INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

ES 611: Algorithms on Advanced Computer Architectures

Introduction to Compute Unified Device Architecture (CUDA)
Report on Computational Lab Project 3.0
by
Shashank Heda
March 19th, 2013

ABSTRACT

As today in so many fields, such as finance and engineering, computation is the main part of the algorithm and takes too much time in execution of the algorithm, it is necessary to parallelize the computation or reduce execution time by optimization of resources.

GPUs are widely used in (HPC) High Performance Computing. To achieve speedup, either we can increase clock frequency or multiple computation cores on the same chip. The clock speeds have reached the physical limit, so the use of many cores is the only way left to achieve speedup. Heterogeneous systems composed by coupling commodity CPUs and GPUs turns out to be relatively cheap, high performing systems. As the GPU is growing demand of the Game Industry and large scientific computations, efforts have been made to take advantages to gain maximum utilization of the GPUs in computation. Recent development of GPGPU technologies gives even more powerful control over them.

The experimental outcomes of this project report confirm that the application written with CUDA leverages the GPU's huge number of cores, to solve with an appreciable preciseness, big problems with many variables and constraints quite faster than the serial version.

1. Parallel Matrix-Matrix Multiplication on GPUs

Implementation of sequential matrix-vector multiplication and matrix-matrix vector multiplication on CPUs and on GPUs of HPCLab using CUDA and checking the performance

Method of approach

To carry out matrix-matrix multiplication, after accepting the arguments of the matrix from command line itself (in putty or ubuntu terminal), we pass the dimensions of the matrix to a function which allocates the required amount of memory to both, the host and the device. In case the memory allocation isn't possible due to limited resources (limited memory), the program would exit with an error (on standard output). After memory allocation, all the threads of the device are synchronized and a timer is started.

We must remember that the timings must be calculated only for the computation part and not for the allocation part as we are making a comparison only between computational time taken by CPUs and GPUs. Each row and column of the matrix computes is assigned to a particular thread of the device (GPU). Finally after computation, all the threads are synchronised and the results are written to the host.

Results and Discussion

Output of the Program that multiplies the given matrix and the vector:

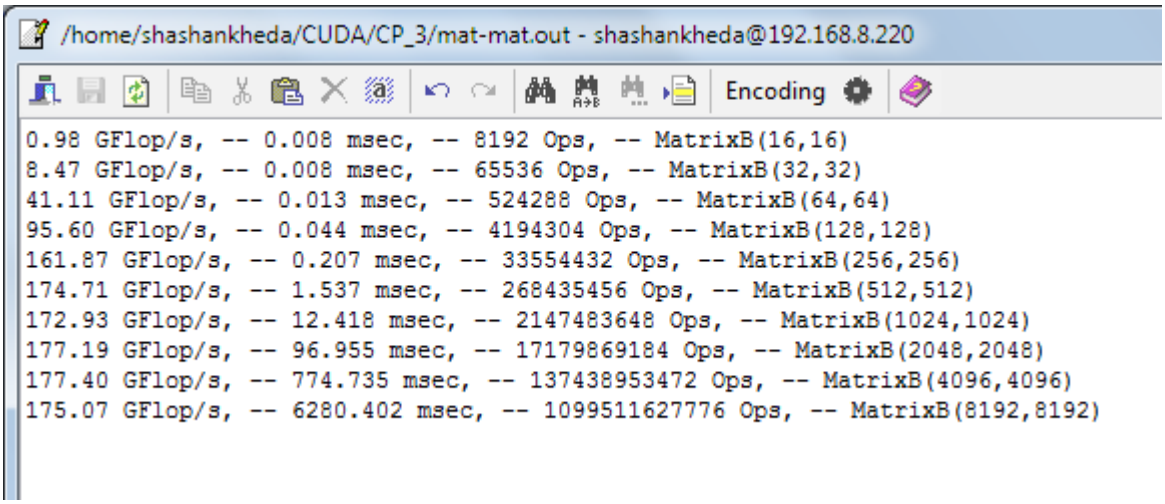

Figure 1.1 Output of the Program for matrix-vector multiplication

Table 1.1: Tabular Data for variation in timing vs n/p using CUDA:

n/p	Timings (seconds)
16	8.00E-06
32	8.00E-06
64	1.30E-05
128	4.40E-05
256	2.07E-04
512	1.54E-03
1024	1.24E-02
2048	9.70E-02
4096	7.75E-01
8192	6.28E+00

