MPI and CUDA Implementation on RSA ENCRYPTION

High-Performance Computing Project Report

Problem Statement: Encryption and Decryption of Very Large Numbers using the RSA Algorithm

Paul Babu Kadali - CED19I002

Project Description:

We implemented the RSA algorithm to deal with very big numbers and not to use the ready libraries/. This code is capable of Encrypt using public or private keys which is both the encryption and decryption of data. Can take 512 bits each from P and Q.

Profiling of Serial Code:

Tools used for Profiling:

1. Gprof - Function-based Profiling
2. Gconv - Line-based Profiling
3. Likwid - HArdware-Based Profiling

Gprof:

Gprof is used to get the frequency of each function call, to determine the computation-intensive function

```
g++ -fopenmp -pg -g -O0 rsa_serial1.cpp && ./a.out
gprof -b a.out > gprof.out
cat gprof.out
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 76.41     23.76    23.76 12609306     0.00     0.00  LongNumbers::Sub(LongNumbers::BigNum, LongNumbers::BigNum)
 12.54     27.66     3.90 13075598     0.00     0.00  LongNumbers::CopyOf(LongNumbers::BigNum)
  4.28     28.99     1.33     2579     0.00     0.00  LongNumbers::Mul(LongNumbers::BigNum, LongNumbers::BigNum)
  3.38     30.04     1.05   233990     0.00     0.00  LongNumbers::DivSmall(LongNumbers::BigNum, LongNumbers::BigNum)
  1.45     30.49     0.45  1033175     0.00     0.00  LongNumbers::Add(LongNumbers::BigNum, LongNumbers::BigNum)
  0.93     30.79     0.29 28469824     0.00     0.00  LongNumbers::EqualZero(LongNumbers::BigNum)
  0.61     30.98     0.19   478950     0.00     0.00  LongNumbers::AddFront(LongNumbers::BigNum, int)
  0.42     31.11     0.13     1550     0.00     0.02  LongNumbers::DivLarge(LongNumbers::BigNum, LongNumbers::BigNum)
  0.00     31.11     0.00   719334     0.00     0.00  std::remove_reference<int&>::type&& std::move<int&>(int&)
  0.00     31.11     0.00   239778     0.00     0.00  std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<int> >, std::is_move_constructible<int>, std::is_move_assignable<int> >::value, void>::type std
::swap<int>(int&, int&)
  0.00     31.11     0.00   239778     0.00     0.00  void std::iter_swap<int*, int*>(int*, int*)
  0.00     31.11     0.00     6124     0.00     0.00  std::__deque_buf_size(unsigned long)
  0.00     31.11     0.00     4086     0.00     0.00  std::_Deque_iterator<LongNumbers::ArrayOfArray, LongNumbers::ArrayOfArray&, LongNumbers::ArrayOfArray*>::_S_buffer_size()
  0.00     31.11     0.00     3069     0.00     0.00  std::_Deque_iterator<LongNumbers::ArrayOfArray, LongNumbers::ArrayOfArray&, LongNumbers::ArrayOfArray*>::_M_set_node(LongNumbers::ArrayOfArray**)
  0.00     31.11     0.00     3051     0.00     0.00  LongNumbers::ArrayOfArray const& std::forward<LongNumbers::ArrayOfArray const&>(std::remove_reference<LongNumbers::ArrayOfArray const&>::type&)
  0.00     31.11     0.00     2035     0.00     0.00  __gnu_cxx::new_allocator<LongNumbers::ArrayOfArray>::_M_max_size() const
  0.00     31.11     0.00     1557     0.00     0.00  std::iterator_traits<int*>::iterator_category std::__iterator_category<int*>(int* const&)
  0.00     31.11     0.00     1557     0.00     0.00  int* std::end<int, 309ul>(int (&) [309ul])
  0.00     31.11     0.00     1557     0.00     0.00  int* std::begin<int, 309ul>(int (&) [309ul])
  0.00     31.11     0.00     1557     0.00     0.00  void std::reverse<int*>(int*, int*)
  0.00     31.11     0.00     1557     0.00     0.00  void std::__reverse<int*>(int*, int*, std::random_access_iterator_tag)
  0.00     31.11     0.00     1035     0.00     0.00  std::_Deque_base<LongNumbers::ArrayOfArray, std::allocator<LongNumbers::ArrayOfArray> >::_M_get_Tp_allocator() const
  0.00     31.11     0.00     1019     0.00     0.00  std::_Deque_iterator<LongNumbers::ArrayOfArray, LongNumbers::ArrayOfArray&, LongNumbers::ArrayOfArray*>::_Deque_iterator(std::_Deque_iterator<LongNumbers::
ArrayOfArray, LongNumbers::ArrayOfArray&, LongNumbers::ArrayOfArray*> const&)
```

As shown in the outputs, The most called functions are Subtraction and Multiplication of the Long Numbers performed during the Encryption.

