PROJECT REPORT

Applying MPI and CUDA on Project

Roll no. - CED19I011

Name - Sandeep Ahirwar

Project Description

Problem Statement : **Parallelize Fitness calculation in Genetic algorithm**

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

There are many different complex problem which can be solved through genetic algorithm, we have taken this parallelize fitness calculation for our project.

INPUT GIVEN

- ORGS : 10000
- GENES : 50
- ALLELES : 4
- MUT 1000

Profiling Inference

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.21     14.85    14.85       81     0.18     0.18  Gen
  1.34     15.05     0.20       82     0.00     0.00  Eval
  0.00     15.05     0.00        1     0.00     0.00  Init
  0.00     15.05     0.00        1     0.00     0.00  Mem
  0.00     15.05     0.00        1     0.00    15.05  Run
```

```
Call graph:
```

```
granularity: each sample hit covers 2 byte(s) for 0.07% of 15.05 seconds

index % time    self  children    called     name
                0.00   15.05       1/1           main [2]
[1]    100.0    0.00   15.05       1         Run [1]
               14.85    0.00      81/81          Gen [3]
                0.20    0.00      82/82          Eval [4]
                0.00    0.00       1/1           Init [5]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00   15.05                 main [2]
                0.00   15.05       1/1           Run [1]
                0.00    0.00       1/1           Mem [6]
-----------------------------------------------
               14.85    0.00      81/81          Run [1]
[3]     98.7   14.85    0.00      81         Gen [3]
-----------------------------------------------
                0.20    0.00      82/82          Run [1]
[4]      1.3    0.20    0.00      82         Eval [4]
-----------------------------------------------
                0.00    0.00       1/1           Run [1]
[5]      0.0    0.00    0.00       1         Init [5]
-----------------------------------------------
                0.00    0.00       1/1           main [2]
[6]      0.0    0.00    0.00       1         Mem [6]
-----------------------------------------------

Index by function name

   [4] Eval                    [5] Init                    [1] Run
   [3] Gen                     [6] Mem
```

From this profiling of our project serial code we get to know that 99% of execution time is spent on **Gen()** function

Previously there was a function call **Sel()** which was impossible to parallelize because of its depedencies in **Gen()** so I have removes the **Sel()** function and merged it in Gen() in such a way that it can be parallelized upto some extent.

So the above profiling result is after removing the **Sel()** function.

Here, I have parallelized only 1 main `for` loop in Gen(). This for loop contains 3 more for loops which I was not able to parallize because of the `break` statements.
I tried removing the break statement for the inner for loops but after removing it the execution time was getting doubled.

MPI Project Code

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <time.h>

#define ORGS 10000
#define GENES 50
#define ALLELES 4
#define MUT 1000

#define FROM_MASTER 0
#define FROM_WORKER 1

char **curG, **nextG, *mod;
int *f, totF, Eval(), Sel(), Run();
void Mem(), Init(), Gen();

int main()
{

printf("HERE ");

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    double start,end;
    start = MPI_Wtime();
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    MPI_Status status;
   char **curG, **nextG, *mod;
    int *f, totF;
```

```
// size allocation
curG = (char **)malloc(sizeof(char *) * ORGS);
nextG = (char **)malloc(sizeof(char *) * ORGS);
mod = (char *)malloc(sizeof(char) * GENES);
f = (int *)malloc(sizeof(int) * ORGS);
for (int i = 0; i < ORGS; i++)
{
    curG[i] = (char *)malloc(sizeof(char) * GENES);
    nextG[i] = (char *)malloc(sizeof(char) * GENES);
}

//assigning random values
for (int i = 0; i < ORGS; i++)
{
    for (int j = 0; j < GENES; j++)
    {
        curG[i][j] = rand() % ALLELES;
    }
}
for (int i = 0; i < GENES; i++)
{
    mod[i] = rand() % ALLELES;
}

int gen = 0;
while (++gen)
{
    if (world_rank == 0) // master
    {
        int flag = 0;
        totF = 0;
        int curF = 0;
        for (int i = 0; i < ORGS; ++i)
        {
            for (int j = 0; j < GENES; ++j)
            {
                if (curG[i][j] == mod[j])
                {
                    ++curF;
                }
            }
            if (curF == GENES)
            {
                flag = 1;
```