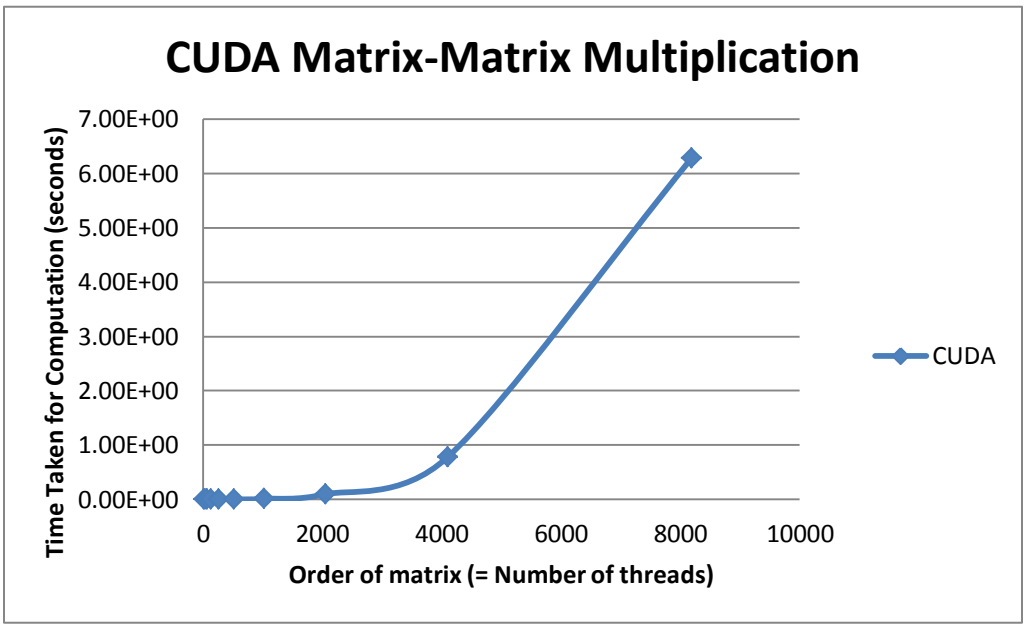

Figure 1.2 Variation in time with change in n/p

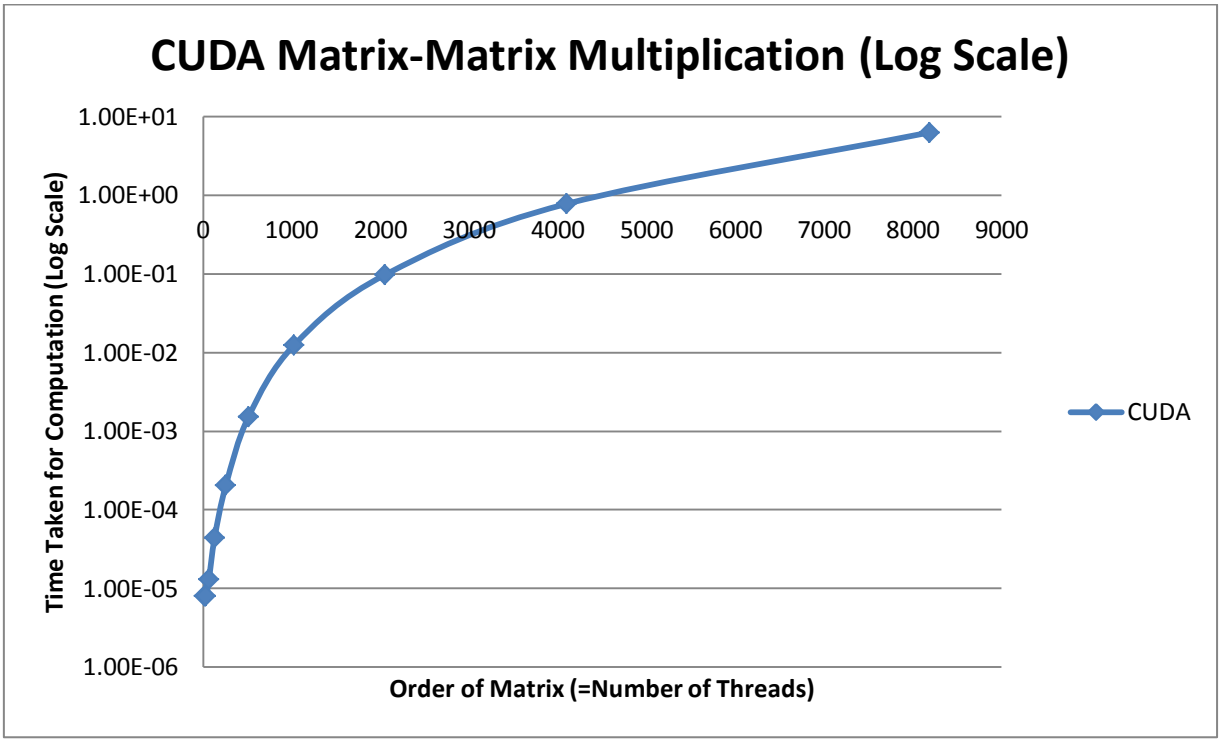

Figure 1.3 Variation in time with change in n/p (log scale)
(The log scale graph shows a better view of the variations taking place for smaller order matrices

Now from the above results shown for n=16, 32, 64, 128, 256, 512, 1024 for varying number of threads, we may conclude following few aspects of matrix-matrix parallel multiplication, when executed using CUDA:

- It is verified that the matrix-vector multiplication parallel program written using OpenMPs works perfectly. When given code is executed for a matrix of order 16 multiplied to a 16*1 vector, with all their entries = 1, no matter what the number of threads is, the output is a 16*1 matrix with all entries = 16, which is true. Hence the practical results coincide with the theoretical results.

- Time taken for execution:
    - Theoretical Prediction: Time taken for calculation of matrix-vector product would increase with increase in order of matrices being multiplied.
    - Practical Results:
    - As we increase the order of matrix, it is found that the time taken for computation increases exponentially.
- Reasons for Increase in time with increase in number of threads:
- Also, we need not bother if the n/p ratio is an integer or not, which gives CUDA architecture an edge over MPI or pthreads. CUDA provides an easy method to create threads. In our code of matrix-matrix multiplication, each thread calculates only 1 value $\sum (a_{ik} * b_{kj})$ for a particular row of A and a particular column of B. Hence number of threads increases with order of matrices.

