Median Element Parallelized Search

Parallel Processing Systems module project
Medical imaging and Applications

Basel Alyafi, Fahad Khalid

Contents

List of Figures

List of Tables

1 Abstract

Median element search is a way to find the element that has a mid-range value in the array. Sequential and parallel solutions are available, however and unsurprisingly, parallel solutions are by far more efficient that sequential ones. In this project, a parallel implementation using Bitonic sorting was used. The merging part after sorting partial arrays was done sequentially. This implementation is automatically-scalable, that is, it can handle arrays of any applicable, fits inside the shared memory, array. An array of size 2^{20} was used to measure the performance on different sub-array sizes (256, 512, 1024, and 2048). Milliseconds were counted for both parallel and sequential part.

2 Problem Statement

The purpose of the project is to implement a median searching algorithm which executes the bitonic process in parallel instead of sequential manner, using CUDA 9.1. The language to use is C++. Comparison between different batch sizes are required to show the effect of shared memory size on the overall performance.

3 Introduction

Parallel computing is the type of computation which allows the calculation and computation or the execution of more then one process to be carried out simultaneously. This concurrent processing enables to break large problems into smaller ones which can then be solved on parallel. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has been associated with high-performance computing, it is subjected to a large amount of interest due to its physical constraints. As power consumption (and consequently, heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors. Parallel hardware has been ubiquitous for some time now. It is difficult to find a laptop, desktop, or server that does not use a multi-core processor. Beowulf clusters are nearly as common today as high-powered workstations were during the 1990s, and cloud computing could make distributed-memory systems as accessible as desktops.

Most people are familiar with serial computing even if they don't realize it, most programs written by programmers on a daily basis are sequential programs. Serial programs run on a single processor, instructions are run one by one, one after the other, i.e., in a series and only one is executed at a time. Figure 1 shows IBM's summit, which is capable of computing 200,000 trillion calculations per second and is the worlds fastest supercomputer in 2018. In order to achieve parallelism, a program must be split into multiple parts so that each processor can execute its part simultaneously with other processors. Parallel computing can be carried out on a single computer with many processors or separate computers connected together on the same network, or a combination of both. Parallel computing has been around for many years but it is only recently that interest has grown outside of the high-performance computing community. This is due to the introduction of multi-core and multi-processor computers at a reasonable price for the average consumer [7].

Figure 1: IBM summit supercomputer [6]

3.1 General Purpose Graphics Processing Unit (GPU)

GPU's were made and used mainly for processing 3D and high-definition images. However, and due to the desperate need for fastening the processing of other parallelizable types of data like computational finances and machine/deep learning training/testing datasets, GPU's were used by dividing the big task into smaller fine-grained tasks assigned to blocks of threads, then, each small task is divided again into smaller finer-grained mini-tasks assigned to individual threads which will run on parallel. Due to the fact that GPU's are equipped with higher ability to manage data processing than cashing and control bits flow, they have been utilized for the purpose mentioned previously [5].

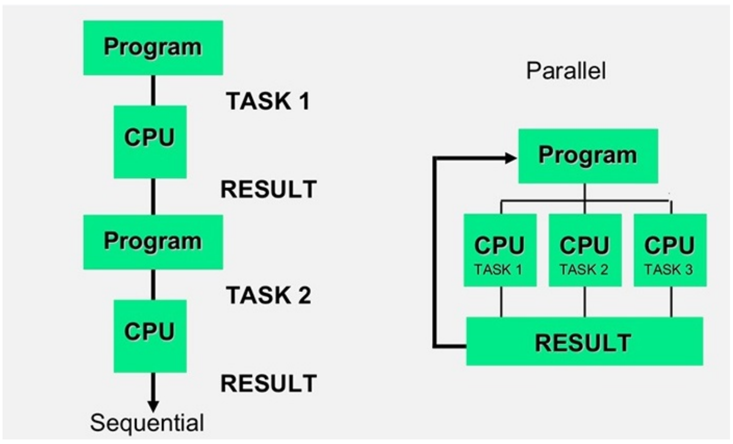


Figure 2: Sequential processing vs parallel processing [3]

The main advantage of parallel computing is to reduce the amount of time it takes to execute the whole process. Consider the block diagram in figure 2, it shows a comparison between sequential and parallel program being executed. The execution chain in sequential programming is carried out one after the other, first the task one followed by task two, where as in the parallel chain, the three tasks are carried out in parallel, which reduces the time consumed.

3.2 CUDA

CUDA is a general-purpose parallel computing platform and application programming interface (API) model powered by Nvidia corporation. It enables software engineers and software developers to make use of GPU (General Purpose computing) for general purpose computing. To make it easier for developers and programmers CUDA has been designed to work with C, C++, FORTRAN, and other high-level programming languages. When it was first introduced by Nvidia, the name CUDA was an acronym for Compute Unified Device Architecture, but Nvidia subsequently dropped the use of the acronym [4].