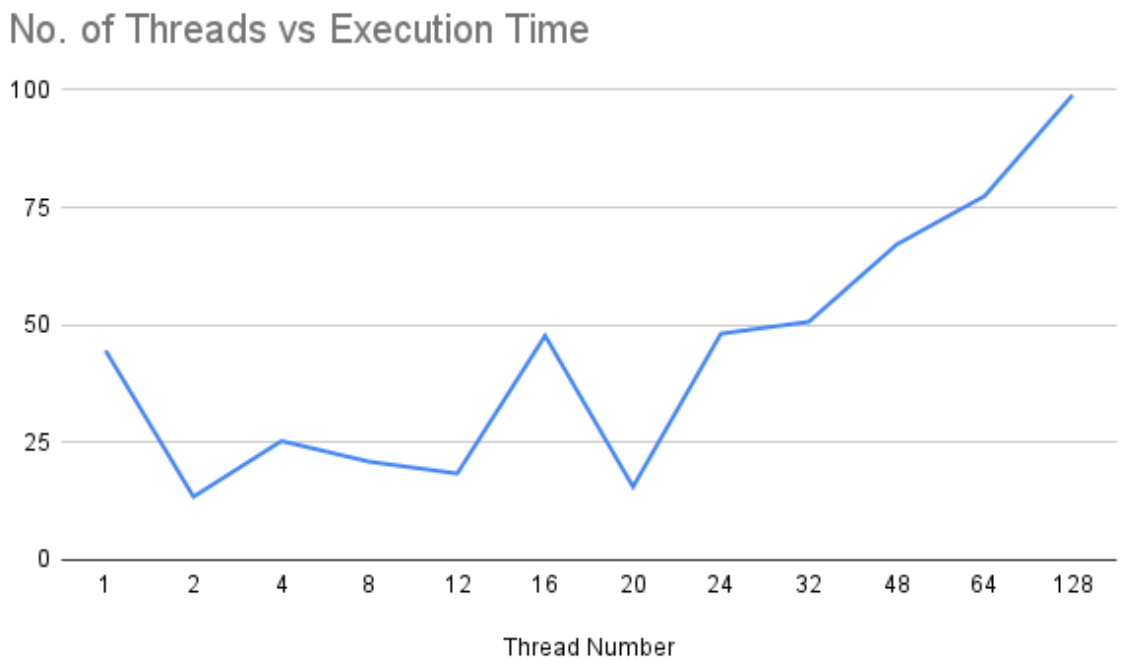
```
                break;
            }
            totF += f[i] = curF;
        }

        printf("%d ", gen);

        if (flag == 1)
        {
            printf("The final generation was: %d\n", gen);
            end = MPI_Wtime();
            printf("\nTotla Time= %f", end-start);
            // Finalize the MPI environment.
            MPI_Finalize();
            return 0;
        }

        int p1, p2;
        int cp = rand() % ALLELES;

        for (int i = 0; i < ORGS; ++i)
        {
            int tot = 0;
            int pt = rand() % (totF + 1);

            // send to worker 1
            MPI_Send(f, ORGS, MPI_INT, 1, FROM_MASTER,
                     MPI_COMM_WORLD);

            MPI_Send(&pt, 1, MPI_INT, 1, FROM_MASTER,
                     MPI_COMM_WORLD);

            MPI_Recv(&p1, 1, MPI_INT, 1, FROM_WORKER,
                     MPI_COMM_WORLD, &status);

            MPI_Send(f, ORGS, MPI_INT, 1, FROM_MASTER,
                     MPI_COMM_WORLD);
            MPI_Send(&pt, 1, MPI_INT, 1, FROM_MASTER,
                     MPI_COMM_WORLD);

            MPI_Recv(&p2, 1, MPI_INT, 1, FROM_WORKER,
                     MPI_COMM_WORLD, &status);

            for (int j = 0; j < GENES; ++j)
```

```
            {
                if (rand() % MUT)
                {
                    if (j < cp)
                    {
                        nextG[i][j] = curG[p1][j];
                    }
                    else
                    {
                        nextG[i][j] = curG[p2][j];
                    }
                }
                else
                {
                    nextG[i][j] = rand() % ALLELES;
                }
            }
        }

        for (int i = 0; i < ORGS; ++i)
        {
            for (int j = 0; j < GENES; ++j)
            {
                curG[i][j] = nextG[i][j];
            }
        }
    }




    else { // worker
    {
        int mpt;
        int *ff;
        ff = (int *)malloc(sizeof(int) * ORGS);
        MPI_Recv(ff, ORGS, MPI_INT, 0, FROM_MASTER,
                 MPI_COMM_WORLD, &status);
        MPI_Recv(&mpt, 1, MPI_INT, 0, FROM_MASTER,
                 MPI_COMM_WORLD, &status);
        int tot = 0;
        int p_1;
        for (int t = 0; t < 10000; ++t)
        {
```