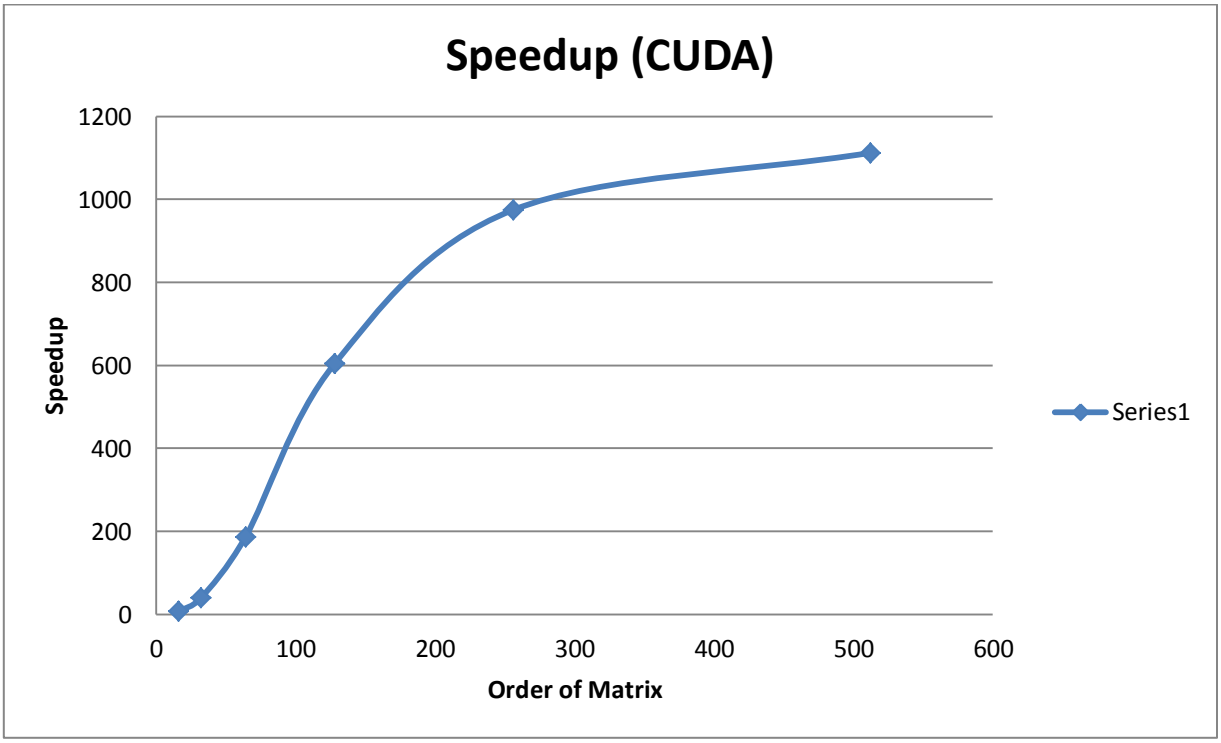

Figure 1.4 Variation in speedup with change in order of matrix

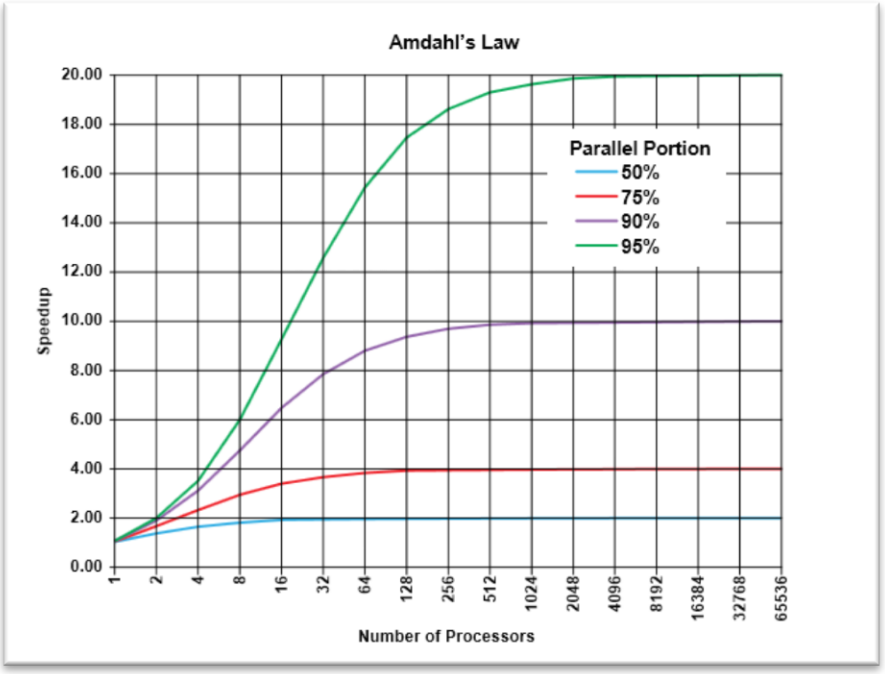

Figure 1.5 Variation in speedup with change in number of threads / order of matrix (Ideal)

- Speedup is found to increase with increase in number of matrices and then tends to saturate as the order of the matrices involved in multiplication are increased. Plotting shows speedup values only up to n=512 since for n > 512, the serial method depicts problems in memory allocation and thus, would depict errors during allocation.

- The plot for speedup matches the Amdahl's Law plot (shown above) as it tends to reach the values obtained by Amdahl's Law for 95% Parallel Portion (in a code).

