1. **Objective:**

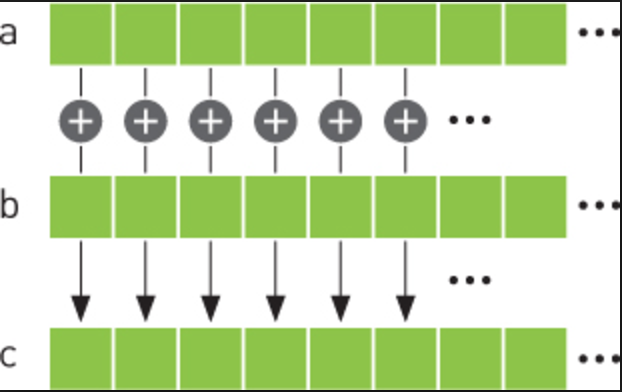
Main objective of this program is to understand the hardware of the GPU machines in the lab and understand the performance improvement after parallelization through various versions of vector-vector addition.

1. **Implementation:**

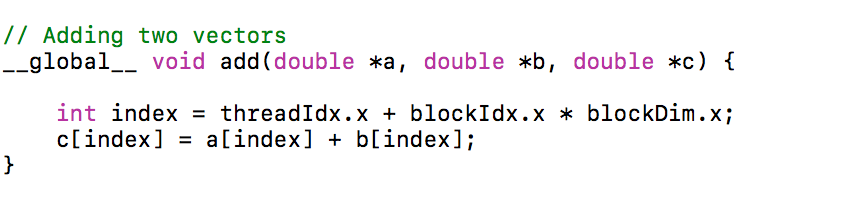
In this vector-vector addition, I have considered two vectors A and B of same sizes and stored the result in C. I have implemented it various ways to understand parallelism and optimize the vector-vector addition.

* 1. **Naïve Implementation of Vector addition:**

In this approach, I have used 1D block and 1D threads. On GPU, one thread is assigned to compute one element of the resulting vector. Each thread loads element of the vector from global memory, apply addition and the store the result to resulting vector in global memory.

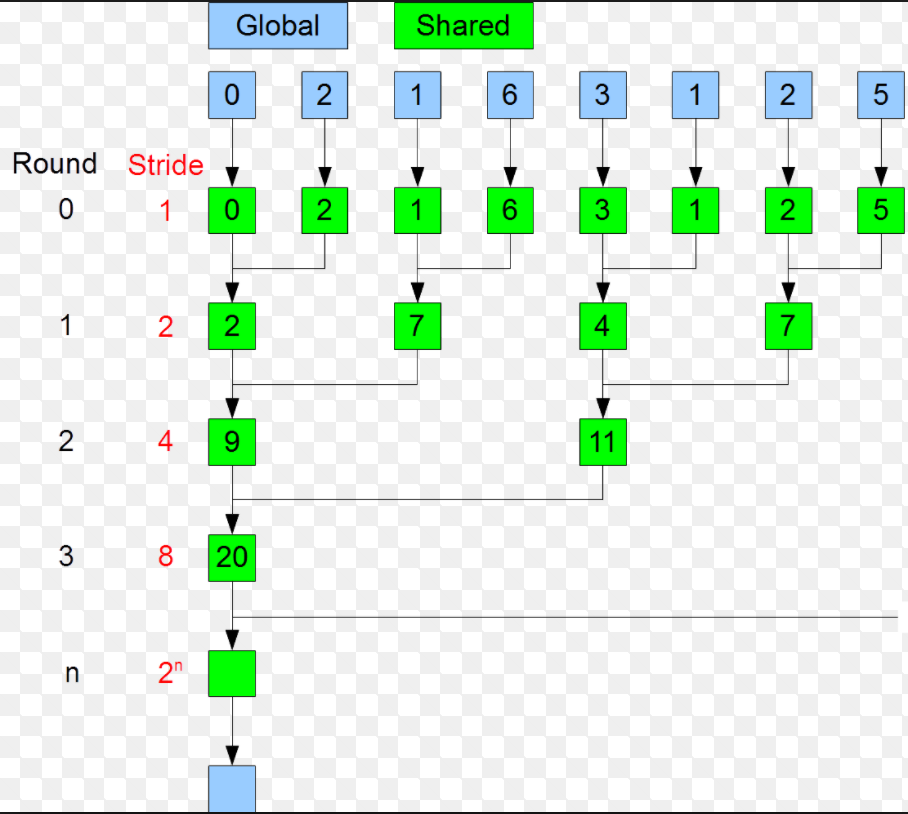


Please see the code snippet for more clear understanding:

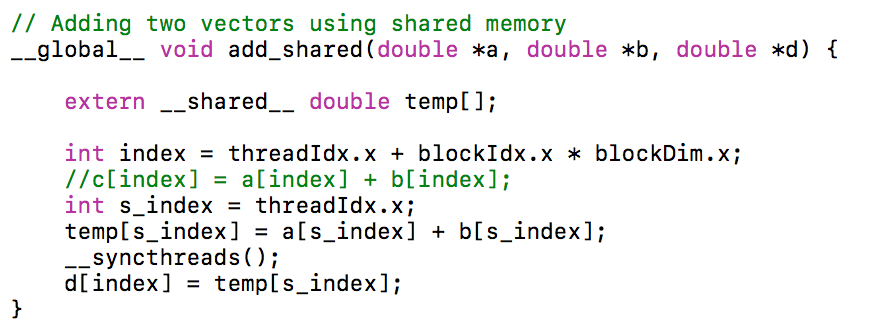


* 1. **Implementation using Shared memory:**

In this approach, a temporary vector variable is created in the shared memory. The scope of this variable is the block so all the addition takes place in the shared memory and the result is then written back to global memory. This approach reduces time

