

Parallel Implementation of Scheduling Algorithms on GPU using CUDA



Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology

In

Computer Science and Engineering

by

U.Harishraj(18BCE2381)

Prajeeth Kumar, M.J.(18BCE2354)

CSE4001 - Parallel and Distributed Computing

Abstract

The future of computation is the GPU, i.e. the Graphical Processing Unit. The graphics cards have shown the tremendous power in the field of image processing and accelerated generating of 3D scenes, and the computational capability of GPUs have promised its developing into great parallel computing units. It is quite simple to program a graphical processor to perform many parallel tasks. But after understanding the various aspects of the graphical processor, it can be used to perform other useful tasks as well. This paper shows how CUDA can fully utilize the tremendous power of these GPUs. CUDA is NVIDIA's parallel computing architecture which enables terrible increase in computing performance, by gearing the power of the GPU. In the first phase, several operating system algorithms in single threaded CPU environment are implemented using C language, then the same algorithms are implemented on CUDA and CUDA enabled GPU in a parallel environment and finally comparison of their performance and results to their implementation in GPU and CPU are shown.

Introduction

GPU computation has provided a huge bound over the CPU with respect to the computational speed. Hence it is one of the most interesting areas of modern computational research and development. GPU is a graphical processing unit which primarily enables us to run high definition graphics on our PC, which are the essential demand of modern computing world. The main job of the GPU is to compute 3D functions. As these types of calculations are very difficult to perform on the CPU, the GPU can help them to run more efficiently. Though, GPU is introduced for graphical purposes, it has now evolved into computing, accuracy and performance. Using the GPU for processing non graphical objects is known as the General Purpose GPU or GPGPU, which is used for performing very complex mathematical operations in parallel to achieve low time complexity.

The arithmetic power of the GPGPU is a result of its highly specialized computing architecture. This report proposes the use of CUDA technology on parallel platform to enhance the performance of the operating system scheduling algorithms: First Come First Serve (FCFS) algorithm, Shortest Job First (SJF) algorithm, Round-Robin (RR) algorithm and Priority based scheduling (PBS) algorithm. Scheduling is the process by which threads, processes and data flows are given access to system resources (e.g. processor time and communications bandwidth). This is usually done to maintain balance and share the system resources effectively and achieve a target quality of service.

The need for these scheduling algorithm arises from the requirement for most modern systems to perform multitasking (executing more than one process at a time) and multiplexing (send multiple data streams simultaneously across a single physical channel) . The operating system algorithms form the core for resource allocation and their maximum utilization. Scheduling is a complex job which may require an extensive processing and thus is better performed on a parallel processor. This work demonstrates the performance of scheduling algorithms as to check how fast the algorithm could perform. It compares and contrast the working time and corresponding

efficiency of normal execution of the serial code implemented in C language on single threaded CPU as compared to GPU implementation. The CUDA implementation of the scheduling algorithms uses the CUDA-C language and the recent NVIDIA CUDA Software Development Kit (SDK) 6.0.

Problem Statement

To compare the execution time between sequential and parallel execution of scheduling algorithms using CUDA.

Literature Survey

Publication Year	Title	Abstract	Scope
Journal of the Chinese Institute of Engineers ,32(7):915-921, 2019.	An efficient sorting algorithm with CUDA	The proposed sorting algorithm is	The current implementation is on
		<p>optimized for modern GPU architecture with the capability of sorting elements represented by integers, floats and structures, while the new merging method gives a way to merge two ordered lists efficiently on GPU without using the slow atomic functions and uncoalesced memory read.</p> <p>Adaptive strategies are used for sorting disorderly or nearly- sorted lists, large or small lists</p>	<p>NVIDIA CUDA with multi-GPUs support, and is being migrated to the new born Open Computing Language (OpenCL). Extensive experiments demonstrate that our algorithm has better performance than previous GPU-based sorting algorithms and can support real-time applications.</p>

International Journal of Computer Science and Information Technologies, Vol. 5(2) , 2016	A Survey on CUDA	A major challenge in image processing is to attain high precision and real - time performance which is difficult to achieve even with most powerful CPU. CUDA has eliminated the bottleneck of large execution time by processing a digital image parallely rather than sequentially. In this paper we critically analyzed various parallel computing techniques	After comparison of CUDA with other parallel computing techniques it is clear that CUDA is much fast. So there is a great scope of NVIDIA's CUDA architecture. CUDA is very efficient and can solve any complex problem in milliseconds
International Journal of Computer Science and Network Security, Vol.18 No.6, June 2018	GPU Acceleration of Image Processing Algorithm Based on MATLAB CUDA	MATLAB is frequently exposed to the memory latency and the issues of slow execution. In order to accelerate MATLAB's processing, we use NVIDIA'S CUDA	This paper presents a demonstration of mixed programming concept by integration MATLAB with CUDA. This is done by replacing the time consuming parts of
		parallel processing architecture. We note that processing can be speed up significantly by interfacing MATLAB with CUDA and parallelizing the most time consuming portion of MATLAB's code white balance	the algorithms running in MATLAB CPU and porting to the Graphical Processing Unit (GPU). By applying this methodology, we not only use the rich programming features of MATLAB, but also minimize its performance bottleneck.