The CUDA programming model is a heterogeneous model which makes use of both the CPU and GPU. The CPU and its memory are the host where as the GPU and its memory are the device. Code will be executed on both the host and the device memory and also launches kernels which are like functions called by the host, yet, executed on the device. GPU threads work in parallel to execute these kernels.
Here is how a typical CUDA program is executed:

1. Declare and allocate host and device memory.
2. Initialize host data.

3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.

3.2.1 CUDA Model Hierarchy, a down-top view

- Threads: It is simply the execution of a kernel using the given index. The aim of the threads is to utilize its index to access each element in an array, the threads divide themselves in such a manner that collectively they are able to process the entire data set.
- Block of threads: A block is a group of threads. For the purpose of better processing and data mapping, threads are grouped into blocks, the number of threads may vary with the availability of shared memory. The number of threads in a block is limited differently on different machines, for instance, Tesla C1060 can handle 512 threads per block, while Tesla C2050 manages to handle 1024 threads per block only, [2]. A thread block can either be executed serially or in parallel. The __syncthreads() function gives control over threads in a block, it is used to stop a certain thread in a kernel until all the other threads in its block reach the same point.
- Grid of Blocks: This is a group of blocks. Each block located in a grid contains the same number of threads. A grid is an efficient means of computation which allows a number of thread blocks (8 - 16) to operate in parallel, see figure 3.

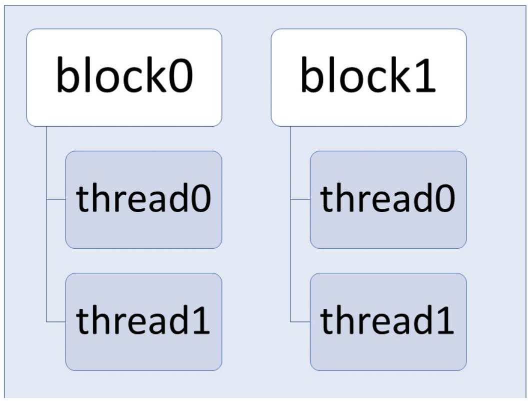


Figure 3: A CUDA grid consisting of two blocks, each has two threads

3.2.2 Kernels

A CUDA kernel is a C function that is, when called, executed N times by N different threads, [5]. The first thing to understand is the global which implies that this function can be called from the host PC or the CUDA device, but executed on device only. As it is known that threads within the kernel are responsible for different data elements, the threads are all running the same code so the only way to indicate which thread is responsible for which data element is by using their threadIdx and blockIdx. threadIdx is a inside-block unique index, which when combined with blockIdx multiplied by blockDim (block size), becomes a globally-unique index.

3.3 Bitonic Sorter

Before we go into the concept of a Bitonic sorter, lets understand bitonic sequence. A Bitonic Sequence is a sequence of numbers which is first strictly increasing then after a point strictly decreasing.

$$Array = [6, 7, 8, 11, 9, 5, 2, 1] \tag{1}$$

Equation (1) above represents an array of integer's starting from 6 moving up to 11 and after 11 we see a decrease in the sequence, hence we can say the sequence of numbers in the array is a bitonic sequence.