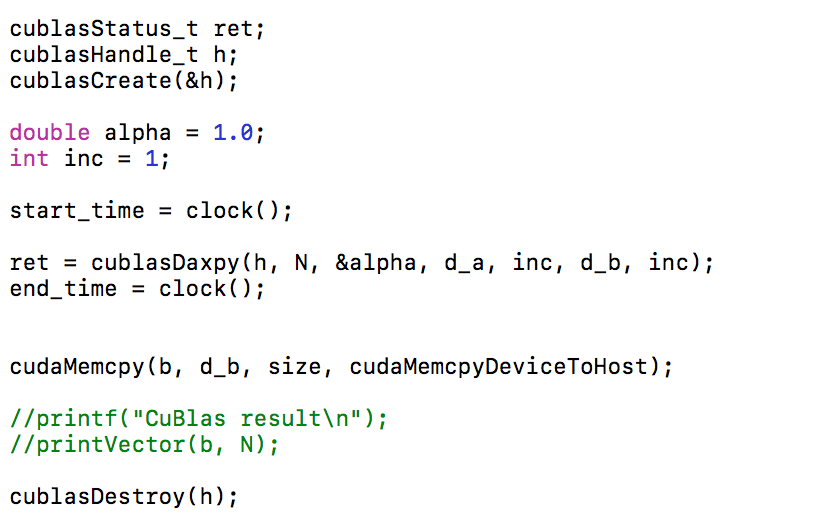


Please see code snippet for this approach:



* 1. **Implementation using cuBLAS:**

This cuBLAS is a library already installed with nvidia toolkit. I have used this to measure my results against cuBLAS.



1. **Output Screenshots:**

**System Specifications:**

GPU Specifications: Tesla M2090: 2.0

Global memory: 5331mb

Shared memory: 48kb

Constant memory: 64kb

Block registers: 32768

Warp size: 32

Threads per block: 1024

Max block dimensions: [ 1024, 1024, 64 ]

Max grid dimensions: [ 65535, 65535, 65535 ]

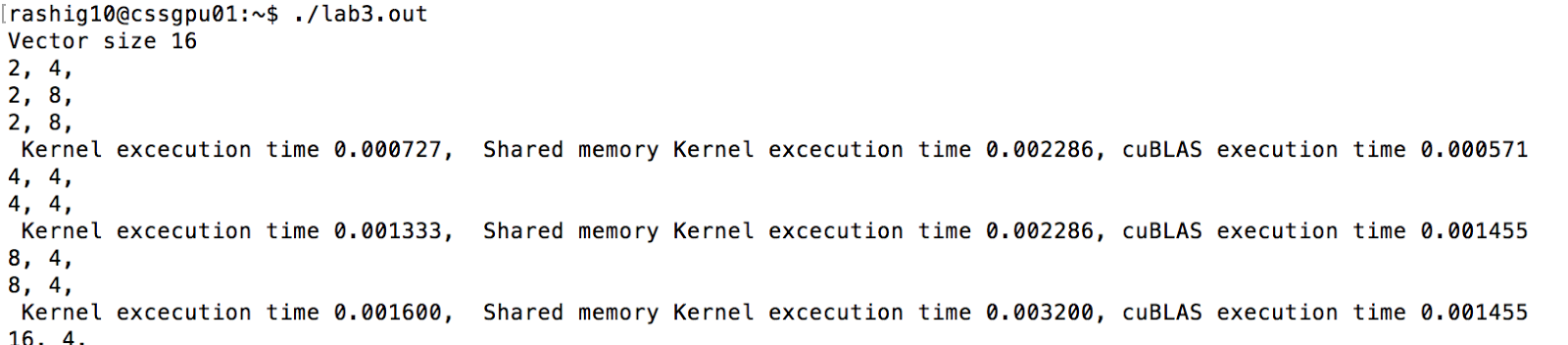
Number of multiprocessors: 16  
Number of CUDA cores: 512

Max mem pitch: [2147483647](about:blank)Memory Clock Rate (GHz): 1.848000  
Memory Bus Width (bits): 384  
Peak Memory Bandwidth (GB/s): 177.408000

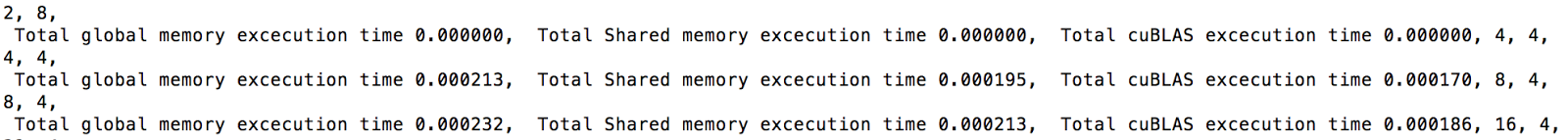
Theoretical Peak Performance: 665 GFlops

