QuickSort Implimentation With CUDA

Aman Kumar 201911018
Harsh vardhan 201911022

1 Context

1.1 Brief description of the problem

Implementing the QuickSort Algorithm which is one of the most famous and efficient sorting algorithms. The algorithm is majorly performed by the help of the recursion. It will take an array of numbers as an input and will return the sorted array of numbers as the result. The algorithm is mainly based on divide and conquer approach, the idea is to break the larger problem into smaller sub-problems and solve them individually. Quicksort carries out two important operations, Partition and Sorting. When we deal with large number of elements the performance of serial execution of QuickSort performs poorly with increase in number of elements hence a parallel approach plays a great role in reducing the time for larger arrays.

1.2 Brief description of the Serial Code

Serial algorithm of quicksort can be done with the help of recursion and iterative approach. We will focus on the iterative approach of the quicksort, every recursion can be executed by the help of the iteration and stack. Stack is used here for storing the indexes of the elements.In recursion we pass these indexes to our parition function, here we do not have the advantage of recursion,so we save these indexes on a stack in order to break them into parts. In every iteration we select an element and put it into its correct position and that element is known as pivot element. All the elements to the left of it are smaller then the pivot element and all the element to the right of the pivot are greater then the pivot element.

1.3 Complexity of the algorithm (serial)

The Complexity of QuickSort in Worst case is O(n^2) , Average case is O(nlog(n)) and Best case is O(nlog(n))

1.4 Mapping of threads for computation in parallelization

In parallel code we compute individual elements of an array with diffrent threads by mapping them to indices using following code:
int index = blockIdx.x*blockDim.x + threadIdx.x;

1.5 CGMA Ratio

The CGMA Ratio of vector Addition Problem is 10:13.

1.6 Theoretical Speedup

Speedup = p
p = Number of Processors

1.7 Profiling Information (gprof)

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.20	46.25	46.25	262146	0.00	657.30	partition
0.04	46.27	0.02	668597	0.00	495.51	swap
0.02	46.28	0.01	1	0.01	46.28	QuickSort
0.00	46.28	0.00	1	0.00	0.00	initialize

- % time - the percentage of the total running time of the program used by this function.
- cumulative seconds - a running sum of the number of seconds accounted for by this function and those listed above it.
- self seconds - the number of seconds accounted for by this 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 ms/call - the average number of milliseconds spent in this function per call, if this function is profiled, else blank.
- total ms/call - the average number of milliseconds spent in this function and its descendants 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) 2012-2019 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 byte(s) for 0.02% of 46.28 seconds

index	% time	self	children	called	name
		0.01	46.27	1/1	main [2]
[1]	100.0	0.01	46.27	1	QuickSort [1]
		46.25	0.02	262146/262146	partition [2]
					spontaneous
[2]	100.00	0.00	46.28		main [2]
		0.01	46.27	1/1	QuickSort [1]
		0.00	0.00	1/1	initialize [5]
		46.25	0.02	262146/262146	QuickSort [1]
[3]	100.00	46.25	0.02	262146	partition [3]
		0.02	0.00	668597/668597	swap [4]
		0.02	0.00	668597/668597	partition [3]
[4]	0.0	0.02	0.00	668597	swap [4]
		0.00	0.00	1/1	main [2]
[5]	0.0	0.00	0.00	1	initialize [5]

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.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called. This line lists:

- index - A unique number given to each element of the table. Index numbers are sorted numerically. The index number is printed next to every function name so it is easier to look up where the function is in the table.

- % time - This is the percentage of the 'total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.
- self - This is the total amount of time spent in this function.
- children - This is the total amount of time propagated into this function by its children.
- called - This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a '+' and the number of recursive calls.
- name - The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

- self - This is the amount of time that was propagated directly from the function into this parent.
- children - This is the amount of time that was propagated from the function's children into this parent.
- called - This is the number of times this parent called the function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.
- name - This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word '¡spontaneous¿' is printed in the 'name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

- self - This is the amount of time that was propagated directly from the child into the function.
- children - This is the amount of time that was propagated from the child's children to the function.
- called - This is the number of times the function called this child '/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the '/'.
- name - This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The '+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Copyright (C) 2012-2019 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.