Gcov:

Gcov is used to get the frequency of each line, to determine the computation-intensive section

```
g++ --coverage -fopenmp -fprofile-arcs -ftest-coverage -O rsa_serial1.cpp -lgcov
&& ./a.out
gcov rsa_serial1.cpp
cat rsa_serial1.cpp.gcov
```

```
  3320694:  206:    BigNum Sub(BigNum firstOriginal, BigNum second)
        -:  207:    {
        -:  208:        //Op1 - Op2 .. first - second
  3320694:  209:        if(EqualZero(second))
        -:  210:        {
    #####:  211:            return firstOriginal;
        -:  212:        }
  3320694:  213:        if(EqualZero(firstOriginal))
        -:  214:        {
        -:  215:            second.negative = true;
   120459:  216:            return second;
        -:  217:        }
        -:  218:
        -:  219:
  3200235:  220:        BigNum Result, tempResult, first;
  3200235:  221:        first = CopyOf(firstOriginal);
        -:  222:        int val = 0, NextToMe = 0;
        -:  223:        bool LastNum = false;
        -:  224:
        -:  225:
  3200235:  226:        if(second.negative)
        -:  227:        {
        5:  228:            if(first.negative)
        -:  229:            {
        -:  230:                first.negative = false;
        -:  231:                second.negative = false;
    #####:  232:                Result = Sub(second, first);
    #####:  233:                return Result;
        -:  234:            }
        -:  235:            else
        -:  236:            {
        -:  237:                second.negative = false;
        5:  238:                Result = Add(first, second);
        5:  239:                return Result;
        -:  240:            }
        -:  241:        }
        -:  242:        else
        -:  243:        {
  3200230:  244:            if(first.negative)
        -:  245:            {
        -:  246:                first.negative = false;
        -:  247:                second.negative = false;
        4:  248:                Result = Add(first, second);
        -:  249:                Result.negative = true;
        4:  250:                return Result;
        -:  251:            }
        -:  252:        }
        -:  253:
        -:  254:
        -:  255:        int i = 0;
1089634104:  256:         for(i; i < Size2048; i++)
        -:  257:        {
1086550387:  258:             if(LastNum)
        -:  259:                break;
1086433878:  260:             if(first.Num[i] >= second.Num[i])
        -:  261:            {
```

Hardware Profiling with likwid:

```
root@felicity:~# lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  8
  On-line CPU(s) list:   0-7
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
    CPU family:          6
    Model:               142
    Thread(s) per core:  2
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            10
    CPU max MHz:         3400.0000
    CPU min MHz:         400.0000
    BogoMIPS:            3600.00
```

Number of NUMA domains: 1
likwid-pin pins threads to cores so applications don’t migrate over the course of the job execution and loose cache locality.

```
root@felicity:/media/root/e4a48a94-0dd2-4202-a0b8-0622fe5b880a/School/Semester-7/Parallel-Computing/RSA_Paralleli
zation# likwid-topology -G
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
CPU type:       Intel Kabylake processor
CPU stepping:   10
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       2
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
4               1               0               0               *
5               1               1               0               *
6               1               2               0               *
7               1               3               0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------
********************************************************************************
```

```
root@felicity:/media/root/e4a48a94-0dd2-4202-a0b8-0622fe5b880a/School/Semester-7/Parallel-Computing/RSA_Paralleli
zation# likwid-pin -c 0-8 ./a.out
[pthread wrapper]
[pthread wrapper] MAIN -> 0
[pthread wrapper] PIN_MASK: 0->1  1->2  2->3  3->4  4->5  5->6  6->7
[pthread wrapper] SKIP MASK: 0x0
        threadid 140228767884864 -> hwthread 1 - OK
        threadid 140228759492160 -> hwthread 2 - OK
        threadid 140228751099456 -> hwthread 3 - OK
        threadid 140228742706752 -> hwthread 4 - OK
        threadid 140228734314048 -> hwthread 5 - OK
        threadid 140228725921344 -> hwthread 6 - OK
        threadid 140228717528640 -> hwthread 7 - OK
P:9065993558967213406641806966124453707194213598625466315916505874642181142029 3
```

The parallelized code runs on 8 initialized threads. (OMP threads set to 8 in this case)
Likwid-perfctr reports on hardware performance events, such as FLOPS, bandwidth, TLB misses and power.