**3.1 Output for Vector Size = 16**

**Computational Time:**

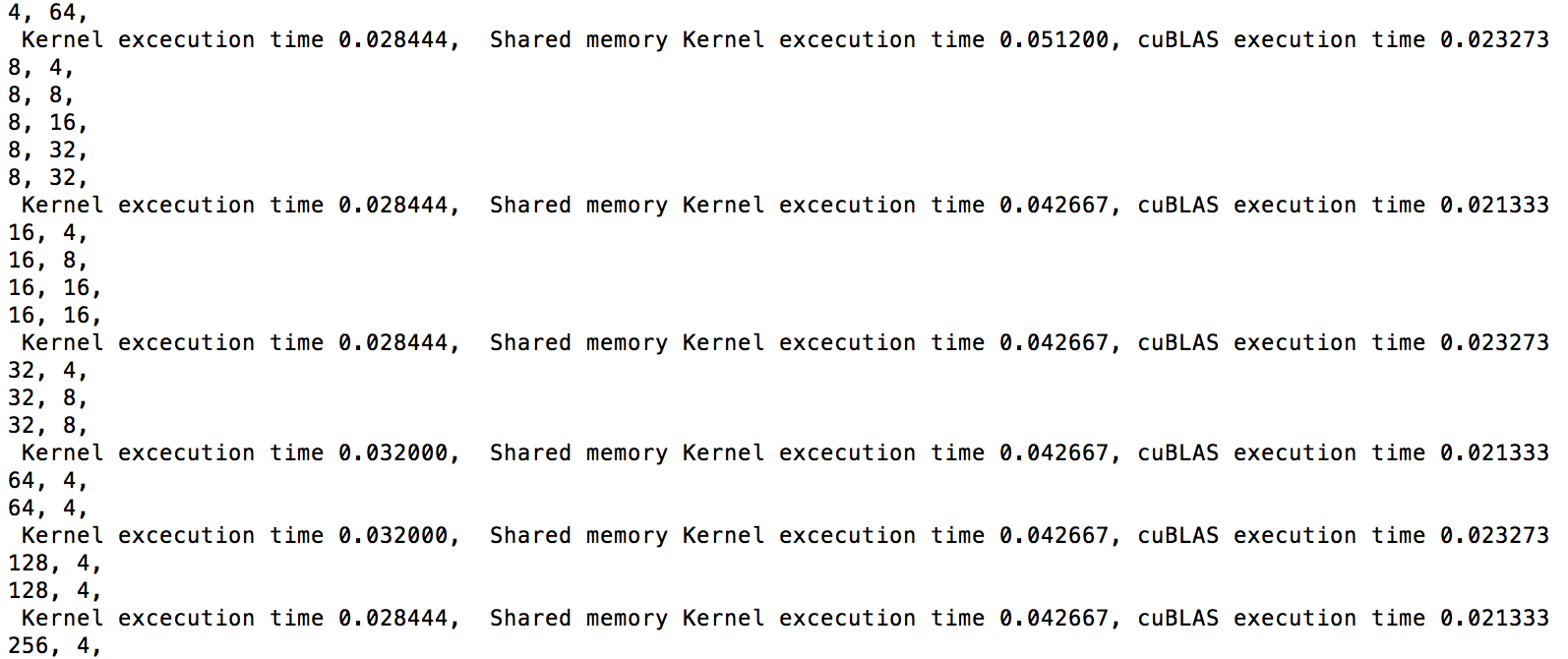


**Total Time:**

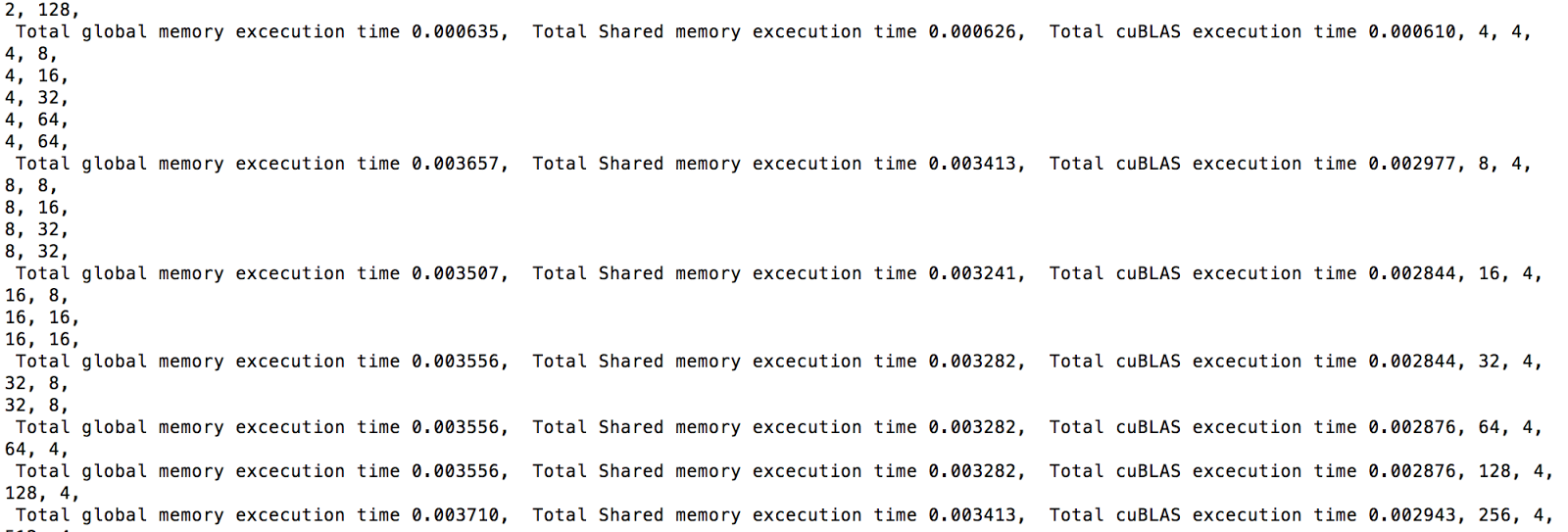
****

**3.2 Output for Vector Size = 256**

**Computational Time:**

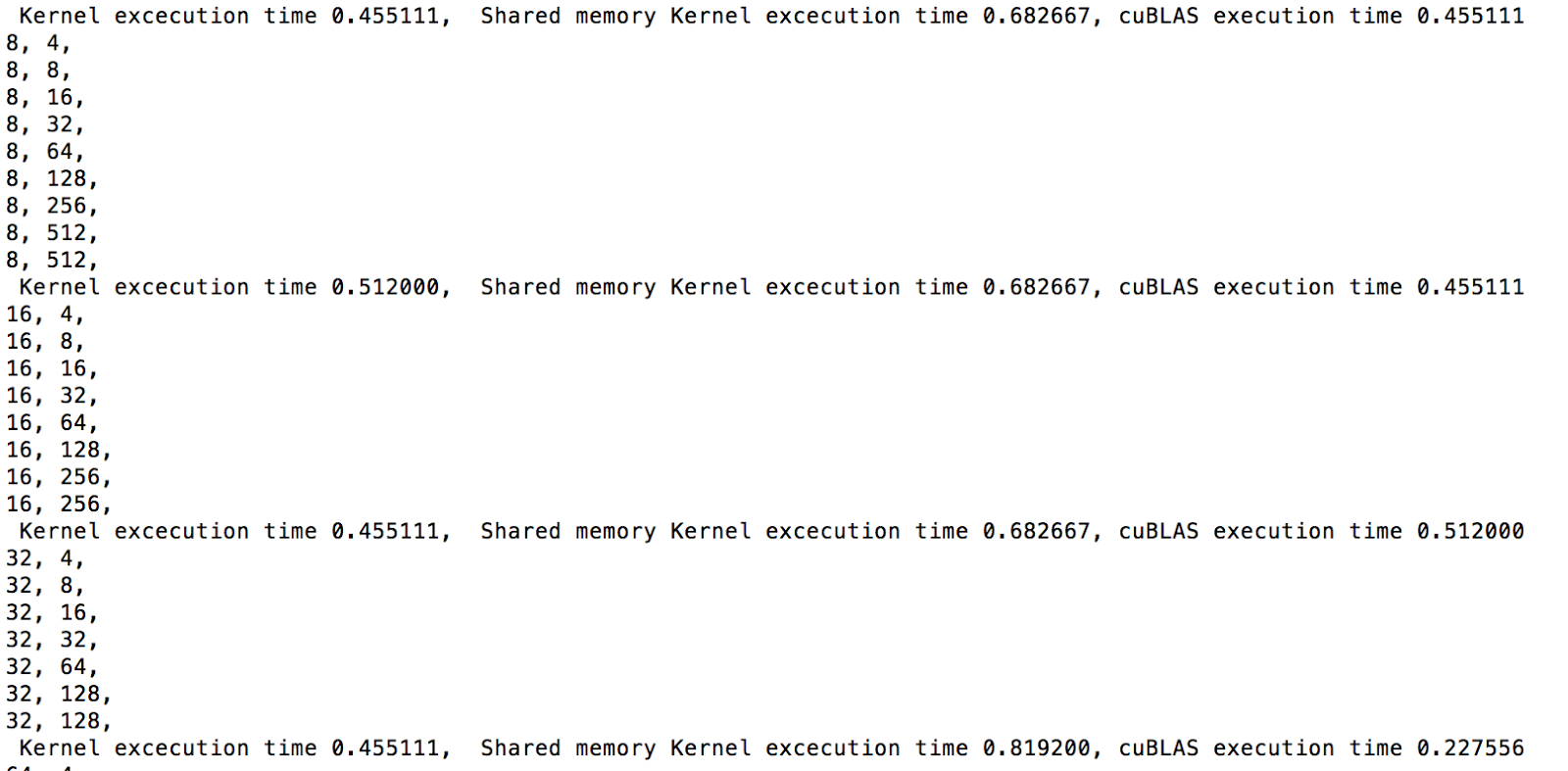
****

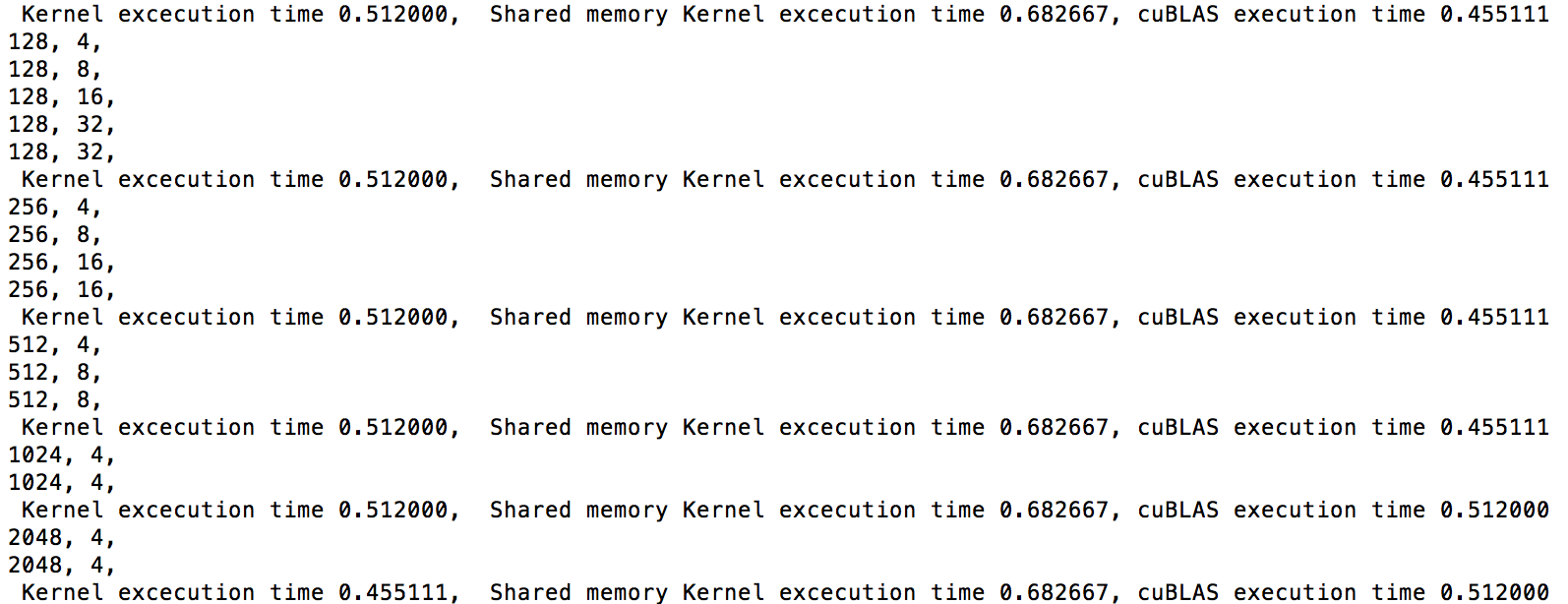
**Total Time:**

****

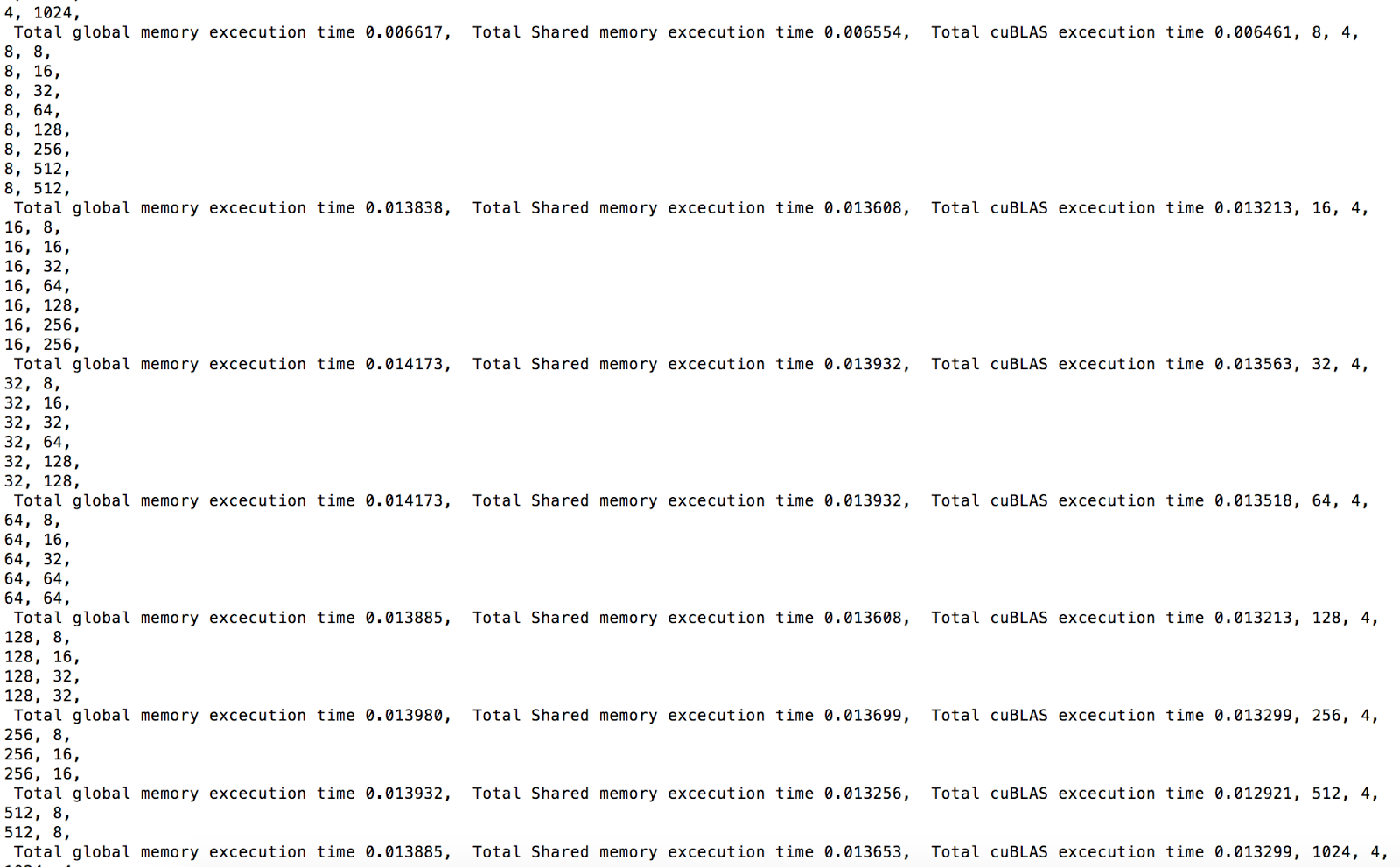
**3.3 Output for Vector Size = 4096**

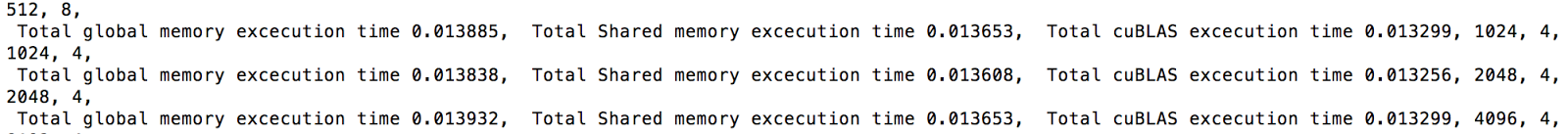
**Computational Time:**

****

****

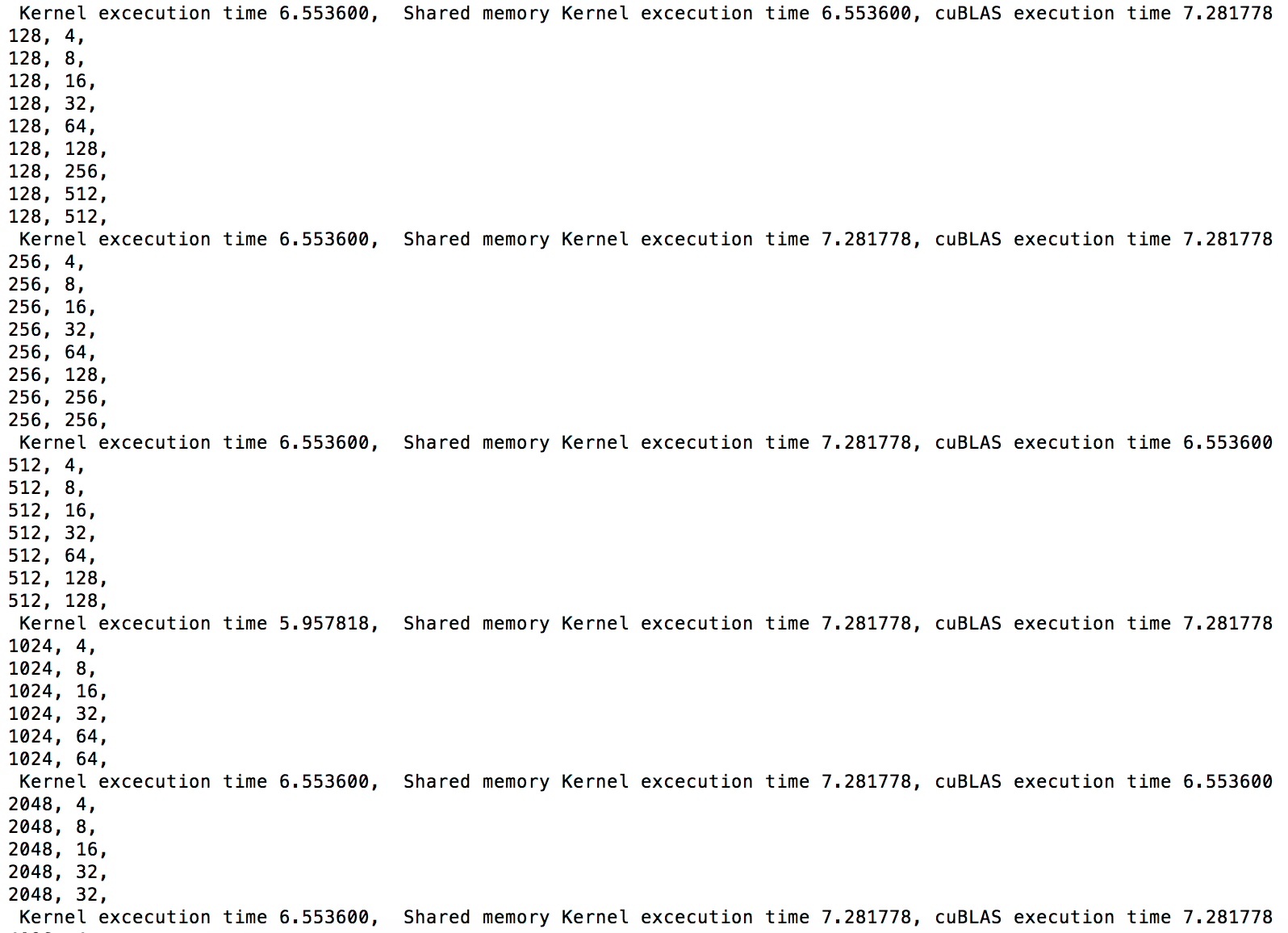
**Total Time:**

****

****

**3.4 Output for Vector Size = 65536**

**Computation Time:**

****

**Total Time:**

****

1. **Benchmarking:**

**Table 1: Some result values for different sizes and their computational time depending on the implementation**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vector size | Number of blocks | Number