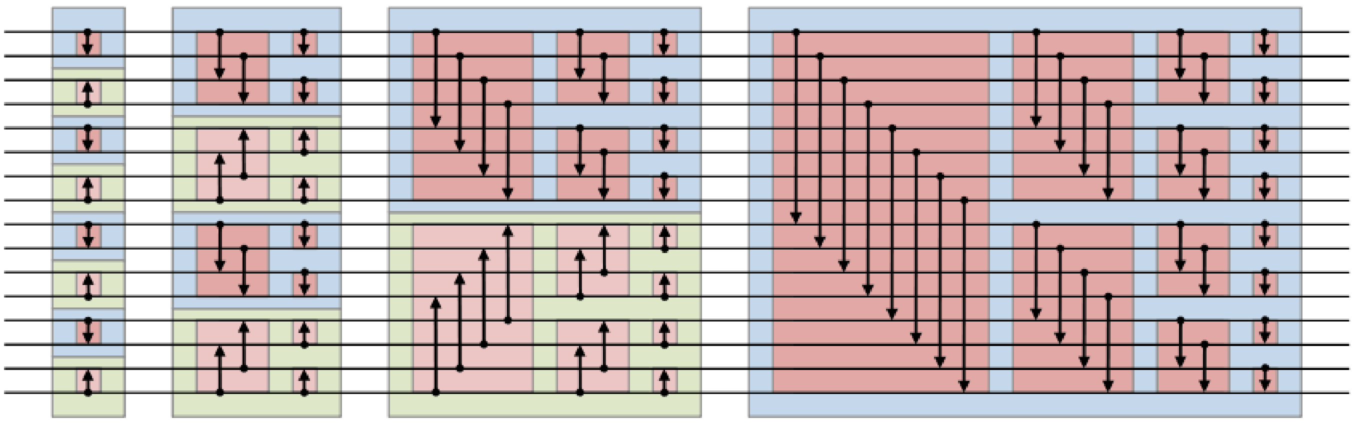


Figure 4: Bitonic Sorter

By considering the Butterfly structure, Figure 4, lines are the elements to be sorted. Each red box (light or dark) has the same structur and direction. The arrows act as comparison operators, two numbers at the two arrow ends are compared and swapped if necessary given the arrow points towards the larger number. The colour combination has no effect it is just for better visualization and have no effect on the algorithm. If the inputs happen to form a bitonic sequence then the output will form two bitonic sequences. The top half of the output will be bitonic, and the bottom half will be bitonic, with every element of the top half less than or equal to every element of the bottom half or vice versa. This will continue until the sequence is sorted, [1].

4 Methodology

First, the complete process is made on multiple phases (with index p), each with multiple sub-phases (with index q), Figure 5. Phases are executed sequentially, each on one block of threads. While different blocks are working simultaneously asynchronously. Threads with local index zeros (threadIdx = 0) responsible for updating the values of thread size (the size of the array this thread will sort), the number of threads per block, and sorting direction per thread. This is done just before each sub-phase. Threads inside each block are synchronized, so no thread starts with a new phase/sub-phase till all other threads have finished the previous one, this was done using barriers (synchthreads). By this way, alive data are protected from being damaged, additionally, outdated variables (thread size and threads per block) are not used. When the kernel is called, the default number of threads per block equals number of elements divided by

2, then this number is updated during the different stages. Finding the median element was done on two main steps: namely, sorting input array then picking the middle element.

4.1 Sorting elements

1. sort sub-arrays using parallel bitonic: in this stage, the input array is divided into smaller sub-arrays with predefined size, see Figure 5 where the black solid box represents one block, while the yellow dotted one represents a thread. The sub-array size should be compliant with the shared memory size of the machine under use. Then, each sub-array was handled by a set of threads, each one handling a mini-batch of the original input array (yellow boxes). For each thread, the swap function is explained in pseudo code, Algorithm 1. Figure 6 shows the mechanism of swapping in a four-element sub-array.

2. merge sorted sub-arrays: after finishing sorting the sub-arrays, it is time to merge sorted sub-arrays to form the completely-sorted array. Merging was done sequentially using the procedure described in pseudo code, Algorithm 2

4.2 Picking the element in the middle

After sorting the complete array, it is possible to pick up the element in the middle to be the median.