Informatics in Medicine, Vol 9, 2017	Survey of using GPU CUDA programming model in medical image analysis	With the technology development of medical industry, processing data is expanding rapidly and computation time also increases due to many factors like 3D, 4D treatment planning, the increasing sophistication of MRI pulse sequences and the growing complexity of algorithms. Graphics processing unit (GPU) addresses these problems and gives the solutions for using their features such as, high computation throughput, high memory bandwidth, support for floating-point arithmetic and low cost.	In this review, we discussed most common areas on GPU computing in medical imaging analysis. The existing works of medical image analysis are investigated and performance gains with the CUDA programming are discussed. This investigation intimates the important of GPU computing in the area of medical industry.
International Journal of Computer Science 29 June 2019.	Parallelizing Multiple Flow Accumulation Algorithm using CUDA	In this paper, the Multiple Flow Direction (MFD) algorithm for watershed analysis is implemented and evaluated on multi-core Central Processing Units (CPU) and many-core Graphics	This has caused great interest in processing, analysis, and visualization of Big geospatial and spatio-temporal data, both offline and a s fast data streams in recent years. GPGPU

		Processing Units (GPU), which provides significant improvements in performance and energy usage. The implementation is based on NVIDIA CUDA (Compute Unified Device Architecture) implementation for GPU	represents a new parallel computing paradigm that provides significant gains in term of performance improvements in various data intensive applications. This paper shows that using OpenACC and CUDA parallel programming paradigms can significantly improve the performance
--	--	--	--