of threads | global memory kernel execution in Gflop/s | shared memory kernel execution in Gflop/s | cuBLAS performance in Gflop/s |
| 16 | 2 | 8 | 0.000727 | 0.002286 | 0.000571 |
| 16 | 4 | 4 | 0.001333 | 0.002286 | 0.001455 |
| 16 | 8 | 4 | 0.0016 | 0.0032 | 0.001455 |
| 256 | 2 | 128 | 0.028444 | 0.042667 | 0.023273 |
| 256 | 4 | 64 | 0.028444 | 0.0512 | 0.023273 |
| 256 | 8 | 32 | 0.028444 | 0.042667 | 0.021333 |
| 256 | 16 | 16 | 0.028444 | 0.042667 | 0.023273 |
| 256 | 32 | 8 | 0.032 | 0.042667 | 0.021333 |
| 256 | 64 | 4 | 0.032 | 0.042667 | 0.023273 |
| 256 | 128 | 4 | 0.028444 | 0.042667 | 0.021333 |
| 4096 | 4 | 1024 | 0.455111 | 0.682667 | 0.455111 |
| 4096 | 8 | 512 | 0.512 | 0.682667 | 0.455111 |
| 4096 | 16 | 256 | 0.455111 | 0.682667 | 0.512 |
| 4096 | 32 | 128 | 0.455111 | 0.8192 | 0.227556 |
| 4096 | 64 | 64 | 0.512 | 0.682667 | 0.455111 |