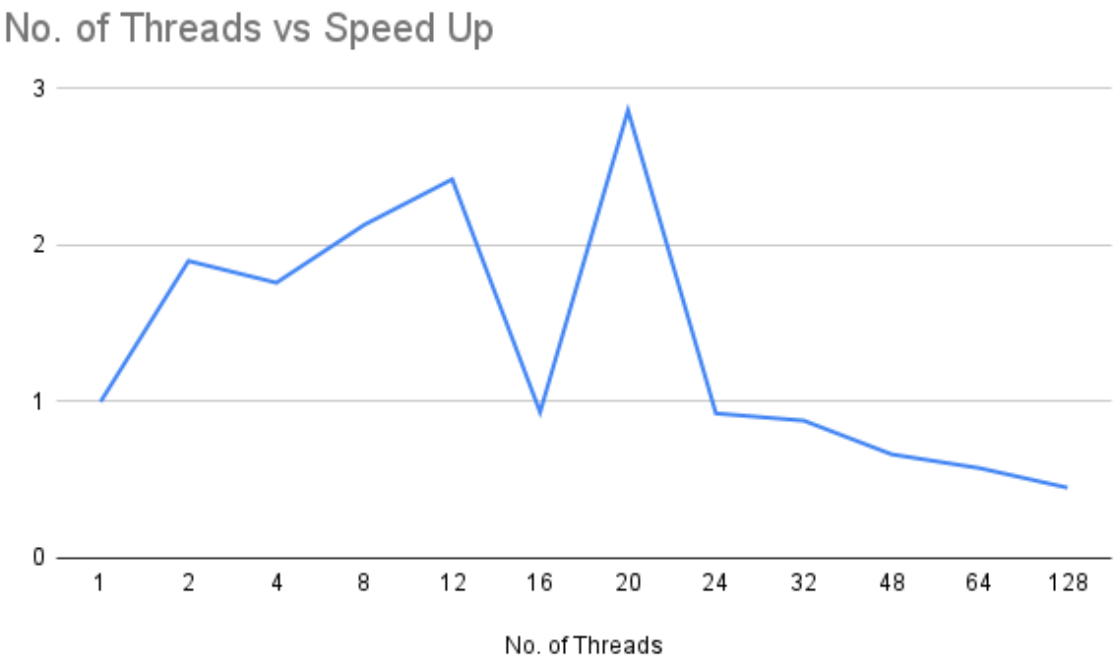
```
                if ((tot += ff[t]) >= mpt)
                {
                    p_1 = t;
                    break;
                }
            }
            MPI_Send(&p_1, 1, MPI_INT, 0, FROM_WORKER, MPI_COMM_WORLD);

        }
        }
    }

}
```