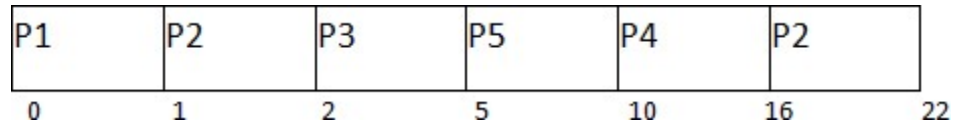
Gantt chart

1)FCFS

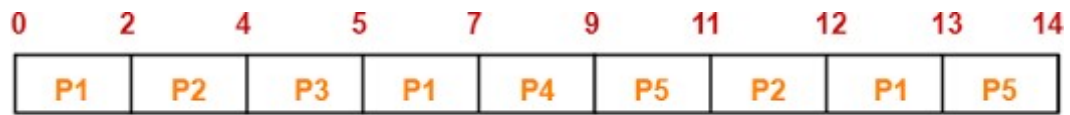


Gantt Chart

2) Priority

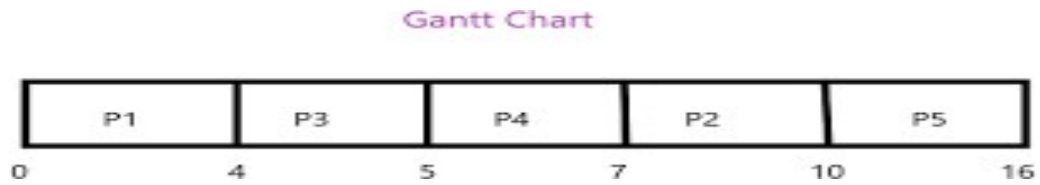


3) Round robin



Gantt Chart

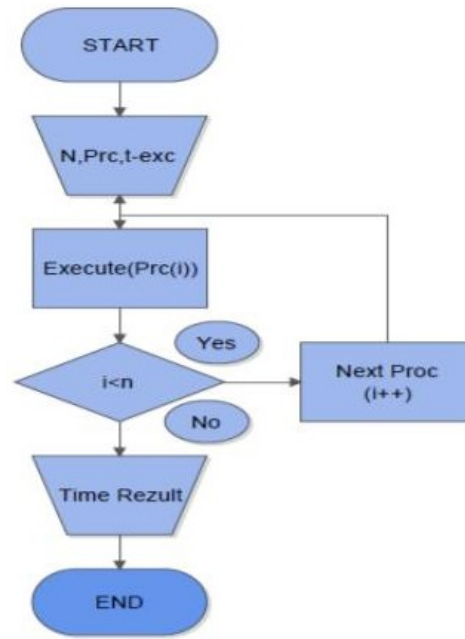
4) SJF



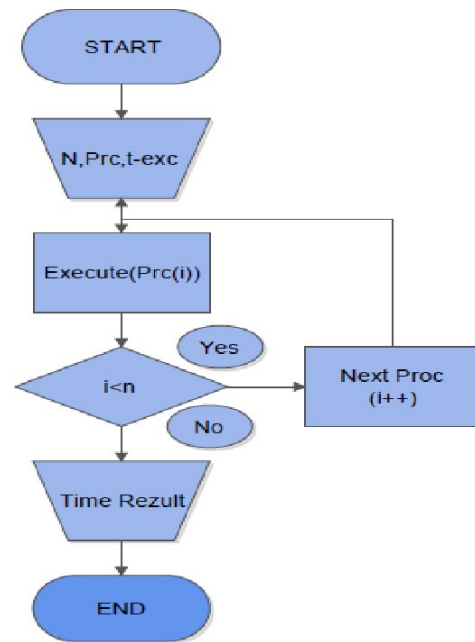
Gantt Chart

Proposed model

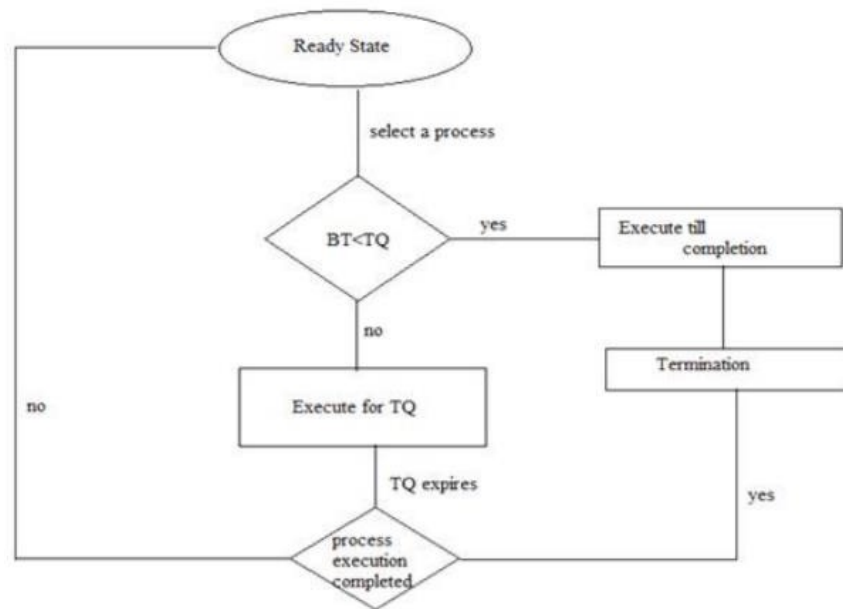
1) FCFS



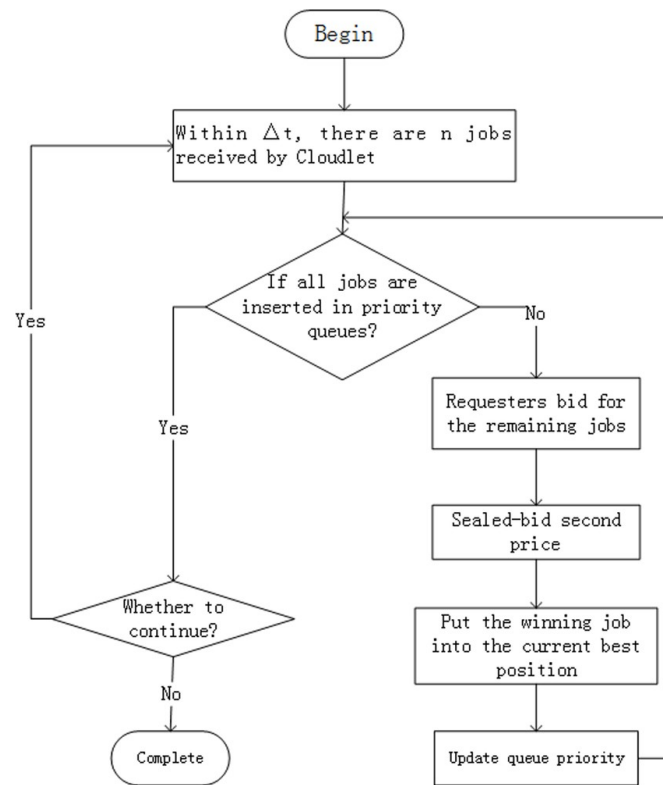
2) SJF



3) Round robin



4) Priority Scheduling



Methodology

1) First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

2) Shortest Job Next (SJN)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processor should know in advance how much time process will take.

3) Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

4) Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

Hardware Configurations

1. **Motherboard** - you must remember that in better graphic cardSupplementary Power Connectors need 2x 6-pins and the Bus Support - PCI Express 3.0. If you want to add more cards you have on board an adequate number of PCI slots.
2. **Power supply** - The power supply should have sufficient power to easily handle the PC and installed graphics card. In the specification of each card you can find such information. In the computer which I use I have 1000W Power Supply (2X GTX graphic card - see picture below).
3. **PC housing** - When performing calculations using graphics cards is growing rapidly the temperature. It is therefore necessary to choose housing that will provide good cooling. In the computer which I use equipped with two GTX 680 cards I use housing Master HAF XM Cooler (see link below), which provides a very good cooling of all components.

Software Configurations

1. **You can use one of the following programming language:** Open CL (OpenCL programs from C, C++, Fortran, or Python.), CUDA (CUDA programs from C, C++ or Python),
2. **You can use software engineering** that support calculations with use of the graphics cards (matlab, mathematica).

IMPLEMENTATION:

As stated earlier the main objective of the present work is to analyze the various CPU scheduling algorithms. The foremost criterion for evaluating CPU scheduling is the waiting time and burst time of the processes that are under same set of conditions. The paper has implemented the four major scheduling algorithms namely **First Come First Serve** (FCFS) Scheduling, **Shortest-Job-First** (SJF) Scheduling, **Priority Based Scheduling** (PBS) and **Round Robin** (RR) Scheduling firstly on single threaded CPU environment and calculated the execution time of each algorithm, then, the same algorithms are implemented with NVIDIA's GPU programming environment, CUDA v6.0. It then compares the execution time of the algorithms on both platform and calculates the speed up achieved in execution of the algorithms on GPU over CPU. The basics of CUDA code implementation are:

- a. Allocate the memory on the CPU.
- b. Allocate the same amount of memory on GPU using library function "CudaMalloc".
- c. Input the data in memory allocated in CPU.
- d. Copy data from CPU to GPU memory using another library function CudaMemCpy having parameter (CudaMemcpyHostToDevice) to give the direction of the copy.
- e. Processing is now performed in GPU memory using kernel calls. Kernel calls transfer the control from CPU to GPU and also specify the number of grids, blocks and threads required for your program. In other words it defines the parallelism.
- f. Copy back the final data from GPU to CPU memory using library function CudaMemCpy having parameter (CudaMemcpyHostToDevice)
- g. Release the GPU memory or other threads using the library function CudaFree. Setting up the environment and writing programs in CUDA is a fairly simple task. But, it requires that the one must have a thorough knowledge of the architecture and knowledge of writing parallel codes.

The most important part of programming in CUDA is the kernel calls wherein the programmer must determine the parallelism that is required by the program. The division of given data into appropriate number of threads is the major area which defines a successful code.

The implementation of gpu enabled cuda was run in google collab.

FCFS ON CPU:

Code:

```
#include <stdio.h>
```

```
#include<time.h>
```

```
int waitingtime(int n, int burst_time[], int wait_time[])
```

```
{
```

```
    wait_time[0] = 0;
```

```
    for (int i = 1; i < n ; i++ )
```

```
        wait_time[i] = burst_time[i-1] + wait_time[i-
```

```
        1] ; return 0;
```

```
}
```

```
int turnaroundtime(int n,int burst_time[], int wait_time[], int tat[])
```

```
{
```

```
    int i;
```

```
    for ( i = 0; i < n ; i++)
```

```
        tat[i] = burst_time[i] +
```

```
        wait_time[i]; return 0;
```

```
}
```

```
int avgtime(int n, int burst_time[]) {
```

```
    int wait_time[n], tat[n], total_wt = 0, total_tat =
```

```
    0; int i;
```

```
    waitingtime( n, burst_time, wait_time);
```

```
    turnaroundtime(n, burst_time, wait_time, tat);
```



```

printf("Burst\t Waiting\t Turnaround \n");

for ( i=0; i<n; i++) {
    total_wt    =    total_wt    +
    wait_time[i]; total_tat = total_tat
    + tat[i];
    printf("%d\t\t %d \t%d\n", burst_time[i], wait_time[i], tat[i]);
}
printf("Average waiting time = %f\n", (float)total_wt / (float)n);
printf("Average turn around time = %f\n", (float)total_tat /
(float)n); return 0;
}

int main()
{
    double
    time_spent=0.0;
    clock_t begin=clock();
    const int n = 4;

    int burst_time[] = {5, 8, 12,
6}; avgtime(n, burst_time);
    clock_t end=clock();
    time_spent+=(double)(end-
begin)/CLOCKS_PER_SEC; printf("time elapsed in
cpu is %f s",time_spent); return 0;
}