| 4096 | 128 | 32 | 0.512 | 0.682667 | 0.455111 |
| 4096 | 256 | 16 | 0.512 | 0.682667 | 0.455111 |
| 4096 | 512 | 8 | 0.512 | 0.682667 | 0.455111 |
| 4096 | 1024 | 4 | 0.512 | 0.682667 | 0.512 |
| 4096 | 2048 | 4 | 0.455111 | 0.682667 | 0.512 |
| 65536 | 64 | 1024 | 6.5536 | 6.5536 | 7.281778 |
| 65536 | 128 | 512 | 6.5536 | 7.281778 | 7.281778 |
| 65536 | 256 | 256 | 6.5536 | 7.281778 | 6.5536 |
| 65536 | 512 | 128 | 5.957818 | 7.281778 | 7.281778 |
| 65536 | 1024 | 64 | 6.5536 | 7.281778 | 6.5536 |
| 65536 | 2048 | 32 | 6.5536 | 7.281778 | 7.281778 |
| 65536 | 4096 | 16 | 6.5536 | 7.281778 | 6.5536 |
| 65536 | 8192 | 8 | 6.5536 | 6.5536 | 6.5536 |
| 65536 | 16384 | 4 | 6.5536 | 7.281778 | 7.281778 |
| 65536 | 32768 | 4 | 5.957818 | 7.281778 | 7.281778 |

**Table 2: Some result values for different sizes and their total implementation time depending on the