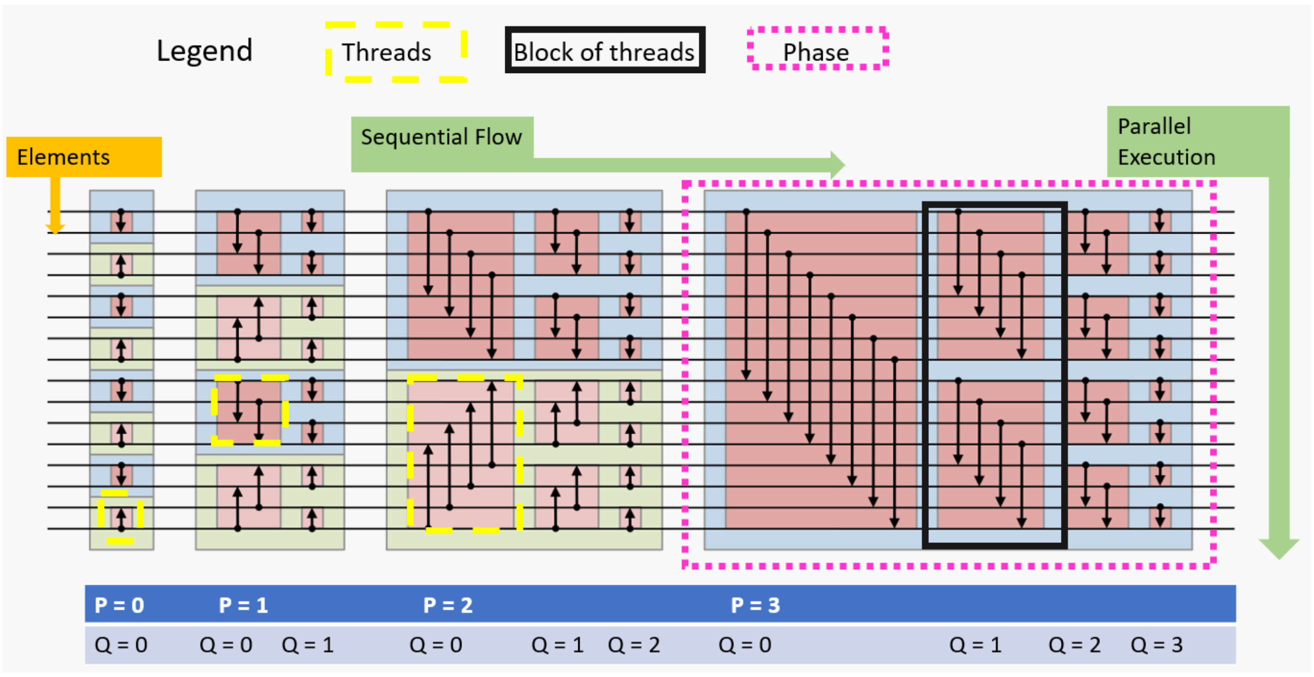


Figure 5: Bitonic Sorter

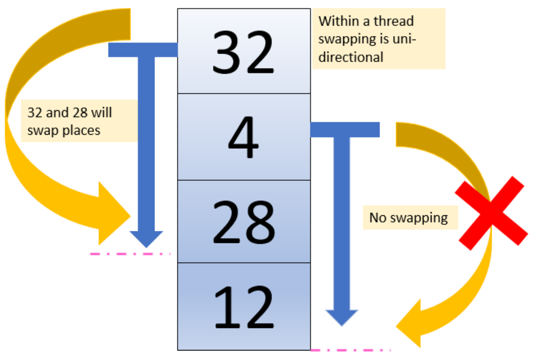

32
4
28
12
Within a thread swapping is uni-directional
32 and 28 will swap places
No swapping

Figure 6: Swapping

Algorithm 1 Swapping Algorithm

```
top_index=0
bottom_index = length/2
**while** (bottom_index does not exceed lower bound) **do**
    **if** $((array[top\_index] > array[bottom\_index]) \neq up)$ **then**
        swap elements
    **end if**
    increase top and bottom indices
**end while**
```

Algorithm 2 Merging sorted Sub-arrays

```
indices = [GRID SIZE]                       ▷ array with length equall to number of blocks
**while** (there are at least two unmerged arrays) **do**
    minimum = Min(unmerged_arrays[indices])
    copy minimum to result array
    indices[minimum] +1
**end while**
copy the remaining sub_array to result array
```

5 Hardware

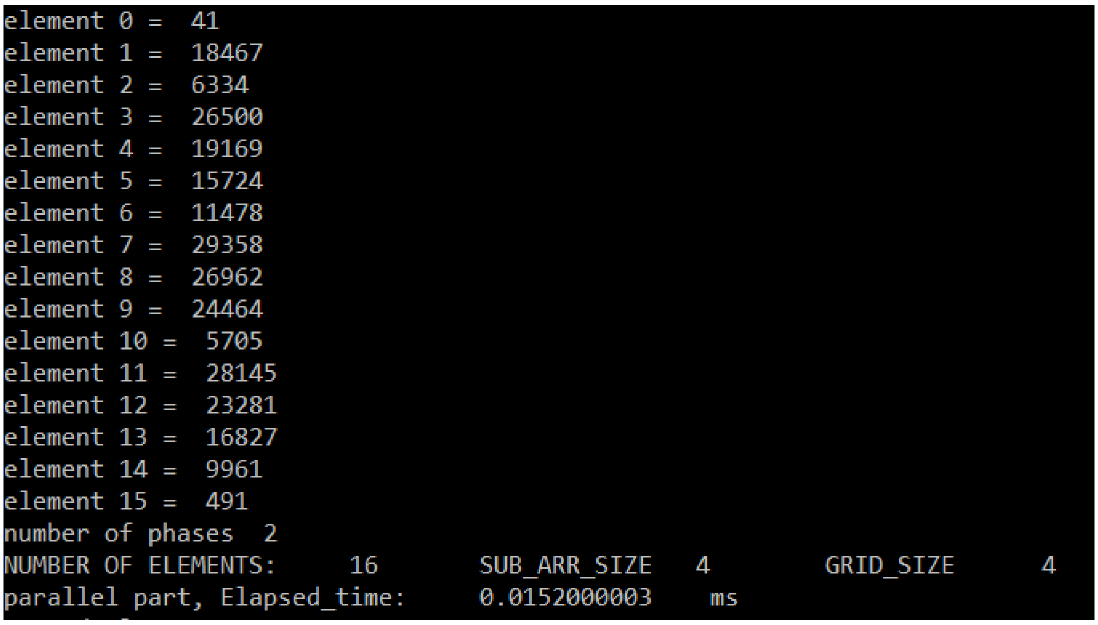
To better understand the settings under which the work has been done, Table 1 shows some important setup and machine properties.