```

Output:

```
"C:\Users\Dr. N Jaisankar\Desktop\pee files\codeblock programs\fcfs_a.exe"
Burst      Waiting      Turnaround
5           0           5
8           5          13
12          13          25
6           25          31
Average waiting time = 10.750000
Average turn around time = 18.500000
time elapsed in cpu is 0.003000 s
Process returned 0 (0x0)   execution time : 7.150 s
Press any key to continue.
```

FCFS ON GPU:

Code:

```
%%cu
```

```
#include<stdio.h
```

```
>
```

```
#include <thrust/reduce.h>
```

```
#include <thrust/execution_policy.h>
```

```
int* findWaitingTime(int n, int bt[])
```

```
{
```

```
    static int wt[100];
```

```
    wt[0] = 0;
```

```
    for (int i = 1; i < n ; i++ )
```

```
        wt[i] = bt[i-1] + wt[i-
```

```
        1];
```

```
    return wt;
```

```
}
```

```
global void findTurnAroundTime(int * d_bt, int * d_wt, int * d_tat)
```

```
{
```

```
int idx = threadIdx.x;
```

```

    d_tat[idx] = d_bt[idx] + d_wt[idx];
}

int main(int argc, char **argv)
{
    int *d_bt;
    int
    *d_wt;
    int
    *d_tat;
    const int n = 4;
    //device copies of variables
    const int ARRAY_BYTES = n*sizeof(int);
    //allocate space for device copies of variables

    cudaMalloc((void **,*)&d_bt,
    ARRAY_BYTES);    cudaMalloc((void
    **,*)&d_wt,    ARRAY_BYTES);
    cudaMalloc((void **,*)&d_tat,
    ARRAY_BYTES);

    int tat[n];
    int bt[n] = {5,8,12,6};
    int avg[2] = {0,0};

    float    elapsed=0;
    cudaEvent_t  start,  stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start,
    0);

```

```
int *wt = findWaitingTime(n, bt);
```

```

//copy inputs to device

cudaMemcpy(d_bt,          bt,          ARRAY_BYTES,
cudaMemcpyHostToDevice);
cudaMemcpy(d_wt,wt,ARRAY_BYTES,
cudaMemcpyHostToDevice);
// launch the function kernel on gpu

findTurnAroundTime<<< 1, n >>>(d_bt, d_wt, d_tat);
//copy result back to host
cudaMemcpy(tat,d_tat,ARRAY_BYTES, cudaMemcpyDeviceToHost);

avg[0] = thrust::reduce(thrust::host, wt, wt + 3,
1); avg[1] = thrust::reduce(thrust::host, tat, tat +
3, 1);

cudaEventRecord(stop,          0);
cudaEventSynchronize          (stop);
cudaEventElapsedTime(&elapsed, start,
stop);          cudaEventDestroy(start);
cudaEventDestroy(stop);

printf("Processes Burst time Waiting time Turn around time\n");
for (int i=0; i<n; i++)
{

    printf(" %d\t ",(i+1));
    printf("    %d\t ", bt[i]);
    printf("    %d\t",wt[i]);
    printf("    %d\n",tat[i]);
}

```

```

printf("Average      waiting      time      =
%f", (float)avg[0]/(float)n); printf("\n");
printf("Average turn around time = %f\n", (float)avg[1]/(float)n);
//cleanup
cudaFree(d_bt);
cudaFree(d_wt)
;
cudaFree(d_tat)
;

printf("The elapsed time in gpu was %f ms", elapsed);

}

```

Output:

```

Processes    Burst time    Waiting time    Turn around time
1            5             0              5
2            8             5             13
3           12            13            25
4            6            25            31
Average waiting time = 4.750000
Average turn around time = 11.000000
The elapsed time in gpu was 0.088096 ms

```

The execution time of fcfs algorithm in cpu is 0.003

s. The execution time of fcfs algorithm in gpu is 0.08ms.

SJF IN CPU:

Code:

```
#include<stdio.h
```

```
>
```

```
#include<time.h
```

```
>
```

```
void run()
```



```

{
    int
    wt[4],tat[4],i,j,n,total=0,pos,temp;

    float avg_wt,avg_tat;

    n =4;

```

```

    int bt[4] = {2,5,3,6};

```

```

    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j+
        +)
        {
            if(bt[j]<bt[pos]
            ) pos=j;
        }

```

```

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp
    ;
}

```

```

    wt[0]=0;
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j+
        +)

```

```
wt[i]+=bt[j];
```

```

        total+=wt[i];
    }

    avg_wt=(float)total/n
; total=0;

printf("\nBurst   Time   \tWaiting   Time\tTurnaround   Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i]
; total+=tat[i];
    printf("\n%d\t\t %d\t\t\t %d",bt[i],wt[i],tat[i]);
}

avg_tat=(float)total/n;
printf("\n\nAverage   Waiting   Time=%f",avg_wt);
printf("\nAverage           Turnaround
Time=%f\n",avg_tat);
}

void main()
{
    double
    time_spent=0.0;
    clock_t begin=clock();
    run();
    clock_t end=clock();
    time_spent=(double)(end-
begin)/CLOCKS_PER_SEC; printf("time elapsed in
cpu is %f seconds",time_spent);
}

```

Output:

```

Burst Time      Waiting Time      Turnaround Time
2                0                2
3                2                5
5                5                10
6                10               16

Average Waiting Time=4.250000
Average Turnaround Time=8.250000
time elapsed in cpu is 0.004000 seconds
Process returned 39 (0x27)   execution time : 5.494 s
Press any key to continue.