implementation**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vector size | Number of blocks | Number of threads | Global memory total time   execution in Gflop/s | Shared  memory total time execution in Gflop/s | cuBLAS total time performance in Gflop/s |
| 16 | 2 | 8 | 0 | 0 | 0 |
| 16 | 4 | 4 | 0.000213 | 0.000195 | 0.00017 |
| 16 | 8 | 4 | 0.000232 | 0.000213 | 0.000186 |
| 256 | 2 | 128 | 0.000635 | 0.000626 | 0.00061 |
| 256 | 4 | 64 | 0.003657 | 0.003413 | 0.002977 |
| 256 | 8 | 32 | 0.003507 | 0.003241 | 0.002844 |
| 256 | 16 | 16 | 0.003556 | 0.003282 | 0.002844 |
| 256 | 32 | 8 | 0.003556 | 0.003282 | 0.002876 |
| 256 | 64 | 4 | 0.003556 | 0.003282 | 0.002876 |
| 256 | 128 | 4 | 0.00371 | 0.003413 | 0.002943 |
| 4096 | 4 | 1024 | 0.006617 | 0.006554 | 0.006461 |
| 4096 | 8 | 512 | 0.013838 | 0.013608 | 0.013213 |
| 4096 | 16 | 256 | 0.014173 | 0.013932 | 0.013563 |