Index by function name

[1] QuickSort [3] partition

[5] initialize [4] swap

1.8 Optimization Strategy

The serial code is implemented by the help of the one stack, hence the performance of this algorithm will fall with increase in input size. To optimize the strategy into parallel execution use of two stacks has been done in order to seperately an parallely perform the execution of partition. Both the left and right partition will each have a stack of their own so we can run these both partitions parallely to achieve better computational time. This approach increases the performance by huge factor and boosts the overall performance of the algorithm on the larger inputs.

1.9 Problems faced in parallelization and possible solutions

- Recursion Not Supported- One of the hindrance was that we cannot take the help of beautiful recursion concept due to the limitation of cuda architecture.

  Solution - To overcome the recursion we need iterative approach for the same algorithm.

- Designing an iterative algorithm - An iterative algorithm was required and then parallelizing the same.

  Solution - Serial iterative execution required one stack, whereas the parallel algorithm is implemented with the two stacks.

- Address Management with Stack- The indexes(address) of left and right partitions must be saved to further execute the algorithm.

  Solution - Two stacks LStack(left partition) and RStack(right partition) are used to preserve the partition indexes during each iteration.

- Thread Synchronization - During the execution if the threads are not synchronized properly then multiple threads can read/write the values at same time which leads to wrong result and data loss.

  Solution - syncthreads() is the method in cuda specifically to synchronize the threads and avoids wrong read/write operations.

2 Hardware Details

2.1 CPU

processor	15
vendor$_id$	GenuineIntel
cpu family	6
model	62
model name	Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
stepping	4
microcode	0x42c
cpu MHz	1202.026
cache size	20480 KB
physical id	1
siblings	8
core id	7
cpu cores	8
apicid	46
initial apicid	46
fpu	yes
fpu$_exception$	yes
cpuid level	13
wp	yes
flags	fpu vme de pse tsc msr pae mce cx8
	apic sep mtrr pge mca cmov pat pse36 clflush
	dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall
	nx pdpe1gb rdtscp lm constant$_tscarch_perfmon$
	pebs bts rep$_goodnoplxtopologynonstop_tsc$
	aperfmperf eagerfpu pni pclmulqdq dtes64
	monitor ds$_cplvmxsmxesttm2ssse3cx16$
	xtpr pdcm pcid dca sse4$_1sse4_2x2apicpopcnt$
	tsc$_deadline_timeraesxsaveavxf16crdrand$
	lahf$_lmepbtpr_shadowvnmiflexpriority$
	ept vpid fsgsbase smep erms xsaveopt ibpb
	ibrs stibp dtherm ida arat pln pts spec$_ctrlintel_stibp$
bogomips	4006.08
clflush size	64
cache$_alignment$	64
address sizes	46 bits physical, 48 bits virtual
power management	

2.2 GPU

Property Name	Value
CUDA Device 0	
Major revision number	5
Name	Tesla K40c
Total global memory	11996954624
Total shared memory per block	49152
Total registers per block	65536
Warp size	32
Maximum memory pitch	2147483647
Maximum threads per block	1024
Maximum dimension 0 of block	1024
Maximum dimension 1 of block	1024
Maximum dimension 2 of block	64
Maximum dimension 0 of grid	2147483647
Maximum dimension 1 of grid	65535
Maximum dimension 2 of grid	65535
Clock rate	745000
Total constant memory	65536
Texture alignment	512
Concurrent copy and execution	Yes
Number of multiprocessors	15
Kernel execution timeout	No
CUDA Device 1	
Major revision number	3
Minor revision number	0
Name	GeForce GTX 680
Total global memory	2097086464
Total shared memory per block	49152
Total registers per block	65536
Warp size	32
Maximum memory pitch	2147483647
Maximum threads per block	1024
Maximum dimension 0 of block	1024
Maximum dimension 1 of block	1024
Maximum dimension 2 of block	64
Maximum dimension 0 of grid	2147483647
Maximum dimension 1 of grid	65535
Maximum dimension 2 of grid	65535
Clock rate	1058500
Total constant memory	65536
Texture alignment	512
Concurrent copy and execution	Yes
Number of multiprocessors	8
Kernel execution timeout	No

3 Parameters

3.1 Input Parameters

Array Arr[N]. The Problem size N is increased from pow(2,10) to pow(2,19).

3.2 Output Parameters

The Final array Arr[N] is the same array in which the sorted array is saved.