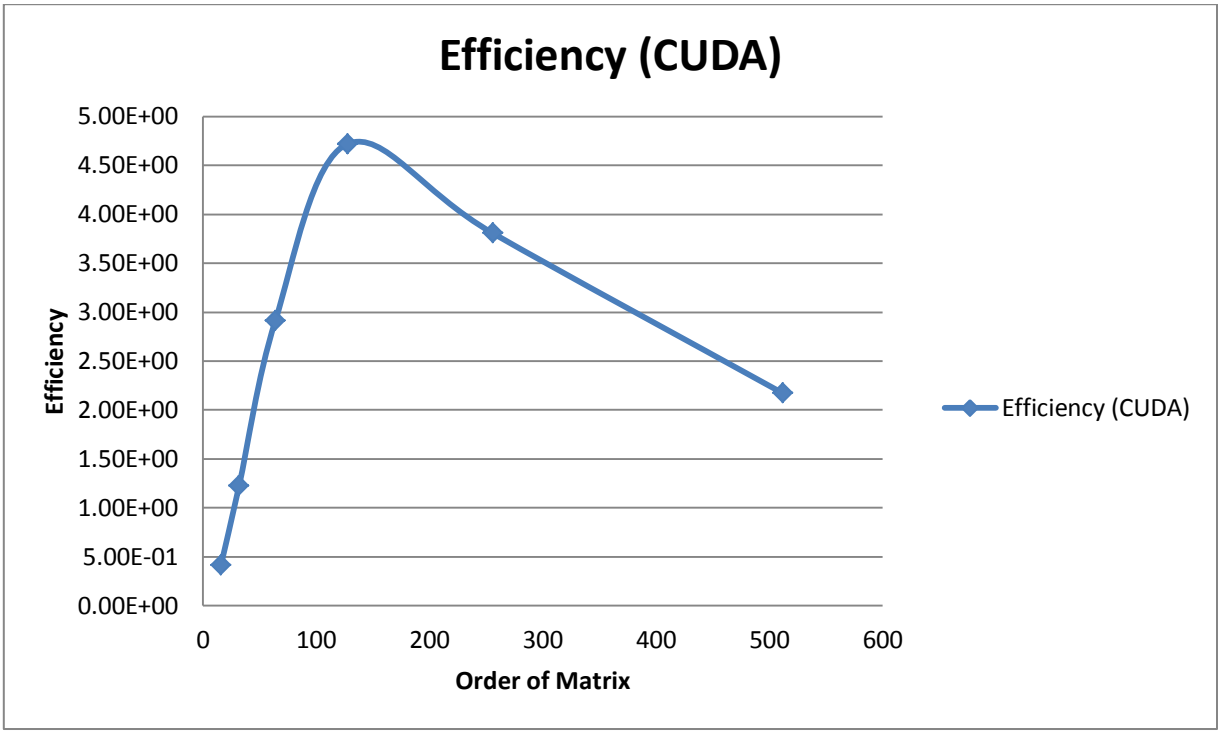

Figure 1.6 Variation in efficiency with change in number of threads / order of matrix

- The plot for efficiency shows a peak for an order of about 150. As our basic coding principle for matrix-matrix multiplication was number of threads = order of matrix, it is found that maximum utilization is achieved for order of matrix ~ 150. It means that for n > 150, it would be good if number of threads is not increased with the order, i.e., increase in number of threads must not be equal to increase in order of matrix.

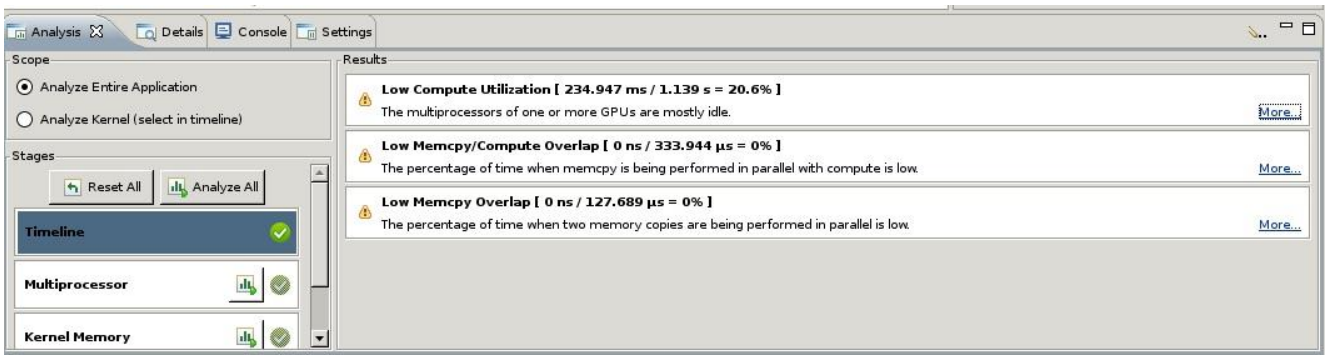

Figure 1.7 Warning when GPUs are not properly and fully utilized

- The plot below (Fig. 1.7) shows comparison between computational times for matrix-matrix multiplication using various techniques on a logarithmic scale. We may see the conclusion of all the projects we did so far.
    - Pthreads takes more time than serial code for lower order of matrices.
    - MPI also takes more time than serial code for lower order matrices.
    - OpenMP takes more time than serial code initially but overtakes when the order of matrices is around 200.