| 4096 | 32 | 128 | 0.014173 | 0.013932 | 0.013518 |
| 4096 | 64 | 64 | 0.013885 | 0.013608 | 0.013213 |
| 4096 | 128 | 32 | 0.01398 | 0.013699 | 0.013299 |
| 4096 | 256 | 16 | 0.013932 | 0.013256 | 0.012921 |
| 4096 | 512 | 8 | 0.013885 | 0.013653 | 0.013299 |
| 4096 | 1024 | 4 | 0.013838 | 0.013608 | 0.013256 |
| 4096 | 2048 | 4 | 0.013932 | 0.013653 | 0.013299 |
| 65536 | 64 | 1024 | 0.0136 | 0.013574 | 0.013549 |
| 65536 | 128 | 512 | 0.014544 | 0.014515 | 0.014486 |
| 65536 | 256 | 256 | 0.015667 | 0.015637 | 0.015604 |
| 65536 | 512 | 128 | 0.01569 | 0.015656 | 0.015619 |
| 65536 | 1024 | 64 | 0.01704 | 0.017 | 0.016961 |
| 65536 | 2048 | 32 | 0.015678 | 0.015645 | 0.015611 |
| 65536 | 4096 | 16 | 0.017076 | 0.017036 | 0.016996 |
| 65536 | 8192 | 8 | 0.015739 | 0.015705 | 0.015667 |
| 65536 | 16384 | 4 | 0.017134 | 0.017093 | 0.017053 |
| 65536 | 32768 | 4 | 0.015709 | 0.015675 | 0.015641 |