```

SJF IN GPU:

Code:

```
%%cu
```

```
#include<stdio.h
```

```
>
```

```
#include <thrust/reduce.h>
```

```
#include <thrust/execution_policy.h>
```

```
#include <thrust/sort.h>
```

```
int* findWaitingTime(int n, int bt[])
```

```
{
```

```
    static int wt[100];
```

```
    wt[0]=0;
```

```
    for(int i=1;i<n;i++)
```

```
    {
```

```
        wt[i]=0;
```

```
        for(int
```

```
            j=0;j<i;j++)
```

```
            wt[i]+=bt[j];
```

```
    }
```

```

    return wt;
}

global void findTurnAroundTime(int * d_bt, int * d_wt, int * d_tat)
{
    int idx = threadIdx.x;
    d_tat[idx] = d_bt[idx] + d_wt[idx];
}

int main()
{
    int *d_bt;
    int
    *d_wt;
    int
    *d_tat;
    const int n = 4;
    const int ARRAY_BYTES = n*sizeof(int);

    cudaMalloc((void **)&d_bt,
    ARRAY_BYTES);    cudaMalloc((void
    **)&d_wt,    ARRAY_BYTES);
    cudaMalloc((void **)&d_tat,
    ARRAY_BYTES);

    int tat[n];
    int bt[n] = {2,5,3,6};
    int avg[2] = {0,0};

    float    elapsed=0;
    cudaEvent_t start, stop;

```

```
cudaEventCreate(&start)
```

```
;
```

```
cudaEventCreate(&stop)
```

```
;
```

```

cudaEventRecord(start, 0);

thrust::sort(thrust::host, bt, bt + n);

int *wt = findWaitingTime(n, bt);

cudaMemcpy(d_bt,          bt,          ARRAY_BYTES,
cudaMemcpyHostToDevice);
cudaMemcpy(d_wt, wt, ARRAY_BYTES,
cudaMemcpyHostToDevice);  findTurnAroundTime<<< 1, n
>>>(d_bt, d_wt, d_tat); cudaMemcpy(tat, d_tat, ARRAY_BYTES,
cudaMemcpyDeviceToHost);

avg[0] = thrust::reduce(thrust::host, wt, wt + 3,
1); avg[1] = thrust::reduce(thrust::host, tat, tat +
3, 1);

cudaEventRecord(stop,          0);
cudaEventSynchronize          (stop);
cudaEventElapsedTime(&elapsed, start,
stop);          cudaEventDestroy(start);
cudaEventDestroy(stop);

printf("\nBurst Time \tWaiting Time\tTurnaround Time");
for(int i=0; i<n; i++)
{
    printf("\n%d\t\t %d\t\t\t %d", bt[i], wt[i], tat[i]);
}

```



```

printf("\n\nAverage  Waiting  Time=%f", (float)avg[0]/(float)n);
printf("\nAverage                                Turnaround
Time=%f\n", (float)avg[1]/(float)n);

cudaFree(d_bt);
cudaFree(d_wt)
;
cudaFree(d_tat)
;

printf("The elapsed time in gpu was %f ms", elapsed);

}

```

Output:

```

Burst Time      Waiting Time      Turnaround Time
2              0              2
3              2              5
5              5              10
6              10             16

Average Waiting Time=2.000000
Average Turnaround Time=4.500000
The elapsed time in gpu was 0.059360 ms

```

The execution time of sjf algorithm in cpu is 0.004
s. The execution time of sjf algorithm in gpu is
0.05ms.

PBS IN CPU:

Code:

```
#include<stdio.h>

#include<time.h>

> void run()
{

    int
    wt[20],tat[20],i,j,n,total=0,pos,temp;

    float avg_wt,avg_tat;

    n = 4;
    int bt[4] = {2,5,3,8};
    int pr[4] = {2,1,4,3};

    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(pr[j]<pr[pos])
                pos=j;
        }

        temp=pr[i];
        pr[i]=pr[pos];
        pr[pos]=temp
    ;
}
```

```
temp=bt[i];
```

```

    bt[i]=bt[pos];
    bt[pos]=temp
;
}

wt[0]=0;
for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j+)
    wt[i]+=bt[j];

    total+=wt[i];
}

avg_wt=(float)total/(float)n
; total=0;

printf("\nPriority\t Burst Time \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i]
    ; total+=tat[i];

    printf("\n%d\t\t %d\t\t %d\t\t\t %d",pr[i],bt[i],wt[i],tat[i]);
}

```

```
avg_tat=(float)total/(float)n;
printf("\n\nAverage           Waiting
Time=%f",avg_wt);
printf("\nAverage Turnaround Time=%f\n",avg_tat);
```

```
}
```

```
void main()
{
    double
    time_spent=0.0;
    clock_t begin=clock();
    run();
    clock_t end=clock();
    time_spent=(double)(end-
    begin)/CLOCKS_PER_SEC; printf("time elapsed in
    cpu is %f seconds",time_spent);
}
```

Output:

```
Priority      Burst Time      Waiting Time      Turnaround Time
1             5              0                5
2             2              5                7
3             8              7               15
4             3             15               18

Average Waiting Time=6.750000
Average Turnaround Time=11.250000
time elapsed in cpu is 0.001000 seconds
Process returned 39 (0x27)   execution time : 4.286 s
Press any key to continue.
```

PBS IN GPU:

Code:

```
%%cu
```

```
#include<stdio.h
```

```
>
```

```
#include <thrust/reduce.h>
```

```
#include <thrust/execution_policy.h>
```

```
#include <thrust/sort.h>
```

```
int* findWaitingTime(int n, int bt[])
```

```
{
```

```
    static int wt[100];
```

```
    wt[0]=0;
```

```
    for(int i=1;i<n;i++)
```

```
    {
```

```
        wt[i]=0;
```

```

    for(int
        j=0;j<i;j++)
            wt[i]+=bt[j];
    }
    return wt;
}

global void findTurnAroundTime(int * d_bt, int * d_wt, int * d_tat)
{
    int idx = threadIdx.x;
    d_tat[idx] = d_bt[idx] + d_wt[idx];
}

int main()
{
    int *d_bt;
    int
    *d_wt;
    int
    *d_tat;
    const int n = 4;
    const int ARRAY_BYTES = n*sizeof(int);

    cudaMalloc((void **)&d_bt,
        ARRAY_BYTES);    cudaMalloc((void
        **)&d_wt,        ARRAY_BYTES);
    cudaMalloc((void **)&d_tat,
        ARRAY_BYTES);

    int tat[n];
    int bt[n] = {2,5,3,8};

```

```
int pt[n]={2,1,4,3};
```

```
int avg[2] = {0,0};
```



```

float          elapsed=0;

cudaEvent_t  start,  stop;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start,

0);

thrust::sort_by_key(thrust::host, pt, pt + n, bt);

int *wt = findWaitingTime(n, bt);

cudaMemcpy(d_bt,          bt,          ARRAY_BYTES,
cudaMemcpyHostToDevice);
cudaMemcpy(d_wt, wt, ARRAY_BYTES,
cudaMemcpyHostToDevice);  findTurnAroundTime<<< 1,  n
>>>(d_bt, d_wt, d_tat); cudaMemcpy(tat, d_tat, ARRAY_BYTES,
cudaMemcpyDeviceToHost);

avg[0] = thrust::reduce(thrust::host, wt, wt + 4,
1); avg[1] = thrust::reduce(thrust::host, tat, tat +
4, 1);

cudaEventRecord(stop,          0);
cudaEventSynchronize          (stop);
cudaEventElapsedTime(&elapsed,  start,
stop);          cudaEventDestroy(start);
cudaEventDestroy(stop);

printf("\nBurst  time  \tWaiting  Time\tTurnaround  Time");
for(int i=0; i<n; i++)

```



```

        printf("\n%d\t\t %d\t\t\t %d",bt[i],wt[i],tat[i]);
    }

    printf("\n\nAverage   Waiting   Time=%f", (float)avg[0]/(float)n);
    printf("\nAverage                               Turnaround
    Time=%f\n", (float)avg[1]/(float)n);

    cudaFree(d_bt);
    cudaFree(d_wt)
;
    cudaFree(d_tat)
;

    printf("The elapsed time in gpu was %f ms", elapsed);

}

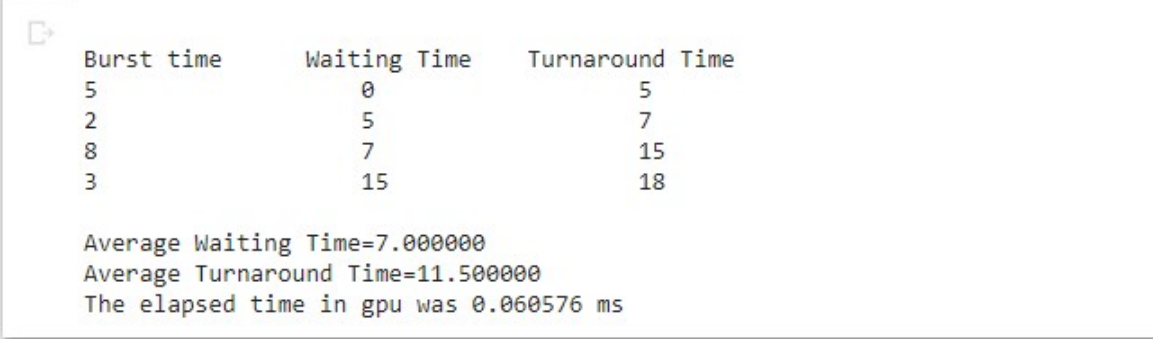
```

Output:

```

    }

```



```

Burst time      Waiting Time      Turnaround Time
5                0                5
2                5                7
8                7                15
3                15               18

Average Waiting Time=7.000000
Average Turnaround Time=11.500000
The elapsed time in gpu was 0.060576 ms

```

The execution time of pbs algorithm in cpu is 0.001 s. The execution time of pbs algorithm in gpu is 0.06ms.

ROUND-ROBIN IN CPU:

Code:

```
#include<stdio.h>

#include<time.h>

void findWaitingTime(int n, int bt[], int wt[], int quantum)
{
    int rem_bt[n];
    for (int i = 0 ; i < n ;
        i++)    rem_bt[i]  =
        bt[i];

    int t = 0;

    while (1)
    {
        int done = 1;
        for (int i = 0 ; i < n; i++)
        {
            if (rem_bt[i] > 0)
            {
                done = 0;

                if (rem_bt[i] > quantum)
                {
                    t    +=    quantum;
                    rem_bt[i]    -=
                    quantum;
                }
            }
        }
    }
}
```

else

```

    {
        t    =    t    +
        rem_bt[i]; wt[i]
        =    t    -    bt[i];
        rem_bt[i] = 0;
    }
}

}

if (done ==
    1) break;
}
}

```

```

void findTurnAroundTime(int n,int bt[], int wt[], int tat[])
{

    for (int i = 0; i < n ;
        i++) tat[i] = bt[i] +
        wt[i];
}

```

```

void findavgTime(int n, int bt[], int quantum)
{
    int wt[n], tat[n], total_wt = 0, total_tat =
    0; int i;

    findWaitingTime(n, bt, wt, quantum);
}

```

```

findTurnAroundTime(n, bt, wt, tat);

printf("\nBurst Time \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i]
    ;
    total_tat+=tat[i];
    total_wt+=wt[i];
    printf("\n%d\t\t %d\t\t %d",bt[i],wt[i],tat[i]);
}

printf("\nAverage waiting time = %f ",(float)total_wt / (float)n);
printf("\nAverage turn around time = %f ",(float)total_tat /
(float)n);
}

```

```

int main()
{

    int n = 4;
    int burst_time[] = {10, 5, 8,3};

    int quantum = 2;
    clock_t
    begin=clock();
    findavgTime(n, burst_time, quantum);

    clock_t end=clock();
}

```

```
printf("\ntime elapsed in cpu is %f seconds",(float)(end-begin)/CLOCKS_PER_SEC);
```



```

    return 0;
}

```

Output:

```

Burst Time      Waiting Time      Turnaround Time
10              16              26
5               13              18
8               16              24
3               12              15
Average waiting time = 14.250000
Average turn around time = 20.750000
time elapsed in cpu is 0.011000 seconds
Process returned 0 (0x0)   execution time : 5.663 s
Press any key to continue.