Profiling Inference:

On application of the mentioned profiling tools, namely, GPROF, GCOV, LIKWID, the hot spot of the program is determined.

Gprof: The most called functions are Subtraction and Multiplication of the Long Numbers performed during the Encryption.

Gconv: The most commonly run lines are the ones within the loop that is Subtracting the Long Numbers and Copyof Long Numbers which can potentially be parallelizable.

Likwid: No inference was drawn from these outputs as the code is not yet parallelized and likwid-perfctr does not support the student’s processor.

MPI code

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<omp.h>
#include<math.h>

#define BOUND 5000
#define PRIME1 74419
#define PRIME2 15731
long long int prime(long long int num,long long int** primearray)
{
   long long int   count, c;
   long long int i=3;
   long long int *tmp = (long long int*) malloc(num*(sizeof(long long int)));
   for ( count = 2 ; count <= num ;  )
   {
       for ( c = 2 ; c <= i - 1 ; c++ ) {
       if ( i%c == 0 )
       break;
       }
       if ( c == i )
       {
       tmp[0]=2;
               tmp[count-1]=i;
       *primearray=tmp;
         count++;
       }
       i++;
}
   return tmp[count-2];
}
       long long int gcd (long long int u , long long int v)
       {

long long  int shift;
long long int diff;
   if (u == 0 || v == 0)
       return u | v;

   for (shift = 0; ((u | v) & 1) == 0; ++shift) {
```

```
        u >>= 1;
        v >>= 1;
    }

    while ((u & 1) == 0)
        u >>= 1;

    do {
        while ((v & 1) == 0)
            v >>= 1;

        if (u < v) {
            v -= u;

        } else {
             diff = u - v;
            u = v;
            v = diff;
        }
        v >>= 1;
    } while (v != 0);

return u<<shift;
}
long long int pollard(long long int *d_primearray,long long int bound, long long int
p, long long int n)
{
long long int i,j,highestPrime;
long long int maxE;
long long int a =2;
long long int x,g,e,c =1;
long long int power,temp;
maxE=0;
// printf("\nEntered Pollard Function\n");
    for(i=0;i<bound/p;i++)
    {
    if(d_primearray[i] > maxE )
        maxE = d_primearray[i];
    }
    bound = maxE;
    x = d_primearray[0];
    for(i=0;x<bound;i++)
        {
```

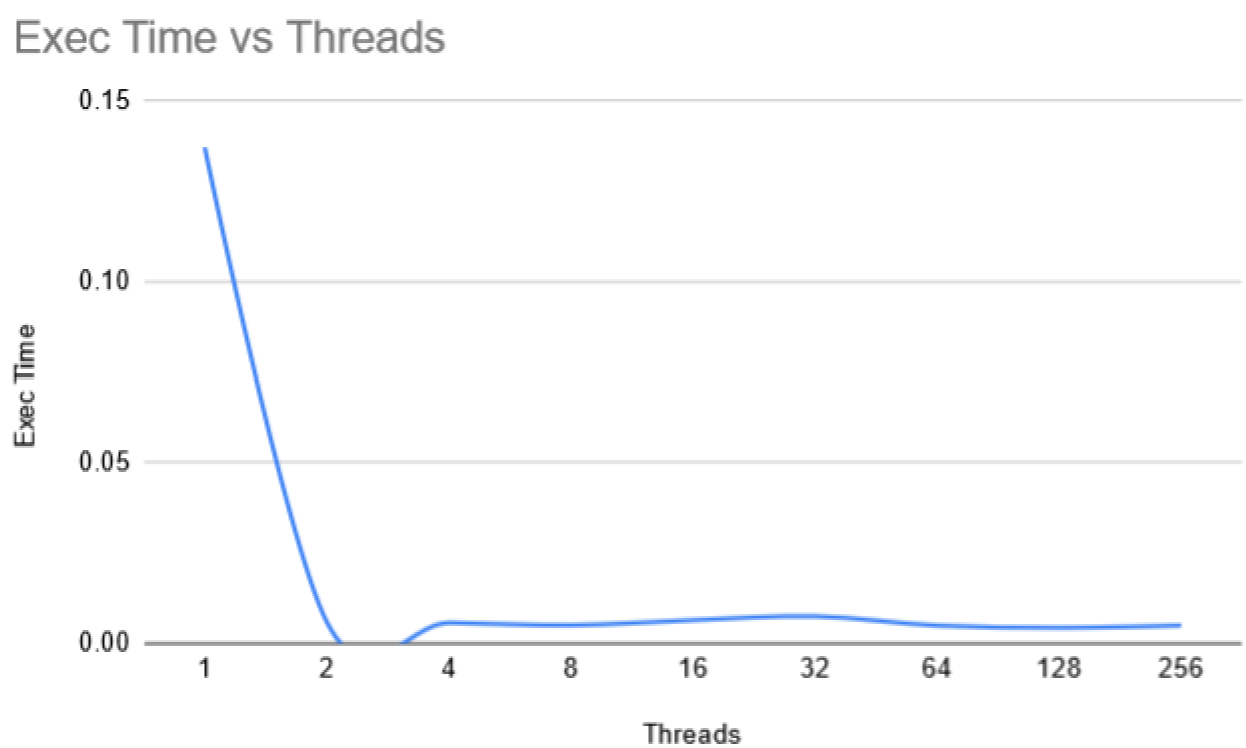
```
        power=(long long int)(log10(bound)/log10(x));
        if(power!=0)
            {
                temp =(long long int)pow(x,power);