Result for MPI Code

No. of Threads vs Execution Time



No. of Threads vs Speed Up



Parallelization Fraction

Thread Count	Execution Time(s)	Speedup	Speedup(%)	Parallelization Factor
1	44.5464	1	0	--
2	23.4501	1.899625162	89.96251615	0.5262003963
4	25.3046	1.760407199	76.04071987	0.4546837975
8	20.90523	2.13087347	113.087347	0.549009185
12	18.3954	2.421605401	142.1605401	0.6021033902
16	47.7206	0.9334836528	-6.651634724	-0.07271024352
20	15.5632	2.862290532	186.2290532	0.7229216178
24	48.1456	0.9252434283	-7.475657173	-0.08977405831
32	50.6754	0.8790537421	-12.09462579	-0.1528743063
48	67.1343	0.6635415875	-33.64584125	-0.5634050488
64	77.4352	0.5752732607	-42.47267393	-0.8203381443
128	98.9085	0.4503798966	-54.96201034	-1.225328784
			Average Parallelization	**-0.00631929073**

Inference for MPI Code

At any given instance of the program's runtime, there are 4 OS (1st main Ubuntu and 3 Ubuntu 18.04) running, using the common resources from the system, this bottlenecks the program. Also, there are multiple other programs running in background which consume resources hence reducing the

performance.
Due to communication overhead between the systems and because of random generation of generation in program leads to continuous variation in time . Every time run the program, even if if the answer is same the time will be different. because of random variables. Genetic algortihm is all about randomness.

CUDA Code

```
#include <stdio.h>

#define ORGS 10000
#define GENES 50
#define ALLELES 4
#define MUT 1000
#define NOT 500


__device__ int point;

__global__ void add(int *cur,int *next, int *c){
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if(index<NOT){
        c[index] = cur[index] * next[index];
        if(cur[index] > next[index]){
            point = index;
        }
    }
}

char **curG, **nextG, *mod;
int *f, totF, Eval(), Sel(), Run();
void Mem(), Init(), Gen();

int main(){

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

    curG = (char **)malloc(sizeof(char *) * ORGS);
    nextG = (char **)malloc(sizeof(char *) * ORGS);
    mod = (char *)malloc(sizeof(char) * GENES);
```

```
    f = (int *)malloc(sizeof(int) * ORGS);
    for (int i = 0; i < ORGS; i++)
    {
        curG[i] = (char *)malloc(sizeof(char) * GENES);
        nextG[i] = (char *)malloc(sizeof(char) * GENES);
    }

    int *cur, *next, *c;
    int *d_cur, *d_next, *d_c;
    int size = NOT*sizeof(int);
    cudaMalloc((void **)&d_cur,size);
    cudaMalloc((void **)&d_next,size);
    cudaMalloc((void **)&d_c,size);
    cur=(int *)malloc(size);
    next=(int *)malloc(size);
    c=(int *)malloc(size);

    for (int i = 0; i < ORGS; i++)
    {
        for (int j = 0; j < GENES; j++)
        {
            curG[i][j] = rand() % ALLELES;
        }
    }
    for (int i = 0; i < GENES; i++)
    {
        mod[i] = rand() % ALLELES;
    }
    for(int i=0; i<NOT; i++){
        cur[i] = NOT * i;
        next[i] = rand() % ALLELES;
    }

    int gen = 0;
    cudaMemcpy(d_cur, cur, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_next, next, size, cudaMemcpyHostToDevice);

    while (++gen)
    {
        int flag = 0;
        totF = 0;
        int curF = 0;
        for (int i = 0; i < ORGS; ++i)
        {
```

