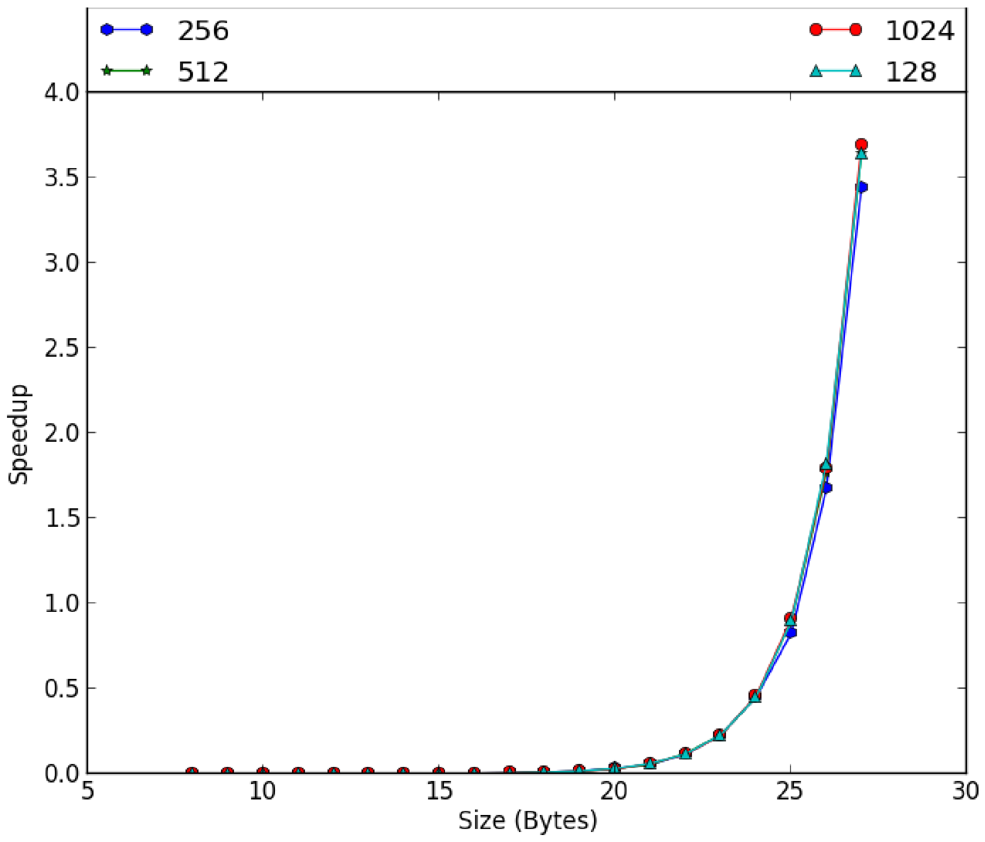

Figure 6: Speedup VS Problem Size for different block size

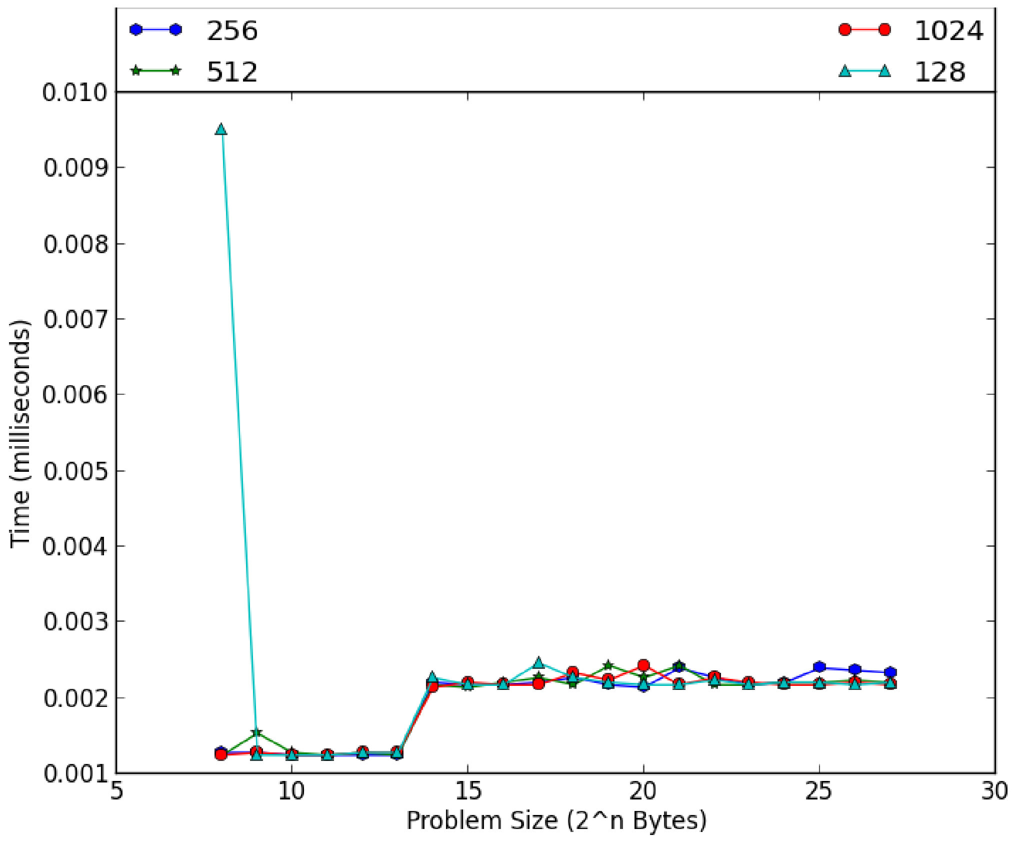

Figure 7: Time VS Problem Size for different block size

With block size greater than 1024, it was observed that the results were incorrect and the code did not execute correctly.

4. Measurement of performance in terms of MFLOPS:

$$= \frac{(\text{problem size}) * (\text{size of each element}) * 10^{-3}}{\text{runtime (ms)}}$$