Introduction to GPU Programming Report

Sumanth Tangirala
Jalansh Munshi
Dhruvil Shah

Dhirubhai Ambani Institute of Information and Communication Technology

1 Context

1(a) Brief and clear description about the Problem

The two problems in question are the vector addition and square of a vector problems. The vector addition problem aims to give the vector, which results from the element-wise addition of two input vectors, as output. The square of a vector problem aims to give the vector, whose elements are the squares of the corresponding elements of the input vector, as output.

In the serial code, iterations over all elements of the vectors in a serialised manner is required in both the problems. There is no dependency between the elements of different indices and therefore this code can be run in parallel by allocating indices to threads which run the instructions on the elements in parallel.

Since the serial code iterates over all indices in a serial manner, spatial locality comes into play and therefore the caches play a major role in throughput. In the case of the vector addition problem, temporal locality is not present since a single element is accessed only once but in the vector square problem, the same element is accessed twice so temporal locality ensures that the second time the element is accessed, time is not lost for the memory access.

Problem	Number of Computations	Number of global accesses	CGMA
Vector Addition	1	2	0.5
Vector Square	1	1	1

In the code for the GPU, we mainly executed the code with each thread working on a single element of the vector. Therefore the number of threads were equal to the size of the vector. We mainly executed the code with 32 threads per block and the number of blocks suitably varied with the size of the vector.

In the code it can be seen that in the kernel, the thread does vector addition or square of the element only when its *tid* is less than the size of the vector. In all other cases the thread enters the kernel and exits it without executing anything.

The code divergence, in the case of having 32 threads per block, doesn't exist since the size of the vector is always a multiple of 32 and therefore there are no threads whose *tids* exceed the size of the vector. But in the case of, let's say, 10 threads per block, the code divergence

1(b) Complexity and Analysis Related

Complexity of serial code: In both the problems, the complexity of the serial code is proportional to the total number of operations for every size of the array which the product of runs and the size of the problem. In our case, the product of runs and size remains constant (2^{26}).

$$Complexity\ of\ serial\ code = O(RUNS * size)$$

Complexity of parallel code: The time complexity in this case is of the order of the product of RUNS and the time complexity required to run the kernel. Since the number of threads is equal to the size of the vector, if each thread handles a single index, then the time complexity required to run the kernel can be considered to be of constant time. Therefore the total time complexity would be of the order of runs.

$$Complexity\ of\ parallel\ code = O(RUNS)$$

Theoretical Speedup: Speedup is the ratio of time taken by serial code and the time taken by the parallel code. The complexity of the serial code is proportional to the total number of iterations. Whereas, the complexity of parallel code is proportional to the ratio of total iterations and the problem size.

Hence, according to the above mentioned theories, the speedup is ideally proportional to the problem size.

$$Theoretical\ Speedup\ \alpha\ size$$

1(c) Profiling Information

Vector Addition						
% time	**Cumulative seconds**	**Self seconds**	**Calls**	**Self ns/call**	**Total ns/call**	**Name**
59.49	36.61	36.61	2147483648	17.05	17.05	vectorAddition
37.33	59.57	22.97	-	-	-	main
3.91	61.98	2.41	-	-	-	frame_dummy

Vector Square (Each sample counts as 0.01 seconds.)						
% time	**Cumulative seconds**	**Self seconds**	**Calls**	**Self ns/call**	**Total ns/call**	**Name**
72.81	88.38	88.38	2147483648	41.16	41.16	vectorSquare
27.5	121.76	33.38	-	-	-	main
0.48	122.35	0.58	-	-	-	frame_dummy

1(d) Optimisation strategy

Since there is no dependency between one element and another element of the vector, we allocate one thread to one element of the vector. This way the GPU schedules the threads to ensure parallelism and due to the lack of dependencies, the GPU is free to schedule them in any order.

1(e) Problems faced during parallelization

Since the strategy implemented was straight-forward there were no issues faced in terms of parallelization.

2 Hardware Details

2(a) GPU Model

There are 2 CUDA devices.