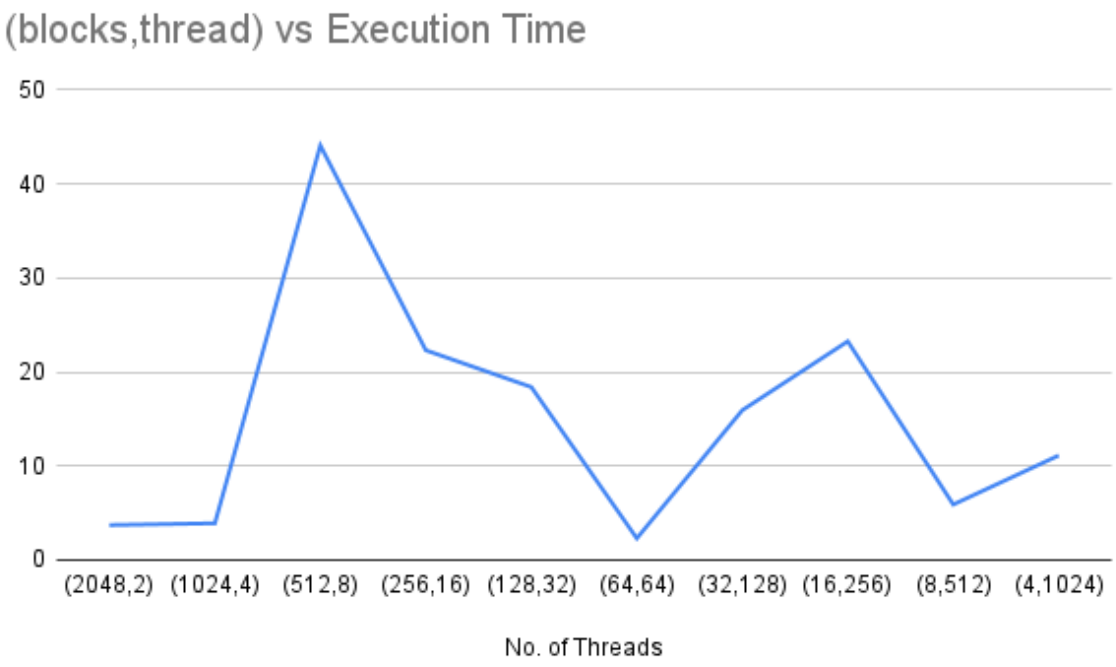
```
            for (int j = 0; j < GENES; ++j)
            {
                if (curG[i][j] == mod[j])
                {
                    ++curF;
                }
            }
            if (curF == GENES)
            {
                flag = 1;
                break;
            }
            totF += f[i] = curF;
        }

        if (flag == 1)
        {
            printf("The final generation was: %d\n", gen);
            break;
        }

        int p1, p2, pt;
        int cp = rand() % ALLELES;

        for (int i = 0; i < ORGS; ++i)
        {
            int tot = 0;
            pt = rand() % (totF + 1);

            add<<<512,8>>>(d_cur,d_next,d_c);
            cudaMemcpy(c,d_c,size,cudaMemcpyDeviceToHost);

            for (int t = 0; t < ORGS; ++t)
            {
                if ((tot += f[t]) >= pt)
                {
                    p1 = t;
                    break;
                }
            }

            for (int t = 0; t < ORGS; ++t)
            {
                if ((tot += f[t]) >= pt)
```

```
            {
                p2 = t;
                break;
            }
        }
        int orgs;
        cudaMemcpyFromSymbol(&orgs, point, sizeof(orgs),
                             0, cudaMemcpyDeviceToHost);

        for (int j = 0; j < GENES; ++j)
        {
            if (rand() % NOT)
            {
                if (j < cp)
                {
                    nextG[i][j] = curG[p1][j];
                }
                else
                {
                    nextG[i][j] = curG[p2][j];
                }
            }
            else
            {
                nextG[i][j] = rand() % ALLELES;
            }
        }
    }

    for (int i = 0; i < ORGS; ++i)
    {
        for (int j = 0; j < GENES; ++j)
        {
            curG[i][j] = nextG[i][j];
        }
    }
}
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_elapsed, start,stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
printf("time is %.5f ms\n", time_elapsed);
free(cur);
```