```

ROUND-ROBIN IN GPU:

Code:

```

%%cu

#include<stdio.h>

>

#include <thrust/reduce.h>
#include <thrust/execution_policy.h>
#include <thrust/sort.h>

void findWaitingTime(int n, int bt[], int quantum, int wt[])
{

    int rem_bt[n];
    for (int i = 0 ; i < n ;
        i++)    rem_bt[i]  =
        bt[i];

```

```
int t = 0;
```

```
while (1)
```

```
{
```

```
    bool done = true;
```

```
    for (int i = 0 ; i < n; i++)
```

```
    {
```

```
        if (rem_bt[i] > 0)
```

```
        {
```

```
            done = false;
```

```
            if (rem_bt[i] > quantum)
```

```
            {
```

```
                t += quantum;
```

```
                rem_bt[i] -=
```

```
                quantum;
```

```
            }
```

```
        else
```

```
        {
```

```
            t = t +
```

```
            rem_bt[i]; wt[i]
```

```
            = t - bt[i];
```

```
            rem_bt[i] = 0;
```

```
        }
```

```
    }
```

```
}
```

```
if (done ==
```

```
    true) break;
```

```

    }
}

```

```

global void findTurnAroundTime(int * d_bt, int * d_wt, int * d_tat)
{
    int idx = threadIdx.x;
    d_tat[idx] = d_bt[idx] + d_wt[idx];
}

```

```

int main()
{
    int *d_bt;
    int
    *d_wt;
    int
    *d_tat;
    const int n = 4;
    const int ARRAY_BYTES = n*sizeof(int);

    cudaMalloc((void **) &d_bt,
    ARRAY_BYTES);    cudaMalloc((void
    **) &d_wt,    ARRAY_BYTES);
    cudaMalloc((void **) &d_tat,
    ARRAY_BYTES);

    int tat[n];
    int bt[n] = {3,2,4,8};
    int  avg[2]  =
    {0,0};    int

```

```
quantum = 2;
```

```
float elapsed=0;
```

```

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,
0);
double
time_spent=0.0; clock_t
begin = clock();

int wt[n];
findWaitingTime(n,bt,quantum,wt
);

cudaMemcpy(d_bt, bt, ARRAY_BYTES,
cudaMemcpyHostToDevice);
cudaMemcpy(d_wt,wt,ARRAY_BYTES,
cudaMemcpyHostToDevice); findTurnAroundTime<<< 1, n
>>>(d_bt, d_wt, d_tat); cudaMemcpy(tat,d_tat,ARRAY_BYTES,
cudaMemcpyDeviceToHost);

avg[0] = thrust::reduce(thrust::host, wt, wt +
n); avg[1] = thrust::reduce(thrust::host, tat, tat
+ n);

printf("\nBurst Time \tWaiting Time\tTurnaround Time");
for(int i=0;i<n;i++)
{
printf("\n%d\t\t %d\t\t\t %d",bt[i],wt[i],tat[i]);
}

printf("\n\nAverage Waiting Time=%f",(float)avg[0]/(float)n);
printf("\nAverage Turnaround

```

```
Time=%f\n", (float)avg[1]/(float)n);
```

```
clock_t end=clock();
```

```

time_spent+=(double)(end-begin)*1000/CLOCKS_PER_SEC;

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed, start,
stop);      cudaEventDestroy(start);
cudaEventDestroy(stop);
elapsed += time_spent;

cudaFree(d_bt);
cudaFree(d_wt)
;
cudaFree(d_tat)
;

printf("The elapsed time in gpu was %f ms", elapsed);

}

```

Output:

```

Burst Time      Waiting Time      Turnaround Time
3               6                 9
2               2                 4
4               7                 11
8               9                 17

Average Waiting Time=6.000000
Average Turnaround Time=10.250000
The elapsed time in gpu was 0.150632 ms

```

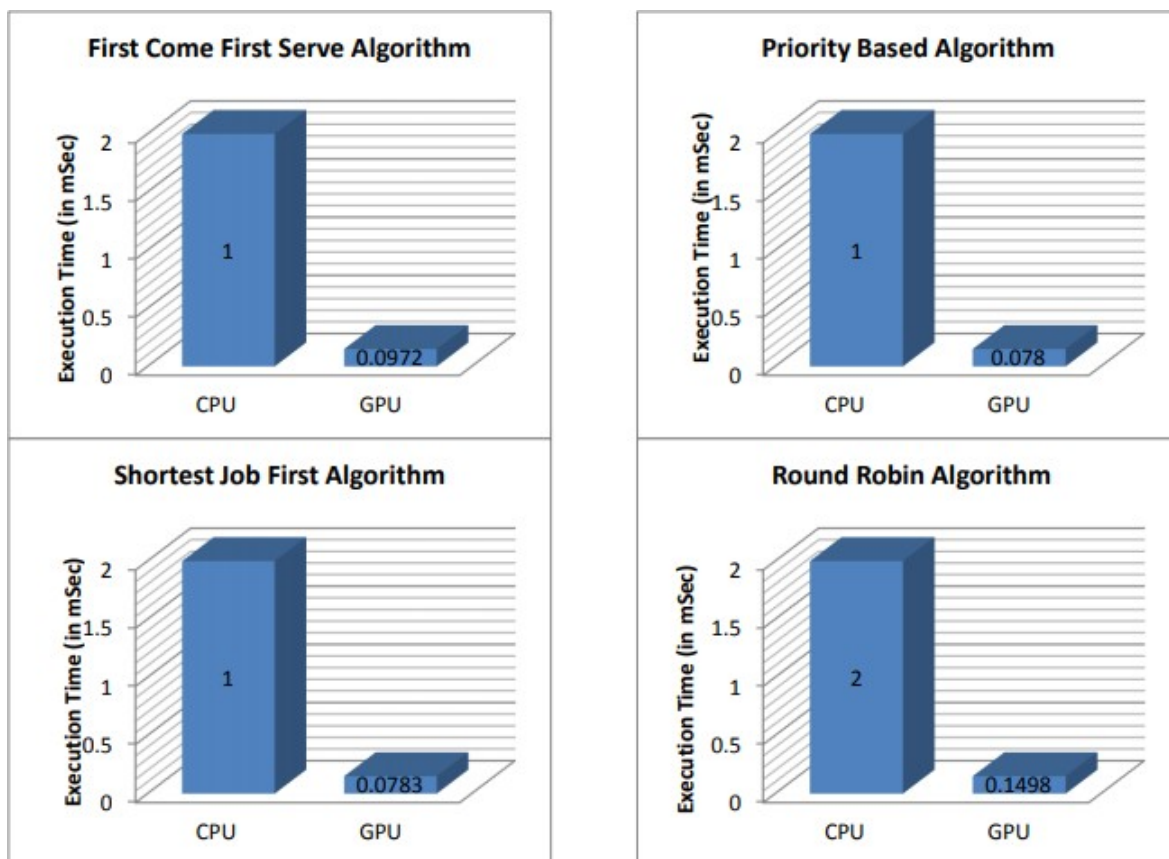
The execution time of rr algorithm in cpu is 0.01 s.