PROPERTIES	MACHINE
CPU	Core i7 2.60 GHz
RAM	16.0 GB
GPU	GeForce 950M ($4GB$)
CUDA VERSION	9.1
VISUAL STUDIO VERSION	2012
WINDOWS VERSION	Windows 10
MEMORY LOCK RATE (KHz)	$9*10^5$
MEMORY BUS WIDTH (BITS)	128
PEAK MEMORY BANDWIDTH (GB/s)	28.8

Table 1: Machine properties

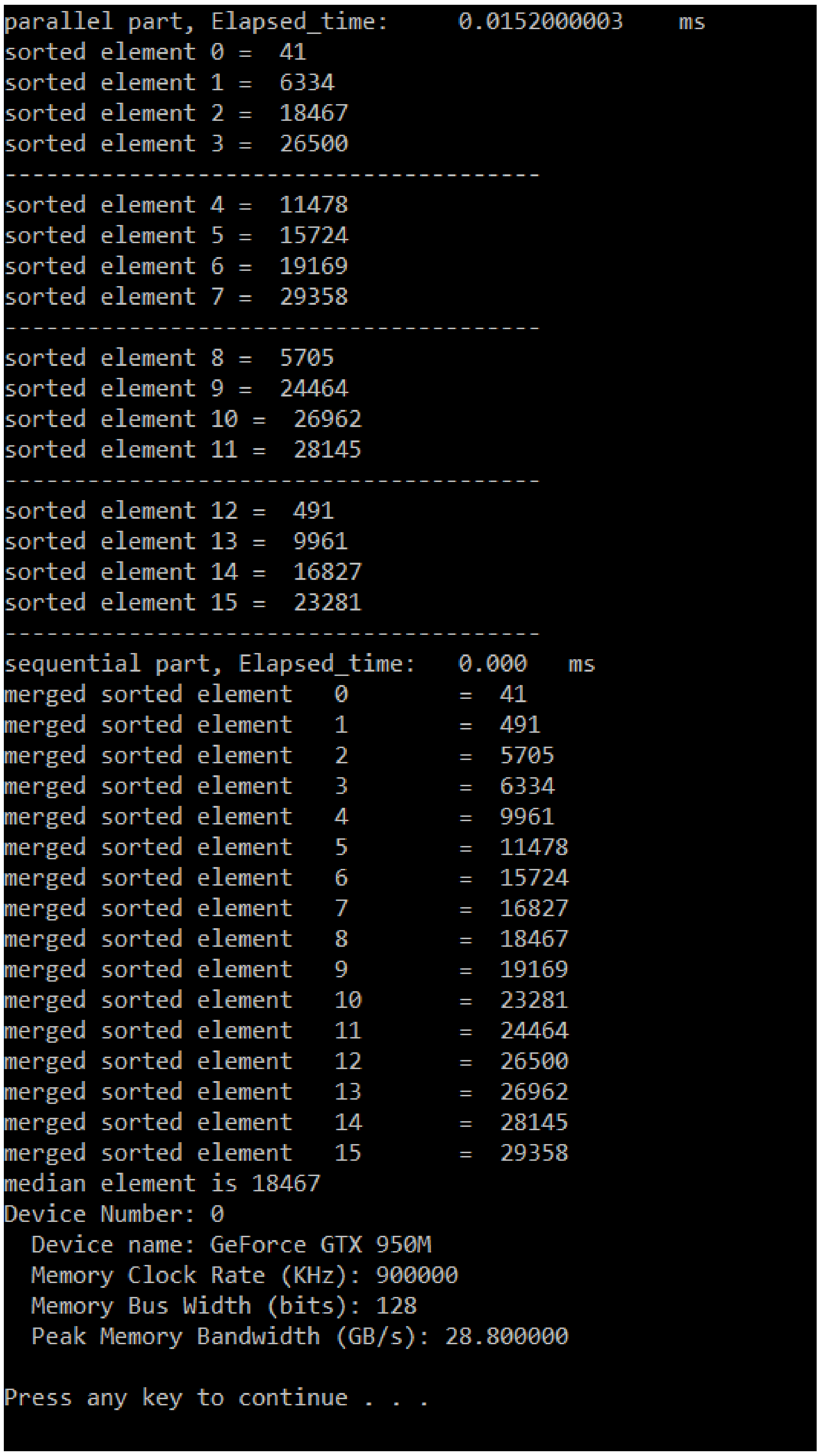
6 A Simple Example

One small array is used to show the result of the algorithm, see Figure 8. At the beginning, the array is initialized randomly, see Figure 7, then it is passed to the kernel with execution configuration (number of blocks = 4, number of threads per block = 2). As can be seen from the figure, first the sub-arrays are sorted (see dash lines separating sub-arrays). After that, sub-arrays are merged using the merging algorithm mentioned earlier. The result of merging is show in the same figure (merged sorted element)



```
element 0 =  41
element 1 =  18467
element 2 =  6334
element 3 =  26500
element 4 =  19169
element 5 =  15724
element 6 =  11478
element 7 =  29358
element 8 =  26962
element 9 =  24464
element 10 =  5705
element 11 =  28145
element 12 =  23281
element 13 =  16827
element 14 =  9961
element 15 =  491
number of phases  2
NUMBER OF ELEMENTS:     16       SUB_ARR_SIZE   4       GRID_SIZE      4
parallel part, Elapsed_time:     0.0152000003    ms
```

Figure 7: initializing an array with 16 random elements



```
parallel part, Elapsed_time:     0.0152000003    ms
sorted element 0 =  41
sorted element 1 =  6334
sorted element 2 =  18467
sorted element 3 =  26500
---------------------------------------
sorted element 4 =  11478
sorted element 5 =  15724
sorted element 6 =  19169
sorted element 7 =  29358
---------------------------------------
sorted element 8 =  5705
sorted element 9 =  24464
sorted element 10 =  26962
sorted element 11 =  28145
---------------------------------------
sorted element 12 =  491
sorted element 13 =  9961
sorted element 14 =  16827
sorted element 15 =  23281
---------------------------------------
sequential part, Elapsed_time:   0.000   ms
merged sorted element   0        =  41
merged sorted element   1        =  491
merged sorted element   2        =  5705
merged sorted element   3        =  6334
merged sorted element   4        =  9961
merged sorted element   5        =  11478
merged sorted element   6        =  15724
merged sorted element   7        =  16827
merged sorted element   8        =  18467
merged sorted element   9        =  19169
merged sorted element   10       =  23281
merged sorted element   11       =  24464