1. **Discussions on Performance Tuning Based on Memory:**

**GFLOP**S =           NUMBER OF FLOATING POINT OPERATIONS

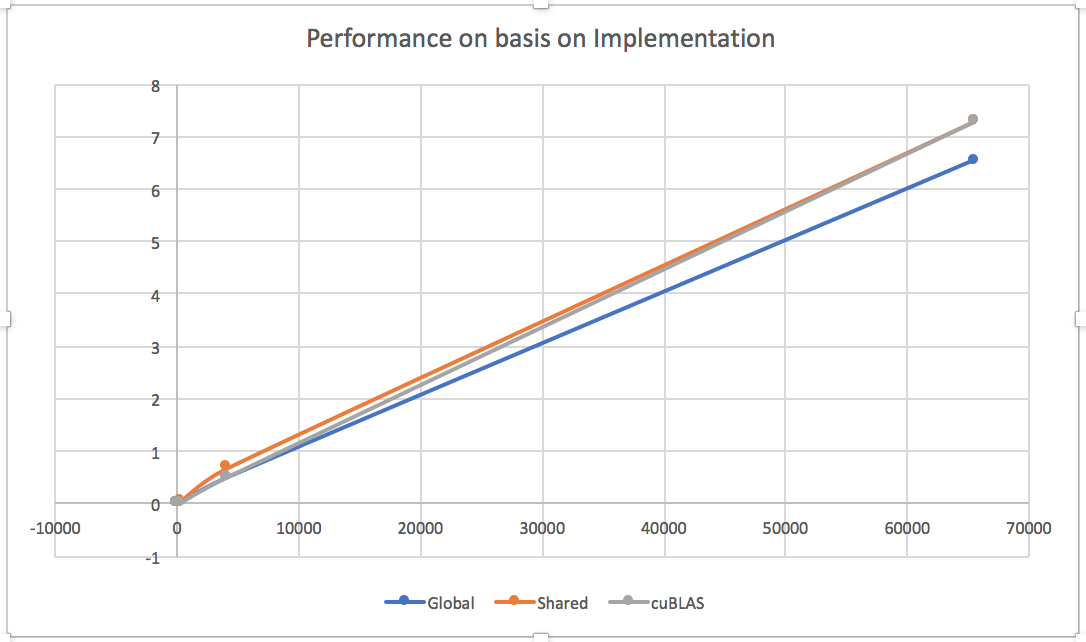
EXECUTION TIME TO ADD VECTORS (seconds)

**FLOPS** = Size of the Vector which is N

**x- axis represents Vector sizes**

**y- axis represents Gflop/s**

**Graph 1 : Performance Tuning based on computation time execution and Memory considerations**

****

In the graph shown above, GFLOPS values are plotted against increasing vector sizes. We are considering the best performance configuration for each vector size to plot in this graph ( eg, for vector size 16, number of blocks = 8 and number of threads = 4 gives the best performance but it can be different for different vector size). From this graph, we can observe that:

* As the size of the vector increases, performance of all the 3 implementations increase
* Global memory shows improvement in performance with increase in vector size but is lower than both the other implementations because threads have to access global memory repeatedly increasing the latency. Hence, the performance is little lower.
* Using shared memory improves the locality of the program. Threads do not have to access global memory again and again. The addition of the elements happens in shared memory ensuring that data used in a loop stays in a cache until it is reused.
* cuBLAS runs slower than the other implementations for matrix size around N <= 256. This is probably due to the fact that the cost of creating and destroying the cuBLAS handle for the subroutine cublasDaxpy call is quite significant. It is only for N > 256, that cuBLAS is faster than all other implementations

**Graph 2 : Performance Tuning based on total time execution and Memory considerations**

**x- axis represents Vector sizes**

**y- axis represents Gflop/s**

****

For computing GFLOPs here, we are taking total number of operations as N considering total memory computations are negligible.  From this graph we can observe that:

* When we include memory transaction time with computation time, the total time goes up significantly since memory transactions are the bottleneck
* Since the total time has increased and total number of operations is still N, the performance goes down significantly
* For global memory, the performance is the worst because with threads have to access global memory repeatedly, which increases the latency.
* The performance with shared memory is better than with global memory because using shared memory improves locality.  Threads do not have to access global memory again and again. The addition of the elements happens in shared memory ensuring that data used in a loop stays in a cache until it is reused.

1. **Discussions on Performance Tuning Based on Registers:**

**x- axis represents Vector sizes**

**y- axis represents Gflop/s**

**Graph 3: Performance Tuning based on Registers with computational time execution**



From above graph, we can observe that:

* Execution of Loop unrolling is almost overshadowing the performance of Shared memory implementation.
* Calculating the sum in registers significantly reduces the communication between threads for large vector sizes. Hence it is faster than the implementation using global memory.
* We can infer from this that reuse of registers is not increasing the performance as it ideally should.
* However since the program complexity is very less, we aren’t able to leverage the full potential of registers and therefore the computation time for loop unrolling approach using registers is comparable to the one using shared memory.

**Graph 4: Performance Tuning based on Registers  with total time execution**

****

For computing GFLOPs here, we are taking total number of operations as N considering total memory computations are negligible.  From this graph we can observe that:

* If we are calculating performance for Loop Unrolling with total execution time, performance is really poor.
* This is because loop unrolling is not performing significantly for vector addition and it consumes more time to finish vector additions as the number of threads are reduced by factor of unroll n
* This should ideally increase the performance as registers are being reused and but due to the increase in computations per thread execution drastically affects the performance.
* Hence, we can conclude that performance tuning based on registers creates more overhead if the calculations are not loop dependent. It is unnecessary and gives poor performance.

1. **Steps to run this assignment:**
2. nvcc lab3.cu -lcublas -o lab3.out
3. ./ lab3.out
4. In case you want to check out the result matrix values, please uncomment the blocks printing vector C.

1. **References**:

<https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/>

<http://docs.nvidia.com/cuda/cublas/#cublas-lt-t-gt-gemm>

<http://stackoverflow.com/questions/9187899/cuda-shared-memory-array-variable>