            for(j=1;j<=temp;j++)
                {
                  c= (c*2)%n;
                }
                a=c;
         g = gcd(a-1,n);
                if((1 < g) && (g < n))
                {

                        break;
                }
      }

        x = d_primearray[i+1];
    }
return g;
}

int main (int argc, char **argv)
{
   int id, p;

   long long int *h_primearray,*in;
   long long int bound= BOUND;
   long long int n = (PRIME1*PRIME2);
       long long int h_highestPrime , res;
       double start, end;
       MPI_Init(&argc, &argv);
       MPI_Comm_rank(MPI_COMM_WORLD, &id);
       MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Barrier(MPI_COMM_WORLD);
   start=omp_get_wtime();
   if(id==0)
   {
        h_highestPrime = prime(bound, &h_primearray);

   }
```

```
    MPI_Bcast(&bound, 1, MPI_LONG_LONG,0, MPI_COMM_WORLD);
    if( bound%p != 0)
    {
        printf("unevenly distributed \n");
        MPI_Finalize();
        return 0;
    }
     in = (long long int *) malloc((bound/p) *sizeof(long long int));
    MPI_Scatter(h_primearray,(bound/p), MPI_LONG_LONG, in, (bound/p), MPI_LONG_LONG, 0,
MPI_COMM_WORLD);
    res = pollard(in,bound,p,n);
    if( res> 1 && res!=n)
    {
    printf("res= %lld\n",res);
    // end=omp_get_wtime();
    }
        MPI_Finalize();
    end=omp_get_wtime();
    if(id==0)
                printf("Total time taken =%f\n",end-start);
                // printf("Total time start =%f\n",start);
                // printf("Total time end =%f\n",end);
    // free(h_primearray);
    //     free(in);
    fflush(stdout);
    return 0;