merged sorted element   12       =  26500
merged sorted element   13       =  26962
merged sorted element   14       =  28145
merged sorted element   15       =  29358
median element is 18467
Device Number: 0
  Device name: GeForce GTX 950M
  Memory Clock Rate (KHz): 900000
  Memory Bus Width (bits): 128
  Peak Memory Bandwidth (GB/s): 28.800000

Press any key to continue . . .
```

Figure 8: sorting an array of 16 random elements

7 Results And Conclusion

The algorithm was run for different sub-array sizes, three times for each size, the average processing time for the parallel and sequential parts were measured and recorded, see Table 2. As can be seen from the table, as the size of sub-arrays increases, the parallel part processing time increases, which could be related to the architecture shared array size). While for the sequential part, it becomes faster when the size of sub-arrays becomes larger.

Sub-arr length (shared mem size)	Parallel processing time (ms)	sequential part processing time (ms)
256	8.19	14.64
512	10.13	5.24
1024	16.5	2.39
2048	31.18	1.14

Table 2: Average Timing Measurements

8 Code

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include<device_functions.h>
#include<device_atomic_functions.h>
#include<malloc.h>
#include<stdlib.h>
#include<cuda.h>
#include<iostream>
#include <stdio.h>
#include<math.h>
#include"device_launch_parameters.h"
#include<algorithm>
#include <cuda_runtime_api.h> //for synchthreads
#include<ctime>

#define N (16)
//total number of elements
#define SUB_ARR_SIZE 4
//size of the smaller sub-arrays, shared array size
#define GRID_SIZE N/SUB_ARR_SIZE
//number of sub-arrays, number of blocks

// this function can be used for debugging
__device__ void print_arr(int * arr, int size, int thread_idx)
{
for(int i =0; i<size;i++)
printf("array[%d]=\t %d \n",size * thread_idx + i, arr[i]);
}

// this function finds the smallest value in specific positions,
//i.e., indices in the array respecting that the returned value is from a non-copied sub-arr
int SmallestElementIndex(int * arr, int * indices, int grid_size, int sub_arr_size)
```

```
{
int index = 0;

//find the first non-copied subarray
for(int k=0; k<grid_size; k++)
{
if(indices[k]/(k+1) < sub_arr_size)
{
index=k;
break;
}
}

//compare the value at the found index with other..
//..non-copied sub-arrays and return the index of the minimum
for(int i=index+1; i<grid_size; i++)
if(arr[indices[i]] < arr[indices[index]] && indices[i]/(i+1) < sub_arr_size)
index = i;
return index;
}