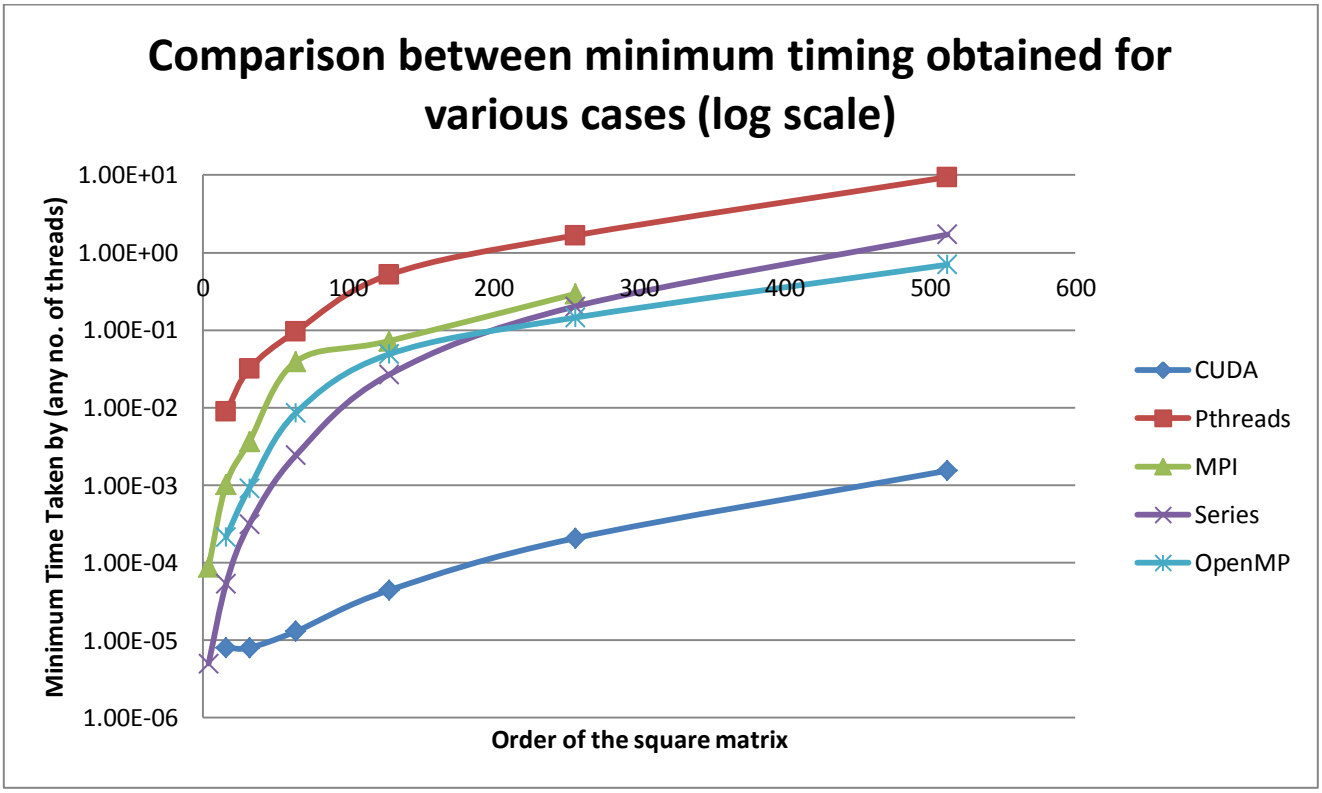

Figure 1.8 Variation in computation time with change in number of threads / order of matrix

- Though it's not advisable to make a comparison between different execution techniques based on CPU and GPU (CUDA), but to have a rough estimate of the overall speedup achieved, timings for CUDA are also plotted with timings for the CPU based parallel programming techniques. The comparison should be used only to understand the ideology behing GPU computing. It is because plots for Pthreads, OpenMP and MPI are based on number of processors/threads = 8, while number of threads for CUDA are actually increasing with increase in order of matrices.
- Figure 1.9 below shows the nvvp compiler for the matrix multiplication. It may be seen that the maximum time goes in allocation of memory to the device, after which comes computation, and finally the time taken to free the memory and reset the device.
- The time required for copying depends on size of data being copied from Host to Device and then back from Device to Host. The Figure below shows a small streak of line (as the matrix file size is 600kb. The time was found to increase significantly for matrix file size of 2MB). This timing increases if the size of matrix increases.