}
```

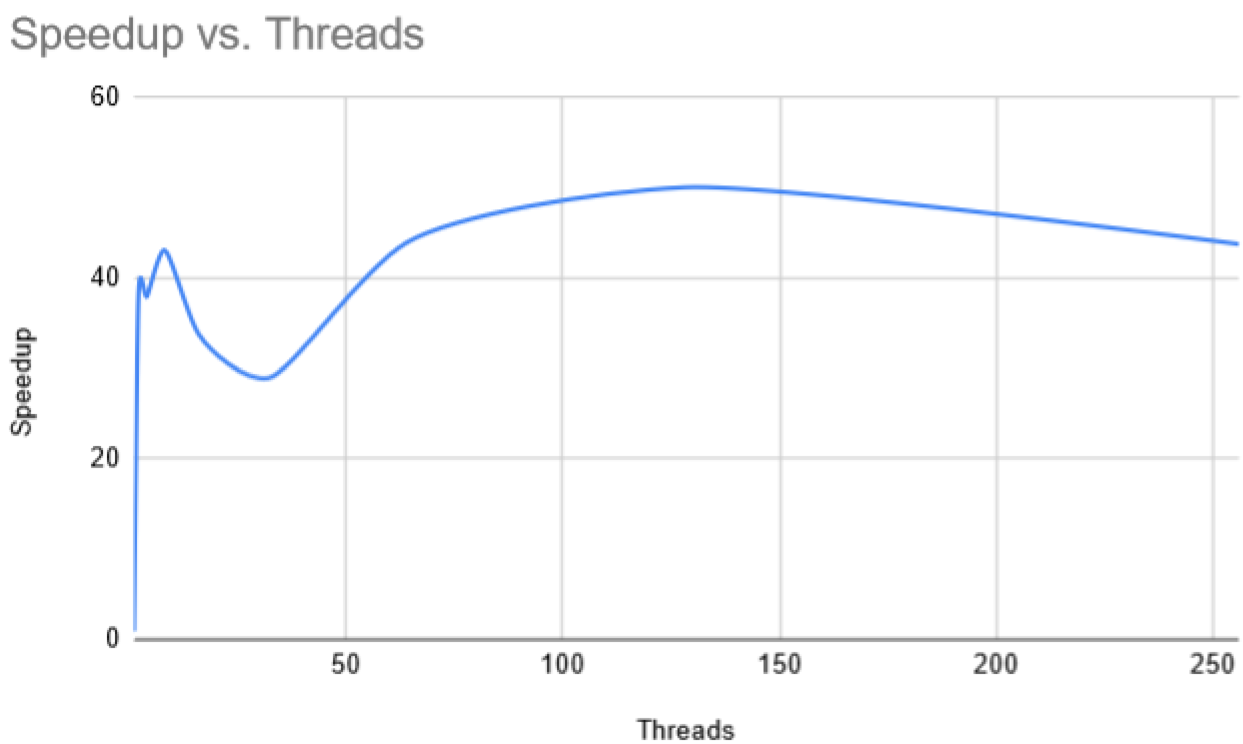
MPI Observations:

no of threads	Exec Time	Speedup	Parallelization Factor
1	0.136962	1	0
2	0.005858	37.25196313	0.8109631476
4	0.005741	38.01114788	0.6491279523
8	0.005062	43.10983801	0.3256011462
16	0.006465	33.75436968	-0.3234580687
32	0.007536	28.95727176	-1.6091106
64	0.004963	43.96977634	-4.234780789
128	0.004358	50.0738871	-9.473618609
256	0.004985	43.77572718	-19.86884457

Execution Time vs Threads



Speedup vs Threads



CUDA code

```
#include "pollardCuda.h"
#include <omp.h>

int Isprime (long long int);


long long int prime(long long int num,long long int** primearray)

{
    long long int   count, c;
    long long int i=3;
    long long int *tmp = (long long int*) malloc(num*(sizeof(long long
int)));

    for ( count = 2 ; count <= num ;  )
    {
```

```
        for ( c = 2 ; c <= i - 1 ; c++ ) {
        if ( i%c == 0 )
        break;
        }
        if ( c == i )
        {

            tmp[0]=2;
            tmp[count-1]=i;

            *primearray=tmp;

            count++;
        }
        i++;
    }
        //printf("abcd %ld",tmp[8]);
    return tmp[count-2];
}


int IsPrime(long long int num)
{
  long long int j;
  long long int k;
  k = sqrt(num);

  for (j=2;j<=k;j++)
  {
    //printf("\nHere");
    if(num%j==0)
    return 0;
  }
  return 1;
}

int main()
```

```
{
  long long int *d_primearray;
  long long int *h_primearray;
  long long int bound=75;
  long long int h_highestPrime;
  long long int *res;

  long long int other_factor,after_e;
  long long int p, q, n;
  int flag;

  long long int t;


  printf("Enter the prime number\n");
  scanf("%lld",&p);

  flag = IsPrime(p);
  if( flag == 0)
  {
    printf("wrong input");
    exit(0);
  }
  printf("Enter another prime number\n");
  scanf("%lld",&q);
  flag = IsPrime(q);
  if(flag ==0 || p==q)
  {
    printf("wrong input");
    exit(0);
  }

  n= p*q;
  t = (p-1)*(q-1);