The execution time of rr algorithm in gpu is
0.15ms.

RESULTS AND OBSERVATIONS:

In order to analyze the performance of the implemented algorithms the speedup achieved on the execution with respect to time was evaluated for all the test results. All the tests on the algorithms were performed with the similar number of processing nodes or processors and therefore, the speedup in execution is not evaluated based on the number of processors used but by analyzing the speedup in execution time because of the change in parallelizing approach taken up in the program. In general, the formula to calculate speedup using execution time for same number of processors is given by: $S = T_s/T_p$ Where, T_s is the time it takes for execution of sequential algorithm and T_p is the it times for execution of parallel algorithm.

Comparative graphs of execution time observed (in milliseconds):

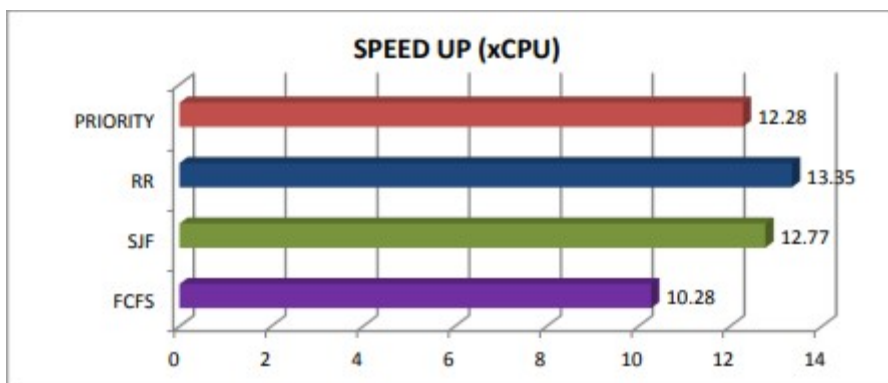


Execution time obtained for various algorithms:

Algorithm Used	CPU Time (in ms)	GPU Time (in ms)
First Come First Serve	1.0	0.0972
Shortest Job First	1.0	0.0783
Priority Based	1.0	0.0780
Round Robin	2.0	0.1498

It contains the data about the execution time as observed for the scheduling algorithms when executed on CPU and GPU respectively. The time complexity is improved to a good extent on GPU due to parallelized approach.

Speedup achieved on various algorithms:



Overall comparison of speed up achieved in different operating system scheduling algorithms as compared with execution time on CPU.

Conclusion

This project introduces a GPU implementation of the operating system scheduling algorithms. The algorithms were designed for the NVIDIA CUDA platform to work in parallel with many threads in execution. It implements the operations of calculating the waiting time and turnaround time on the GPU and it uses the latest features of the NVIDIA CUDA SDK 6.0 on Geforce 740m processor. With the help of the GPU the execution time of various algorithms was found to be 10.28- 13.35x faster than on the CPU using C code. This is a significant enhancement in the performance of the algorithms. GPU has shown tremendous potential and a further performance increase is expected with better optimization and more advanced GPUs.

REFERENCES:

- [1] Shuai C., Michael B., Jiayuan M., David T., Jeremy W. S., Kevin S., “Performance Study of General-Purpose Applications on Graphics Processors Using CUDA”, Journal of parallel and distributed computing, vol 7, no 4, pp.6-25, 2019.
- [2] Maria Andreina F. Rodriguez, “CUDA: Speeding Up Parallel Computing”.
- [3] Wikipedia- “<http://en.wikipedia.org/wiki/CUDA>”
- [4] Anthony Lippert – “NVIDIA GPU Architecture for General Purpose Computing”
- [5] David Kirk/NVIDIA and Wen-mei Hwu, 2006-2008 – “CUDA Threads”
- [6] Yadav K., Mittal A., Ansari M. A., Vishwarup V., “Parallel Implementation of Similarity Measures on GPU Architecture using CUDA”
- [7] Direct Compute Programming Guide
(http://developer.download.NVIDIA.com/compute/DevZone/docs/html/DirectCompute/doc/DirectCompute_Programming_Guide.pdf)
- [8] Singh B.M., Mittal A., Ghosh D., Parallel Implementation of Niblack’s Binarization Approach on CUDA.
- [9] Peter Zalutski “CUDA – Supercomputing for masses.”
- [10] Practical Applications for CUDA
(<http://supercomputingblog.com/cuda/practicalapplicationsfor-cuda/>)
- [11] Matthew Guidry, Charles McClendon, “Parallel Programming with CUDA”.