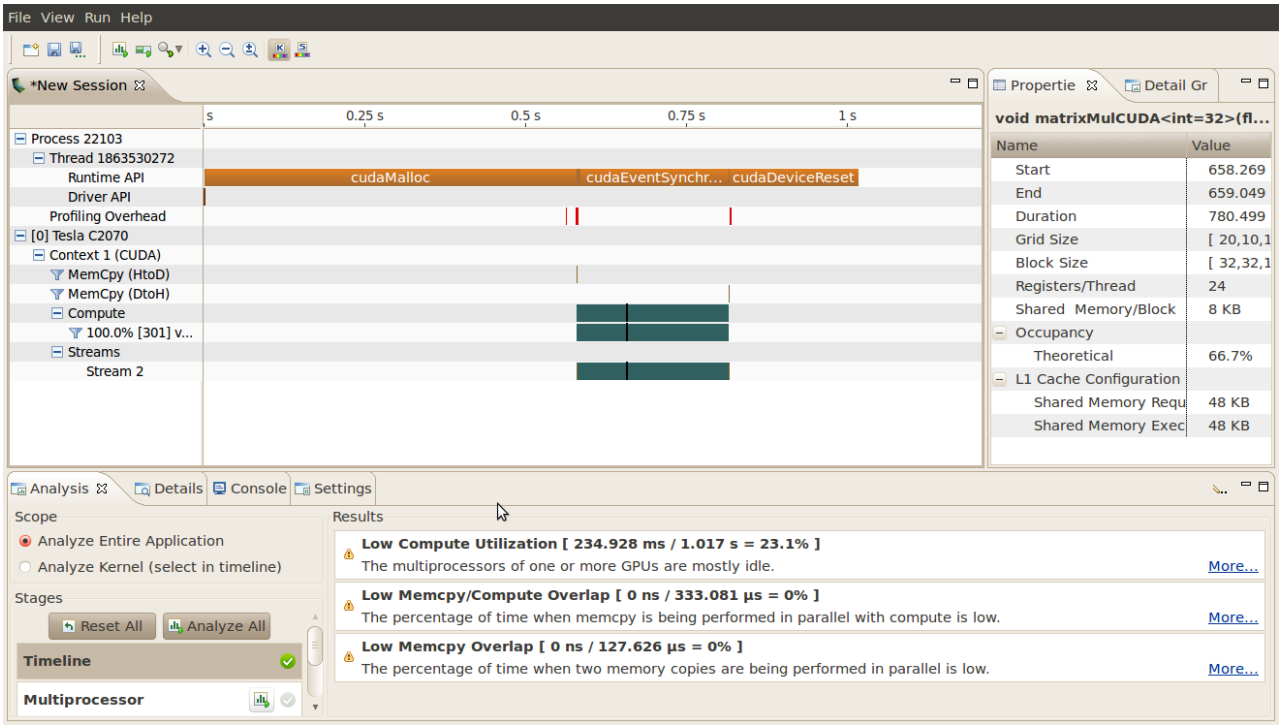

Figure 1.7 nvvp profiler for matrix-matrix multiplication

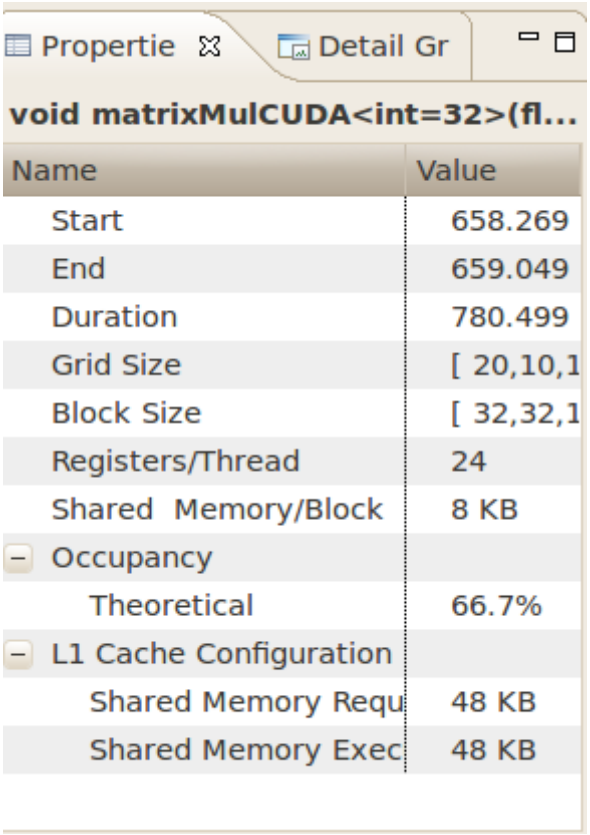

Figure 1.7 Device Statistics from nvvp profiler for matrix-matrix multiplication

2. Calculation of pi using infinite series approximation

Method of Approach

This problem involving calculation of pi using Monte Carlo method can be easily performed using CUDA. Initially, after defining the dimensions of the grid and the block and allocating memory to the device, a separate function is called to perform calculation on the device which calculates number of darts which fall inside the circle of radius 1. After getting each thread to have calculated the required value, another function is called on the device to reduce the pi values obtained from different threads. The results from the device are then retrieved and stored in a host array using which the value of pi is calculated. After accepting the number of threads from the user, a loop is run in parallel on all the threads which multiply each fraction of the individual series by factor 1 or -1 depending on the position of the fraction in the infinite series and calculate local sums. Thus, the value of sum obtained from various threads is reduced to get the final value of sum which is then multiplied by 4 to get the value of pi.

Results and Discussion

Table 1 showing variation in time with change in number of threads/blocks

No. of Threads\Blocks	256	128	64	32	16	8	4
256	6.25	1.2	1.05	1.09	1.09	0.93	0.96
128	1.87	1.24	0.96	0.99	0.91	0.96	0.92
64	1.03	1.27	0.89	1.48	0.95	1.05	1.04
32	1.08	1.34	1.3	0.95	0.98	1.1	1
16	1.04	0.93	0.89	1.15	1.78	1.07	
8	1.02	0.95	0.92	0.88	1.1	0.99	
4	1.83	0.92	0.87	1.25	1.38	0.94	

Table 2 showing variation in time with change in number of threads for a particular value of total threads

Threads in one block	Total Th = 16384	Total Th = 8192	Total Th = 4096	Total Th = 2048	Total Th = 1024	Total Th = 512	Total Th = 256	Total Th = 128
256	1.05	1.09	1.09	0.93	0.96			
128	1.24	0.96	0.99	0.91	0.96	0.92		
64	1.03	1.27	0.89	1.48	0.95	1.05	1.04	
32		1.08	1.34	1.3	0.95	0.98	1.1	1
16			1.04	0.93	0.89	1.15	1.78	1.07
8				1.02	0.95	0.92	0.88	1.1
4					1.83	0.92	0.87	1.25

- Table 1 above show time taken in calculation of value of pi for different block_size in a grid, each with different number of threads.
- Table 2 above shows time taken in calculation of values of pi keeping the value of total number of threads = CONSTANT. The variables in this case are: Num_threads in one block and Num_block.