  /**************************************************/
```

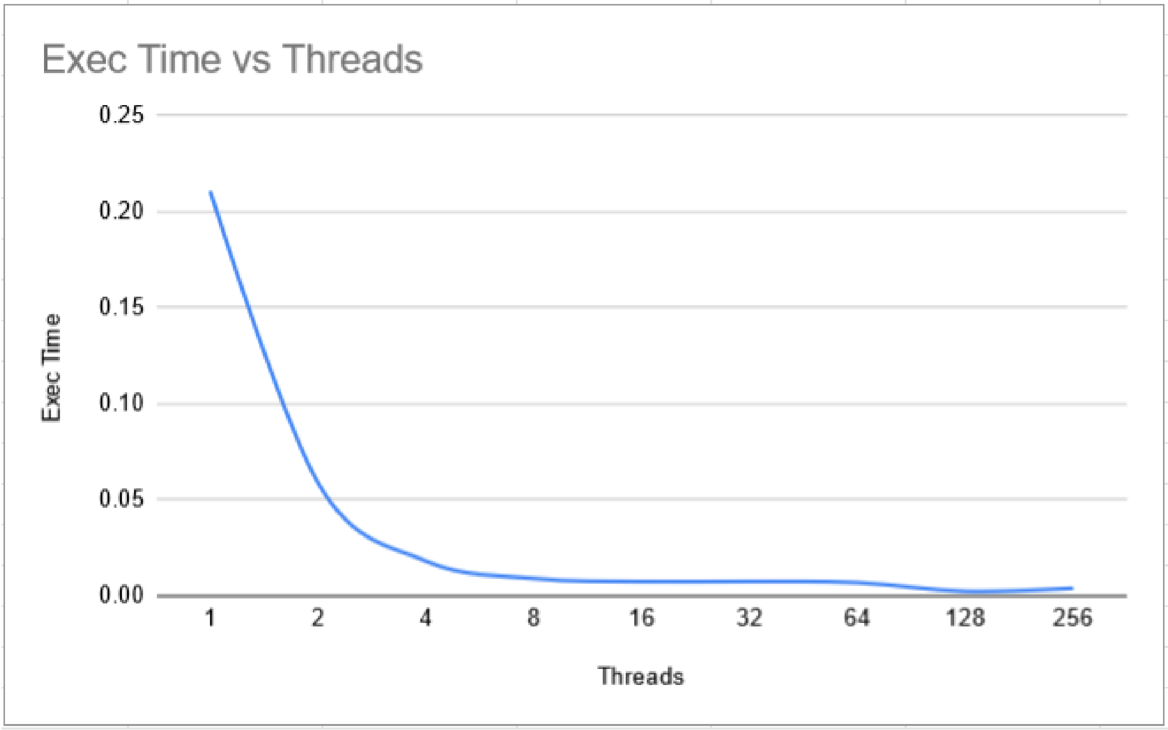
```
  h_highestPrime = prime(bound, &h_primearray);

  cudaEvent_t start,stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);
    cudaMalloc((void**)&d_primearray,bound*sizeof(long long int));
    cudaMalloc((void**)&res,sizeof(long long int));
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float time_cmalloc = 0;
    cudaEventElapsedTime(&time_cmalloc,start,stop);
  cudaEventRecord(start);
    cudaMemcpy(d_primearray,h_primearray,bound*sizeof(long long
int),cudaMemcpyHostToDevice);
  cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float time_htod=0;
  double NUM_BLOCKS;
  NUM_BLOCKS = ceil((double)bound /NUM_THREADS);

    cudaEventElapsedTime(&time_htod,start,stop);

  cudaEventRecord(start);


pollard_gpu<<<(int)NUM_BLOCKS,NUM_THREADS>>>(d_primearray,h_highestPrime,bo
und,res, n);
  cudaError_t err = cudaGetLastError();
  if (err != cudaSuccess) {
        printf("Error: %s\n", cudaGetErrorString(err));
  }
  cudaDeviceSynchronize();
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
```

```
    float time_kernel = 0;
    cudaEventElapsedTime(&time_kernel,start,stop);
  long long int *gpu_res;
  gpu_res = (long long int*) malloc(sizeof(long long int));
  cudaEventRecord(start);
    cudaMemcpy(gpu_res,res,sizeof(long long int),cudaMemcpyDeviceToHost);
  printf("Factor = %lld\n",*gpu_res);