CUDA Device #0
Name: Tesla K40c
Shared Memory Per Block: 49152
Warp size: 32
maxThreadPerBlock: 1024
clockRate: 745000
maxThreadsDim: -1055684204
maxGridSize: 1024
SM count: 15
concurrentKernels: 1
major: 3 minor:5

CUDA Device #1
Name: GeForce GTX 680
Shared Memory Per Block: 49152
warp size: 32
maxThreadPerBlock: 1024
clockRate: 1058500
maxThreadsDim: -1055684204
maxGridSize: 1024
SM count: 8
concurrentKernels: 1
major: 3 minor:0

2(b) CPU Model

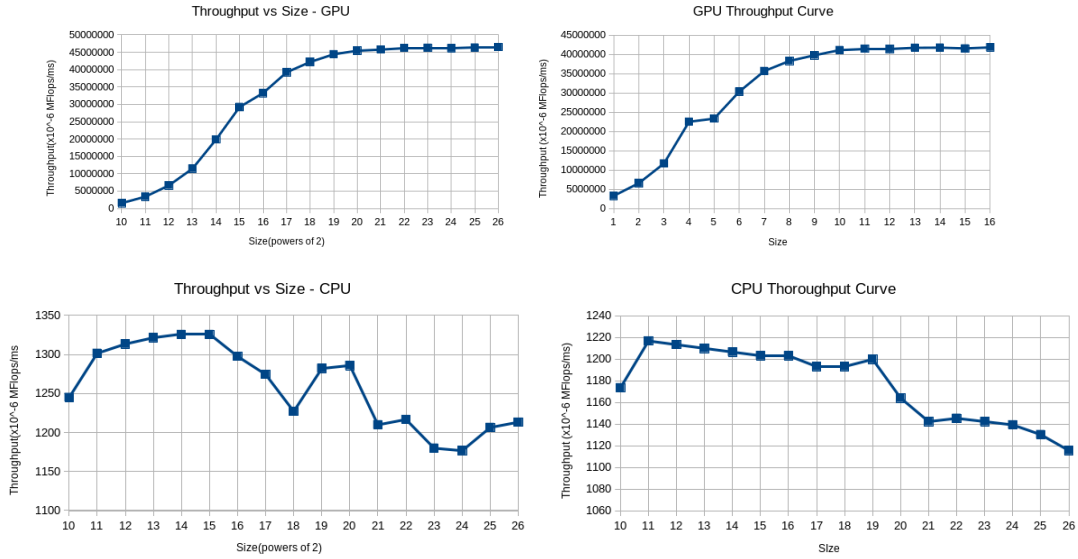
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian
CPU(s): 16
On-line CPU(s) list: 0-15
Thread(s) per core: 1
Core(s) per socket: 8
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 62
Model name: Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
Stepping: 4
CPU MHz: 1200.195
CPU max MHz: 2500.0000
CPU min MHz: 1200.0000
BogoMIPS: 3999.68
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K
NUMA node0 CPU(s): 0-7
NUMA node1 CPU(s): 8-15

3 Curve Based Analysis

3(a) Time Curve related analysis (as problem size increases, also for serial)

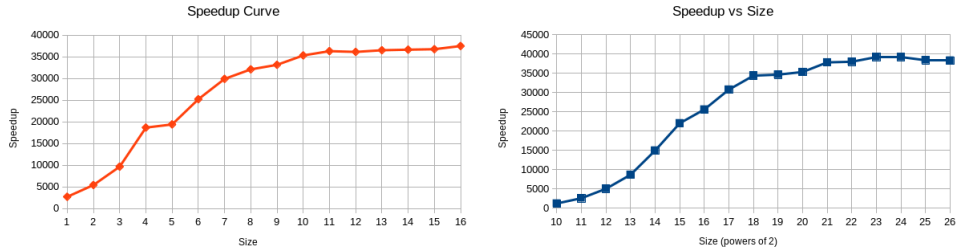
The curves for Vector Addition and Square are as under. The left half depicts the curves for Vector Square performance on GPU and CPU and the right half depicts the same for Vector Addition.



It can be observed that the throughput curves of the CPU follow the pattern expected out of the a CPU due to the capacities of the caches. In the case of the GPU, it can be seen that the throughput increases as size increases, however there it saturates at high sizes.

3(b) Speedup Curve related analysis

The curve on the left half shows the speedup analysis of Vector Addition and the right half shows the speedup analysis of Vector Square in the case of 32 threads per block.



It was also found that the performance remained nearly the same with multiples of 32. However, the speedup was very low in the case when the number of threads per block was not a power of 2.

4 Comparison Between Serial and Parallel Approaches

From the curves that were experimentally obtained, it is apparent that the parallel code performs better as compared to serial code, when the problem size

increases. The reason behind this might be the fact that the GPU is able to perform the tasks in parallel and it is able to do it even more efficiently as the problem size increases. On the other hand, it is not the case with CPU because the tasks are performed one-by-one in CPU and as the problem size increases, the time taken to complete the size also increases.