3.3 Comparison of results from serial and parallel code

After sorting values in Arr[N], using both Serial and Parallel code, we found that there is a clear difference in the time taken for various problem sizes, graph of which will be shown below.

log(N)	Serial time(sec)	Parallel time(sec)
10	0.000295	0.002095
11	0.000839	0.004144
12	0.002943	0.007616
13	0.015218	0.009131
14	0.045432	0.016168
15	0.151952	0.034294
16	0.682315	0.072944
17	2.496395	0.126581
18	9.932307	0.233966
19	38.378501	0.454057

4 Observations

4.1 Serial Time vs Problem Size curve.

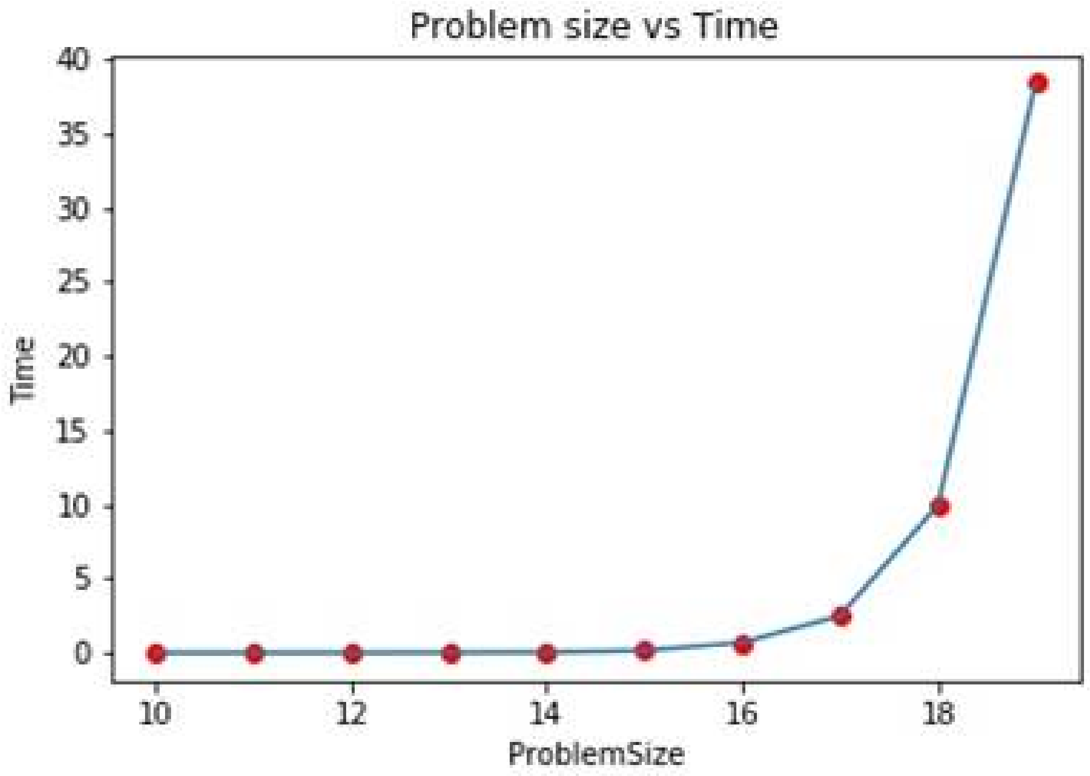

Fig. 1 Serial Time vs Problem size

In this graph at problem size N equals to pow(2,18) the cache and main memory are full resulting in a sudden increase in the time taken to compute such large problem.