    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float time_dtoh = 0;
    cudaEventElapsedTime(&time_dtoh,start,stop);
  cudaFree(d_primearray);
  cudaFree(res);
  other_factor = n/ (*gpu_res);
  after_e = ((*gpu_res)-1)*(other_factor-1);

double total_time = time_cmalloc+time_dtoh+time_htod+time_kernel;
printf("Total time - %f\n", total_time/300);
    return 0;
}
```

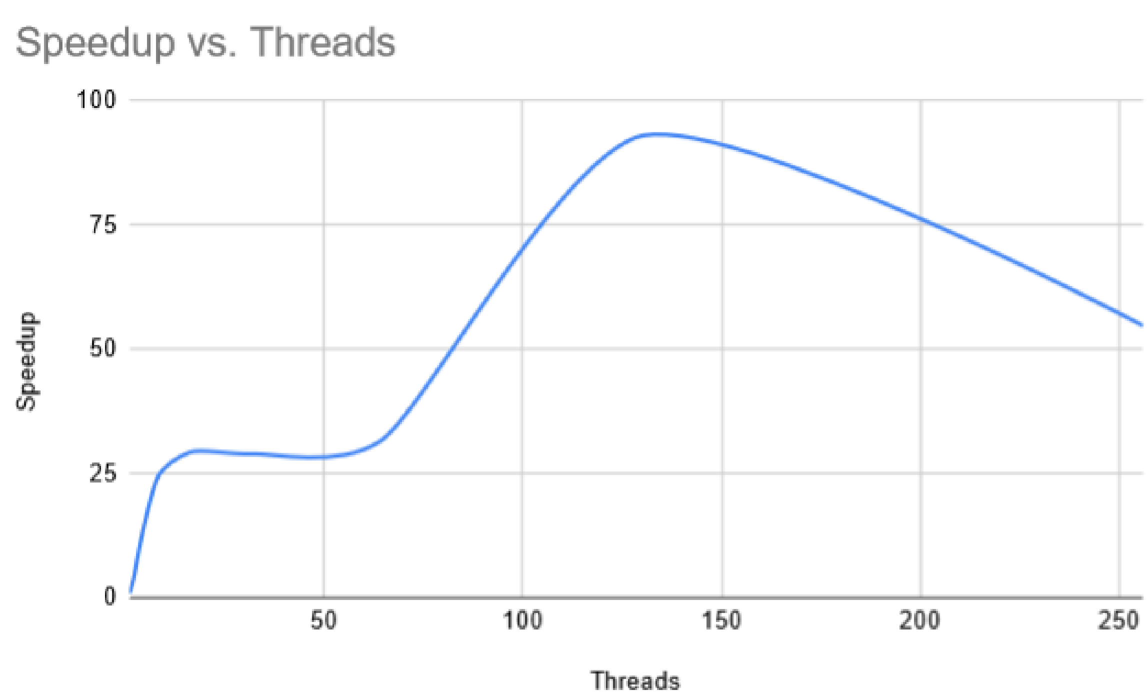
CUDA Observations:

no of threads	Exec Time	Speedup	Parallelization Factor
1	0.210438	1	0
2	0.058584	3.724941964	0.6096162012
4	0.018091	12.062462	0.6113987896
8	0.009062	24.08099757	0.3194911604
16	0.007465	29.23268587	-0.3219305722
32	0.007536	28.95727176	-1.6091106
64	0.006936	31.46222607	-4.195602032
128	0.002358	92.54537744	-9.562213403
256	0.003985	54.7608532	-19.96202186

Execution Time vs Threads



Speedup vs Threads



Inference:

From the above observations, we can see that the OpenMP code scales well with increase in threads, giving an effective 5x speedup with 12 threads over the serial code. The MPI code sees little increase in performance, as there will be communication overhead among thecluster even though its comparable to as that of OpenMP it is still less. The cost of communication outweighs the speedup benefit from parallelization, giving only an effective 1.3x speedup over the serial code.

The CUDA code is able to take the massive number of threads and with little changes to the algorithm, able to provide an almost ~15x speedup over the serial code. We can also see the trend in bandwidth requirement: the MPI code is bottlenecked by the bandwidth, OpenMP can utilize all the memory channels available in the CPU and the GPU is much faster considering its large bandwidth advantage over the CPU.