// this function finds the index of the min/equal element
int min_index (int * arr, int size)
{
int index = 0;
for(int i=1; i<size; i++)
if(arr[i] < arr[index])
index = i;
return index;
}

void merge(int * arr, int grid_size, int sub_arr_size, int * result_arr)
{
int   * indices        =        new int [grid_size];
//moving indices, each index points at the beginning of a sub-array at the beginning
int   * counters        =        new int [grid_size];
//counts the number of elements, in the corresponding sub-array,
//which have been copied to the result

int argmin, result_index=0;
//argmin is used for finding the last sub-array which has not copied completely yet
indices[0] = 0;
counters[0]=0;

//initialization
for(int k=1; k<grid_size; k++)
{
indices[k] = indices[k-1] + sub_arr_size;
counters[k]=0;
```

```
}

int merged_arrays =0;                  //number of completely copied sub-arrays

//while there are more than one sub-array not copied
while(merged_arrays <grid_size-1)
{
//find the smallest value among the ones pointed by indices
argmin = SmallestElementIndex(arr, indices, grid_size, sub_arr_size);

//copy the smallest value to the result and increase the corresponding index
result_arr[result_index] = arr[indices[argmin]];
indices[argmin]++;
counters[argmin]++;

if((counters[argmin]%sub_arr_size)==0)
merged_arrays++;
result_index++;
}
//for the last sub-array
int min_idx = indices[min_index(counters, grid_size)];

for(int i = result_index; i< (grid_size * sub_arr_size); i++, min_idx++)
result_arr[i] = arr[min_idx];
delete [] counters;
delete [] indices;
}
//This function takes a thread-part array along with the number of elements,
//thread indices are for debugging
__device__ void swap(int * sub_arr, bool up, int thread_size, int thread_idx, int p, int q)
{

int j=0 , i = thread_size/2;
//two indices, one pointing at the beggining of the array, the pther at the half
int temp;
// used for swapping two elements

//print_arr(sub_arr, thread_size, thread_idx); // was used for debugging

while(i<= thread_size-1)
{
//this comparison cares about the direction as well as the values,
//values should be sorted in the defined direction, or swap
if((sub_arr[j] > sub_arr[i]) == !up)
{
temp = sub_arr[j];
sub_arr[j] = sub_arr[i];
sub_arr[i] = temp;
}
```

```
i++;
j++;
}
}

//bitonic sort kernel
__global__ void bitonic(int * arr)
{
__shared__ int  shared_arr [SUB_ARR_SIZE] ;
//the block shared array
__shared__ int threads_per_block;
//dynamic block size, phase (p) and sub-phase (q) dependant
__shared__ int thread_size;
//dynamic thread size, phase and subphase dependant
int direction_assisstant;
//to be used in deciding the direction of comparing elements for each thread
int idx = threadIdx.x + blockDim.x * blockIdx.x;
//global thread index

//thread 0 in each block is the responsible for..
//.. controlling block and thread sizes for each sub-phase
if(threadIdx.x == 0)
{
threads_per_block = SUB_ARR_SIZE;
thread_size= 2;
}
//in parallel, each thread copies two elements from the
//original array to the corresponding position in the shared array
for(int i=0; i< 2; i++)
{
shared_arr[threadIdx.x*2 +i] = arr[idx*2 +i];
}
__syncthreads();
//here,  we are sure that data has been copied from global to shared mem completely

//loop for each phase, p
for(int p=0; p<log2f(SUB_ARR_SIZE); p++)
{
//printf("---------------------------------------\n");
//thread 0 in each block changes the sizes of..
//.. blocks and threads using the value of p
if(threadIdx.x == 0)
{
threads_per_block /=powf(2, p+1);
thread_size= SUB_ARR_SIZE/threads_per_block;
}
__syncthreads();
//Here all the threads in this block know the
//proper size of themselves and blocks, as well
```

```
for(int q=0; q<=p; q++)
{
//activate the required number of threads only
if(threadIdx.x <threads_per_block )
{
// calculate the up-to-date global index for each
//block using threads_per_block instead of blockDim.x
idx = threadIdx.x + threads_per_block * blockIdx.x;

direction_assisstant = powf(2,q);
//call the swap function which does all the needed
// swappings inside the corresponding part of the shared array
swap(shared_arr + threadIdx.x * thread_size, (threadIdx.x/direction_assisstant)
//idx, p, andq are for debugging only
}
__syncthreads();

//here all threads in this block have finished the current subphase
//update thread and block sizes for next subphase and suspend
// til all the threads in the current block have known the new values
if(threadIdx.x == 0)
{
threads_per_block *= 2;
thread_size = SUB_ARR_SIZE/threads_per_block;
}
__syncthreads();
}
}
__syncthreads();
//here, all block-thread have finished all the sub-phases
//copy the sorted shared array to the propper position in the result array, on parallel
for(int i=0; i< 2; i++)
{
arr[idx * 2 +i] = shared_arr[threadIdx.x * 2 + i];
}
}
int main()
{

int * arr;                                                   //to save the unsorted data
int * resulted_arr;                                   //to save the partially-sorted data
int * merged_sorted_arr;                   //to save the completely-sorted data
int * dev_arr;                                          //to use in device
int size = N * sizeof(int);

cudaEvent_t start, stop;
clock_t begin, end;
//to measure the parallel processing time in sorting the array partially
```