4.2 Parallel Time vs Problem Size curve.

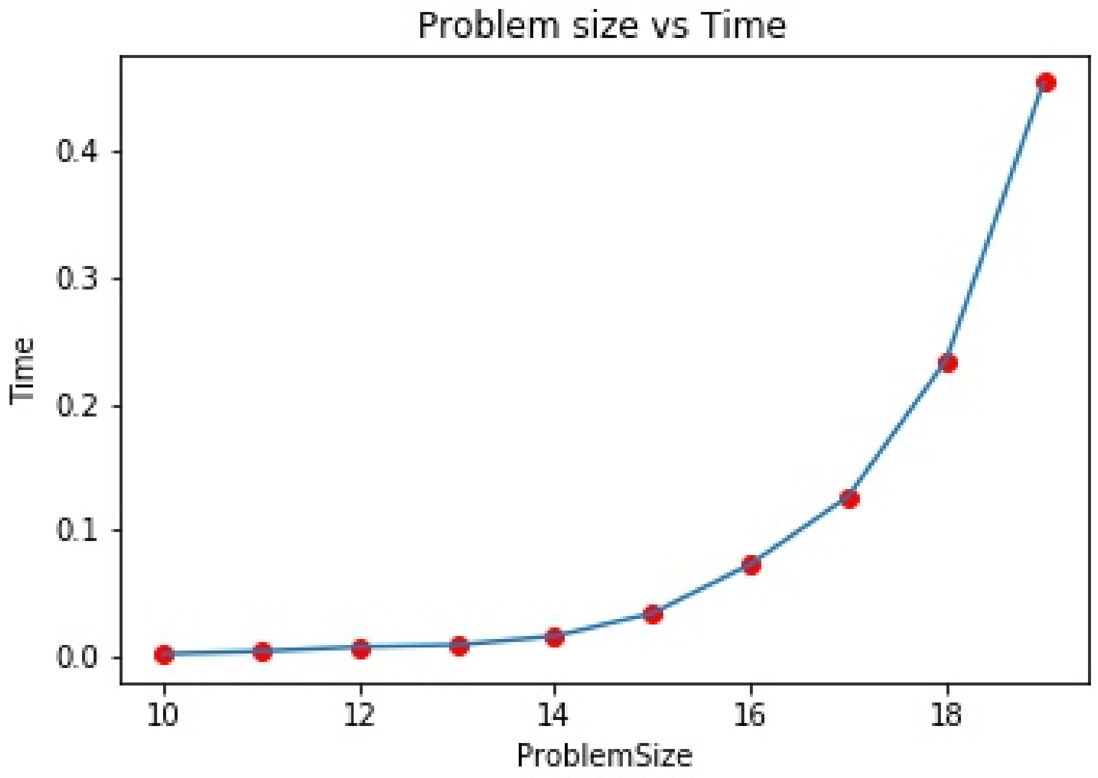

Fig. 2 Parallel Time vs Problem Size

In this graph the time taken by parallel code is increasing because the problem size is increasing exponentially.