NUM_BLOCK * NUM_THREADS = TOTAL THREADS

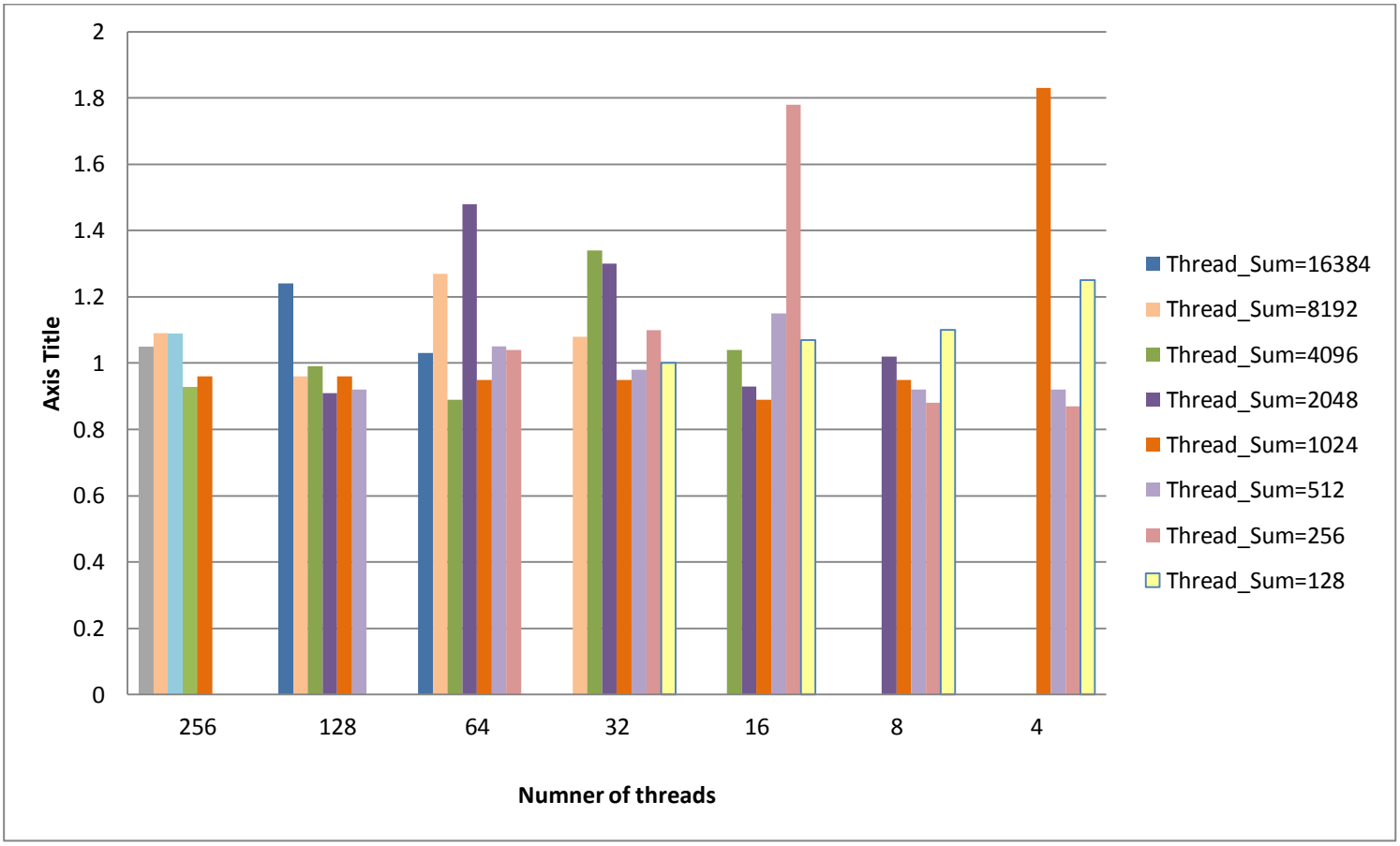

Figure 2.1 Variation in time taken for a particular number of total threads with change in number of threads/blocks

- Figure 2.1 shows a plot of time taken for computation for a particular number of total threads. The number of blocks and number of threads in each block are varying such that their product is constant. The computation time is found to be lowest in case when number of threads = 4 for num_blocks = 64. (Total threads = 256)
- The computation time is found to be maximum when number of threads =4 for num_blocks =256. (Total threads = 1024)

Table 3 showing minimum/maximum computation time taken for change in number of threads/blocks

Red indicates – maximum computation time for a particular number of threads

Green indicates – minimum computation time for a particular number of threads

No. of Threads\Blocks	256	128	64	32	16	8	4
256	6.25	1.2	1.05	1.09	1.09	0.93	0.96
128	1.87	1.24	0.96	0.99	0.91	0.96	0.92
64	1.03	1.27	0.89	1.48	0.95	1.05	1.04
32	1.08	1.34	1.3	0.95	0.98	1.1	1
16	1.04	0.93	0.89	1.15	1.78	1.07	
8	1.02	0.95	0.92	0.88	1.1	0.99	
4	1.83	0.92	0.87	1.25	1.38	0.94	

- From table 1, it is clear that the variation is time differs for different block sizes and different number of threads.
    - For 256 threads/block, it is seen that the computation time decreases with decrease in number of blocks, which depicts that the total number of threads being used for a particular problem must be optimised as in this case, using more number of threads (Total threads) doesn't prove to be fruitful.
    - From table 3, it is evident that it is always a good choice to keep
        - Total number of threads must be optimised based on the length of computation in a program.
        - For a particular value of threads, Number of threads/block ~ Number of blocks for intermediate values of num_threads and num_blocks (num_threads * num_blocks = Total threads) in order to have minimum time as depicted in Table 3 above.
        - The number of threads/block must be kept at an optimum level. Lesser number of threads/block => incomplete utilisation of GPU. A very high number of threads/block => most threads go idle as the load/thread decreases.
        - The number of blocks within a grid must be minimised in order that the communication within GPU is faster.
        - The number of Grids must as well be minimised to decrease the time spent in inter grid communication inside a GPU.