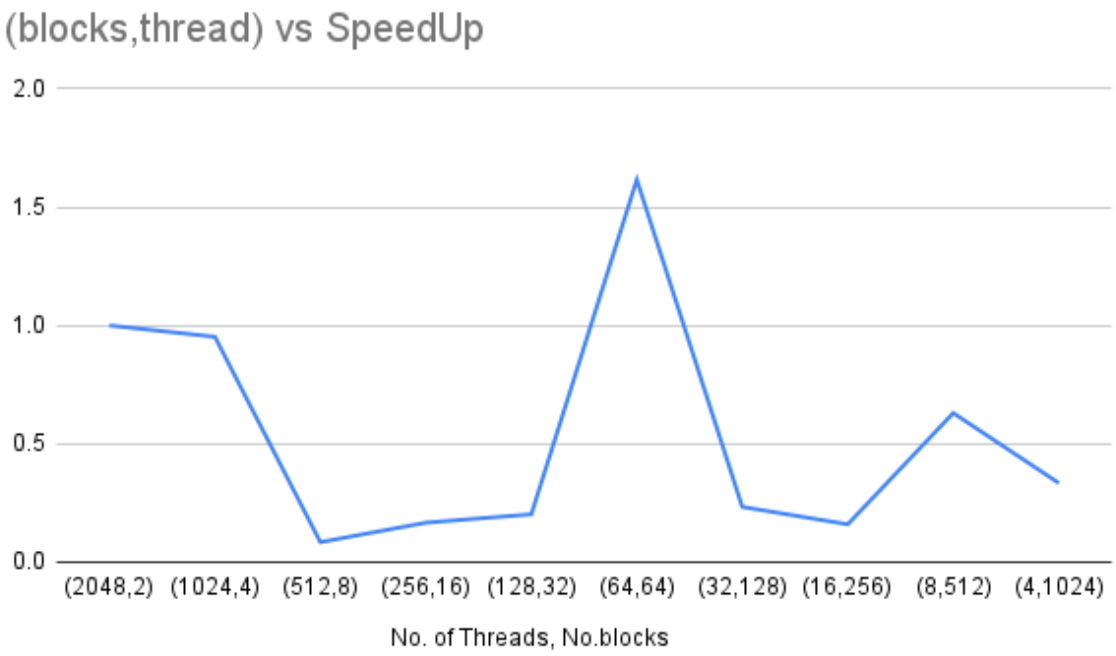
```
    free(next);
    free(c);
    cudaFree(d_cur);
    cudaFree(d_next);
    cudaFree(d_c);
    return 0;
}
```

Result for MPI Code

No. of Block,Threads vs Execution Time



No. of Blocks,Threads vs Speed Up



Threads	Blocks	Execution Time(s)	Speedup	Speedup(%)	Parallelization Factor
2048	2	3.715	1	0	--
1024	4	3.902	0.9520758585	-4.792414147	-0.067115
512	8	44.094	0.08425182565	-91.57481744	-12.42191
256	16	22.309	0.166524721	-83.3475279	-5.338788
128	32	18.3954	0.2019526621	-79.80473379	-4.079128207
64	64	2.299	1.615919965	61.59199652	0.3872075883
32	128	15.932	0.2331785087	-76.68214913	-3.314454065
16	256	23.262	0.1597025191	-84.02974809	-5.282275882
8	512	5.897	0.6299813464	-37.00186536	-0.588497997
4	1024	11.119	0.3341127799	-66.58872201	-1.994949539
2	2048	NA	-	-	-
1	4096	NA	-	-	-
			Average Parallelization	**-3.633324551**	

Inference for Cuda code

The performance of the program is better than openMP and MPI because the least time I got here is 2.299 seconds but in some cases time is rapidly increased the reason behind this is random values in this algorithm. when it gets small random values the the execution time is less and vice versa.

The programs also throw an error “Error: invalid configuration argument” if we increase the number of threads above 1024 as it is mentioned in the critical section of the report above and the documentation of Nvidia CUDA Toolkit. This is cuda limitation that we cannot give no. of threads more than 1024.

Thank You