4.3 Speedup Curve

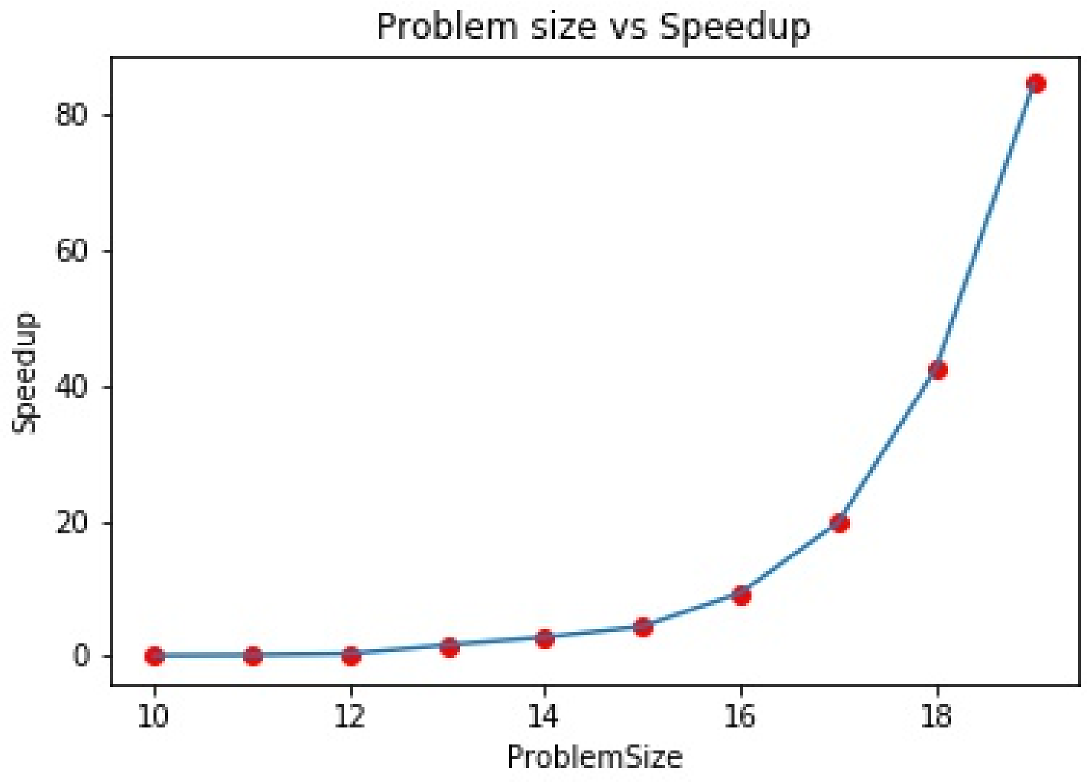

Fig. 1 Speedup vs Problem size

As Problem size increases parallelism increases thus resulting in increase in speedup.

4.4 Memory Input Time Curve

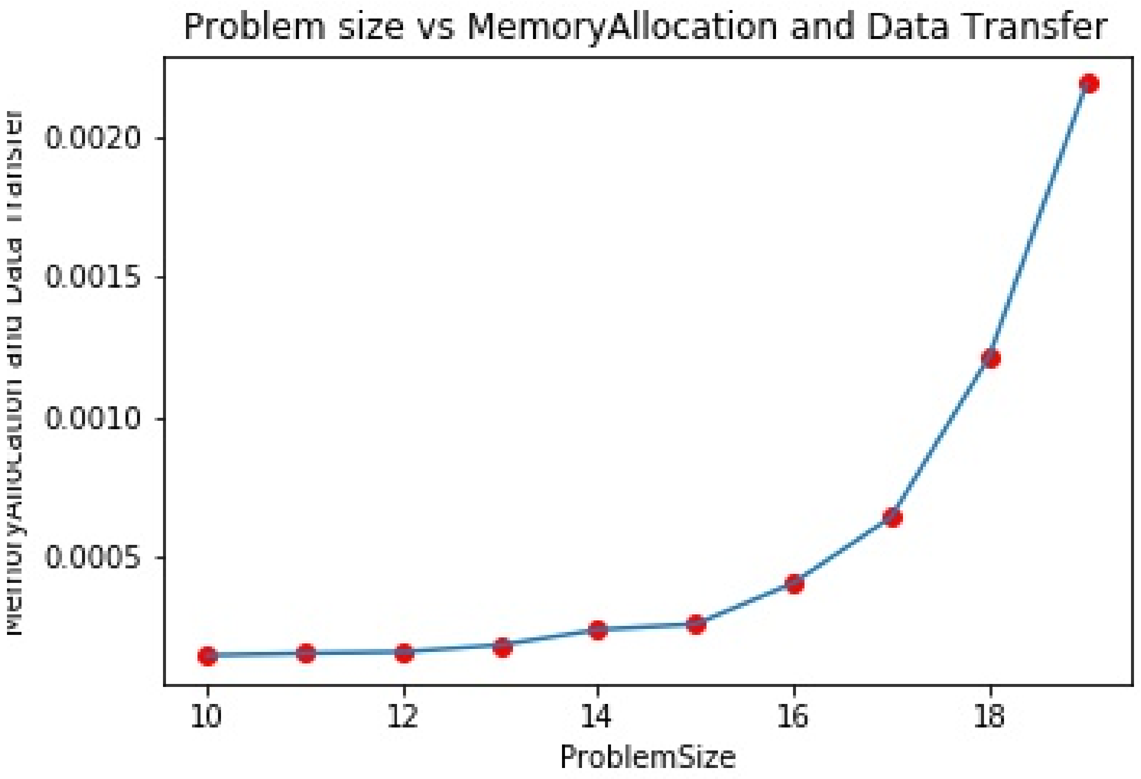

Fig. 1 Memory Input Time vs Problem size

As problem size increases the size of array increases thus increasing time used in allocating resources.

5 Measure performance in MFLOPS/sec

log(N)	Parallel time(sec)	Throughput(MFlops/sec)
10	0.002095	1954.974388
11	0.004144	1976.650856
12	0.007616	2151.278638
13	0.009131	3588.550387
14	0.016168	4053.407037
15	0.034294	4080.029380
16	0.072944	4096.784835
17	0.126581	4141.906756
18	0.233966	4281.747484
19	0.454057	4318.699756