```
float elapsed_time;


//random intialization, memory allocation

cudaMalloc((void **) &dev_arr, size);
arr = (int *)malloc(size);
merged_sorted_arr = (int *)malloc(size);
resulted_arr = (int *)malloc(size);

for(int i=0; i<N; i++)
{
arr[i] = rand();

printf("element %d =  %d \n", i, arr[i]);
}

//copy data from host to device
cudaMemcpy(dev_arr, arr, size, cudaMemcpyHostToDevice);

std::cout<<"number of phases  "<<log10(SUB_ARR_SIZE)/log10(2)<<std::endl;

printf("NUMBER OF ELEMENTS:\t%d\t SUB_ARR_SIZE \t%d\t
GRID_SIZE \t%d\n", N, SUB_ARR_SIZE, GRID_SIZE);

//record timing
cudaEventCreate(&start);
cudaEventRecord(start,0);

///////////////////////////////////////////// kernel invokation
bitonic<<<GRID_SIZE, SUB_ARR_SIZE/2>>>(dev_arr);

cudaEventCreate(&stop);
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed_time, start, stop);

printf("parallel part, Elapsed_time:\t %3.10f \t ms \n", elapsed_time);

//copying the result from device to host
cudaMemcpy(resulted_arr,dev_arr, size, cudaMemcpyDeviceToHost);

for(int i=0; i<N;)
{
printf("sorted element %d =  %d \n", i, resulted_arr[i]);
i++;
//to split sub-arrays in display, for the sake of easyness of monitoring results
if(i % SUB_ARR_SIZE ==0)
printf("---------------------------------------\n");
//This was for debugging, when errors show this line of hashes#
if(i<N)
if(resulted_arr[i] < resulted_arr[i-1] && ((i % SUB_ARR_SIZE) !=0) )
printf("#############################################\n");
}

//To merge sorted sub-arrays, measre sequential complexity
begin = clock();
merge(resulted_arr, GRID_SIZE, SUB_ARR_SIZE, merged_sorted_arr);
end = clock();
elapsed_time = (float)(end-begin)/CLOCKS_PER_SEC;
printf("sequential part, Elapsed_time:\t %3.3f \t ms \n", elapsed_time);

//to print completely-sorted array
for(int i=0; i<N;)
{
printf("merged sorted element \t%d\t =  %d \n", i, merged_sorted_arr[i]);
i++;
//again for debugging, when errors, show a line of hashes#
if(i<N)
if(merged_sorted_arr[i] < merged_sorted_arr[i-1])
printf("#############################################\n");
}
std::cout<<"median element is "<<merged_sorted_arr[N/2]<<"\n";

//liberating the allocated memory in both host and device
cudaFree(dev_arr);
free(arr);
free(resulted_arr);
free(merged_sorted_arr);

int nDevices;

cudaGetDeviceCount(&nDevices);
for (int i = 0; i < nDevices; i++) {
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, i);
printf("Device Number: %d\n", i);
printf("  Device name: %s\n", prop.name);
printf("  Memory Clock Rate (KHz): %d\n",
prop.memoryClockRate);
printf("  Memory Bus Width (bits): %d\n",
prop.memoryBusWidth);
printf("  Peak Memory Bandwidth (GB/s): %f\n\n",
2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
}
}
```

References

[1] Bitonic sort. https://en.wikipedia.org/wiki/Bitonic_sorter. Accessed June 25, 2018.

[2] Compute capability. https://devblogs.nvidia.com/how-query-device-properties-and-/ /handle-errors-cuda-cc/. Accessed June 25, 2018.

[3] Difference between serial and parallel processing. http://www.itrelease.com/2017/11/ difference-serial-parallel-processing/. Accessed June 20, 2018.

[4] The supercomputing blog. http://supercomputingblog.com/cuda/ cuda-tutorial-2-the-kernel/. Accessed June 23, 2018.

[5] Nvidia Corporation. *CUDA C PROGRAMMING GUIDE PG-02829-001_v9.1.* March 2018.

[6] Frederic Lardinois. Ibm and the doe launch the worlds fastest supercomputer. https://techcrunch.com/2018/06/08/ ibmsnewsummit-supercomputer-for-the-doe-delivers-200petaflops/. Accessed June 8, 2018.

[7] Craig Zilles, Sanjay J. Patel, Sarita V. Adve, Ralph E. Johnson, and Rakesh Kumar. *Parallel computing Research at illinois The UPCRC Agenda.* 2008.