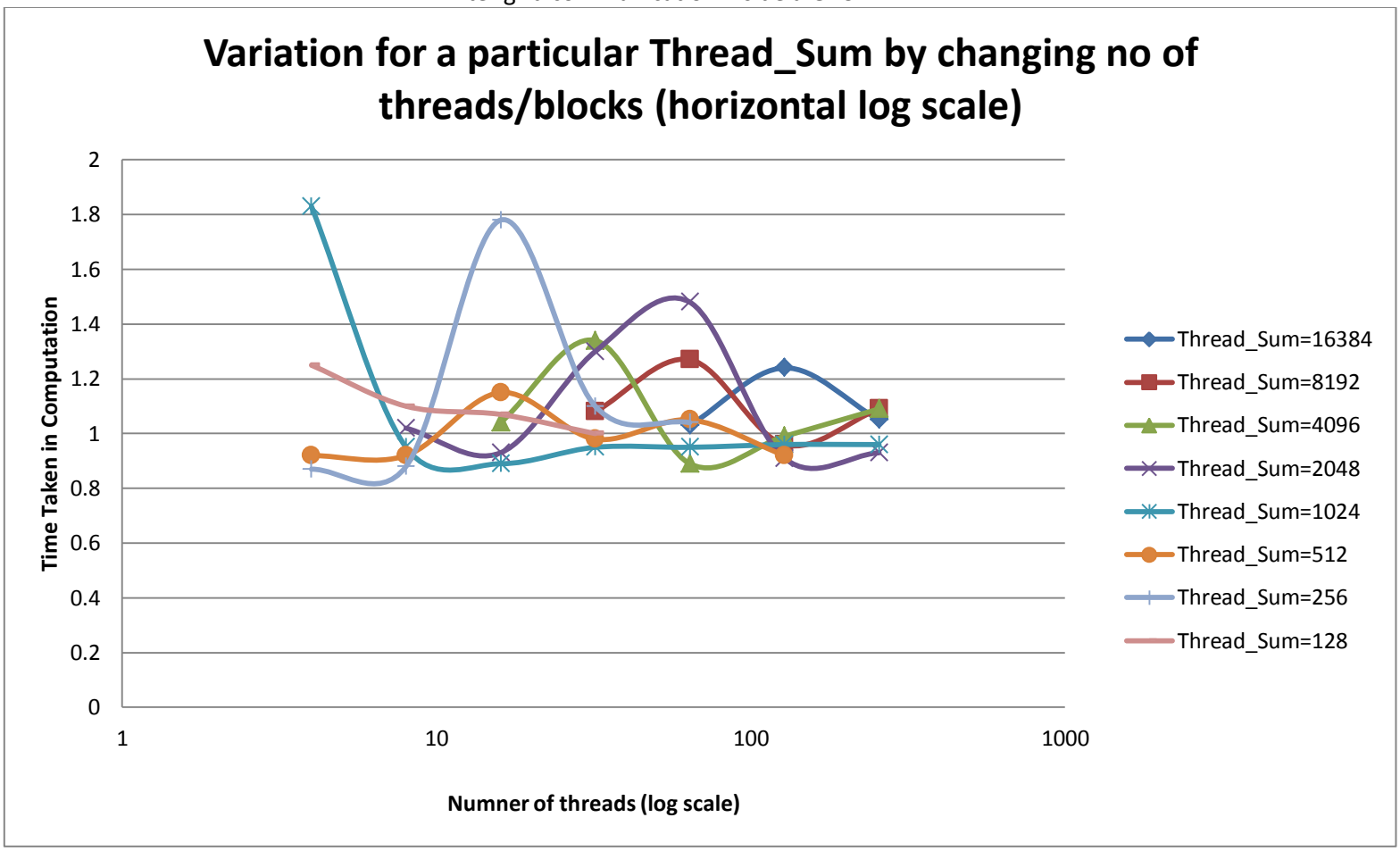

Figure 2.2 Variation in time taken for a particular number of total threads with change in number of threads/blocks (horizontal log scale)

3. Playing Ping-Pong on GPUs on Node 1 and Node 2

- It was noticed that though an algorithm may be written for Ping-Pong on GPUs on Node1 and Node 2 but the timings and the throughput (from nvvp compiler) indicate that is is not at all meaningful to do so.
- It is because for different grids performing a task on a GPU, the internal communication speed in terms of throughput is of the order of 144 gbps (Fig 3.1).

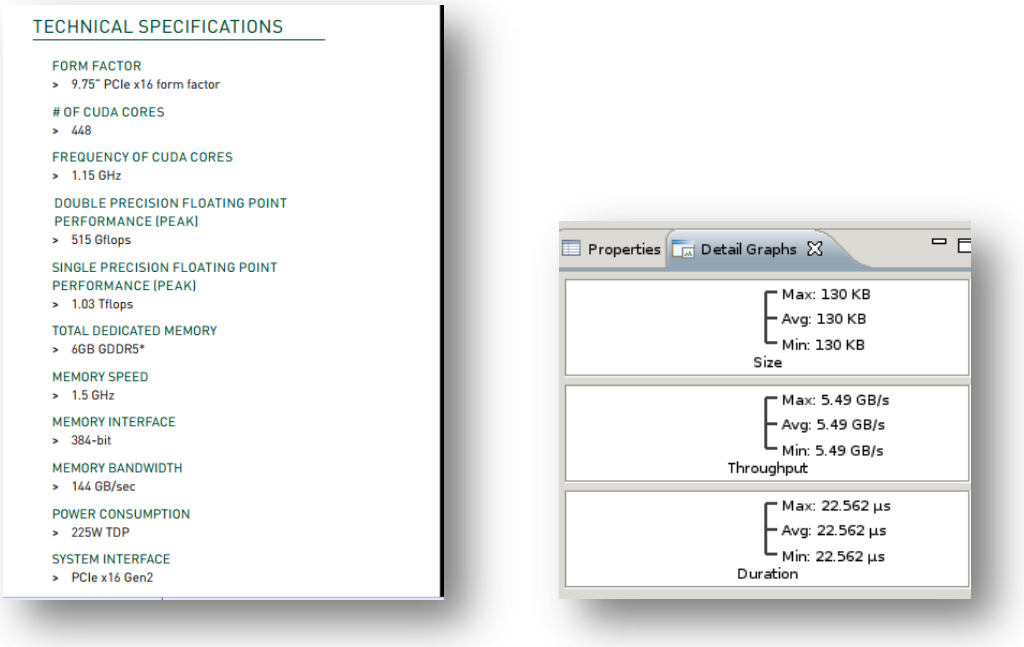

Fig 3.1 Technical Specification, GPU and througput (Host -> Device)

- But to play ping-ping between two GPUs on Node1 and Node2 (we have 1 GPU on each node), we need to have the following mechanism to control the inter GPU message passing.
  DEVICE1 (GPU, NODE1) -> HOST 1 (CPU, NODE1) -> HOST 2 (CPU, NODE2) -> DEVICE 2 (GPU, NODE2)
- This is because the two GPUs being on two different nodes need to be controlled by two hosts on each node. Now, the communication speed in terms of throughput is ~ 5.5 gbps between host and device (CPU and GPU). The communication speed between two hosts (node 1 CPU and node 2 CPU) is limited by bandwidth of ethernet cable of bandwidth 1gigabit ps = 125 mbps.
- Thus the overall high execution speed comes down from 144gbps to a minimal 125mbps which makes the use of 2 GPUs meaningless.
- The usage of 2 GPUs is meaningful only when both GPUs work on different dataset without interfering with each other. There must be no communication between them. Communication is feasible in case both GPUs are located on same node, in which case, the overall execution speed will be governed by host -> device speed (~5.5 gbps).

Acknowledgement

I have taken efforts in this project. However, it would not have been possible without the kind support and help I received from Prof. Murali Damodaran. I would like to extend my sincere thanks to him. I am highly indebted to his guidance and constant supervision as well as for providing necessary information regarding the project & also for his support in completing the project.
My thanks and appreciations also go to my friends Ravi, Parth, Nishant, Himanshu, Rachit and Vinay in verifying the correctness of project codes which we developed individually and people who have willingly helped me out with their abilities.