Factorization of a Large Number

Author: Morgan Cooper
Date: Fall 2009
Class: CSE710

Outline

- Goal
- Problem Set
- Expected Values
  - Input; Output;
- Plan of Attack
- Solution (Executed on Device)
- Distributed Solution (Multiple Devices)
- Results
- Conclusion

Goal

- To find the optimal amount of computation a thread should compute when using CUDA devices
- Idea (Steps)
  - 1: Implement an easy solution that requires minimal computation
    - The problem must be scalable relatively easy with respect to threads
  - 2: Run tests for a set number of threads with varying input size
  - 3: Change thread count and repeat step 2 until complete range of thread domain is exhausted
  - 5: Analyze results for conclusion

Problem Set

- Factoring a number
  - Simple task but can get very time consuming once the number becomes very large
  - If the number only contains two factors then you can assume it is the product of two prime numbers
- For cryptography it's essential to use a one-way, or trapdoor mathematical function.
  - A mathematical function that's easy to do in one direction but very difficult, or impossible to reverse.
- Factoring of prime numbers
  - Easy to find product of two large prime numbers.
  - Difficult to factor large product to two prime numbers.
  - Very large prime number used, because larger the prime number, the more difficult factoring becomes.

Expected Values

- Input
  - Very large integer
- Output
  - Array of values which are factors of the input

Plan of Attack

- Brute force solution
  - Iterate over all values in range
  - Do Modulus operation to test for factor
  - Remember proper factors
  - Return results
- Data Structure Storage (Results)
  - Array of integers that hold the results
  - Size of array is sqrt(“input number”) *2
    - Array multiplied by two to hold pair of the factor
  - Each successful modulus operation sets appropriate location in array to integer value found
    - Also sets adjacent value of array which is offest by sizeCount(number of iterations required per device)

Array Data Structure

- In this example assuming a device had to compute Factors for the number 20
  - sizeCount would be sqrt(20) +1 which is 5
  - factor found at Index = 1 would mean it needs to set its adjacent factor at (Index + sizeCount) which would be position 6 in the array.
    - Ex. Array[1] * Array[6] = 20 or 10*2 = 20

Index	0	1	2	3	4	5	6	7	8	9
Value	1	2	0	4	5	20	10	0	5	4

Solution (Executed on Device)

- Input number N
- Each Iteration
  - Perform Modulus operation for each index on N looking for resulting 0 value and set according Factor position
  - Device's starting number is set according to starting location passed in plus it's own threadId.x
    - Ex. tempNumber%N = start + threadId.x
  - If ((tempNumber%N) == 0) Array[Index] = tempNumber%N
    Array[Index+sizeCount] = N/tempNumber

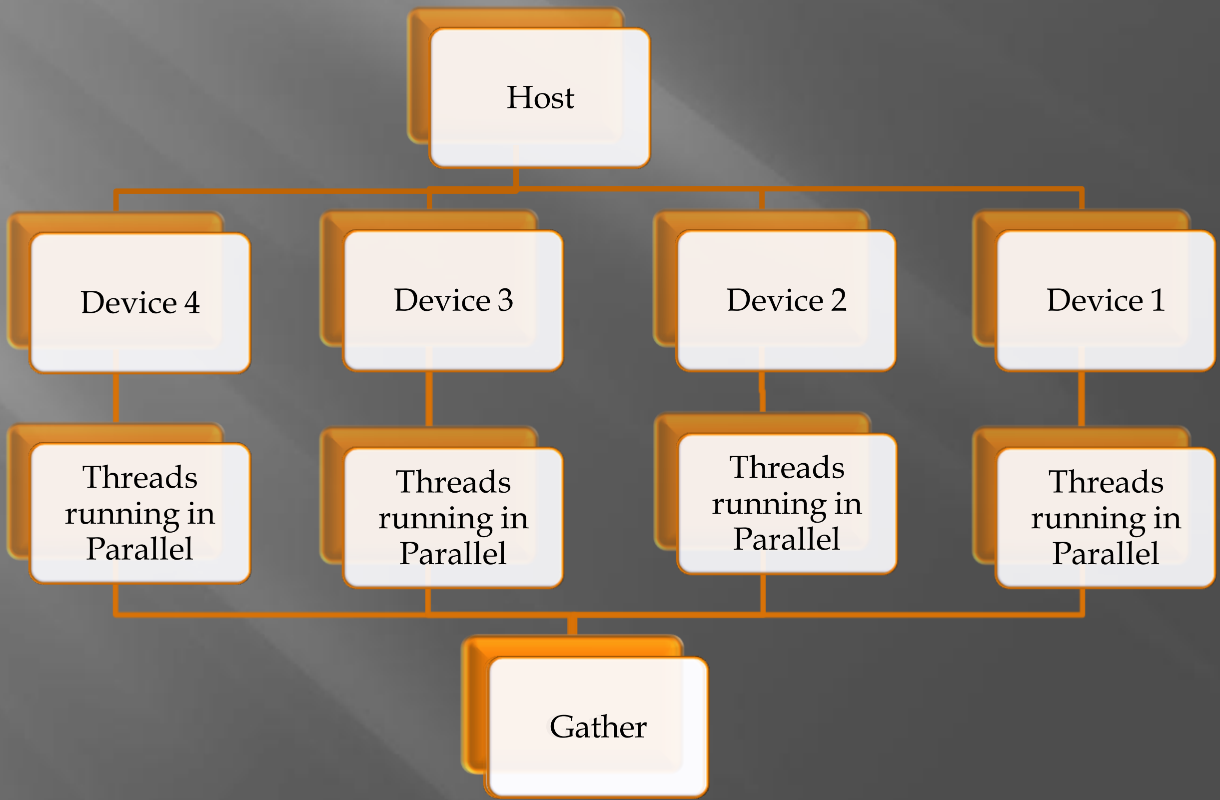
Index	0	1	2	…	…	…	Index+ SizeCount	Index+ SizeCount
Operation	%N	%N	%N	…	…	…	/N	/N
Array	0	0	0	…	…	…	0	0

Solution (Executed on Device)

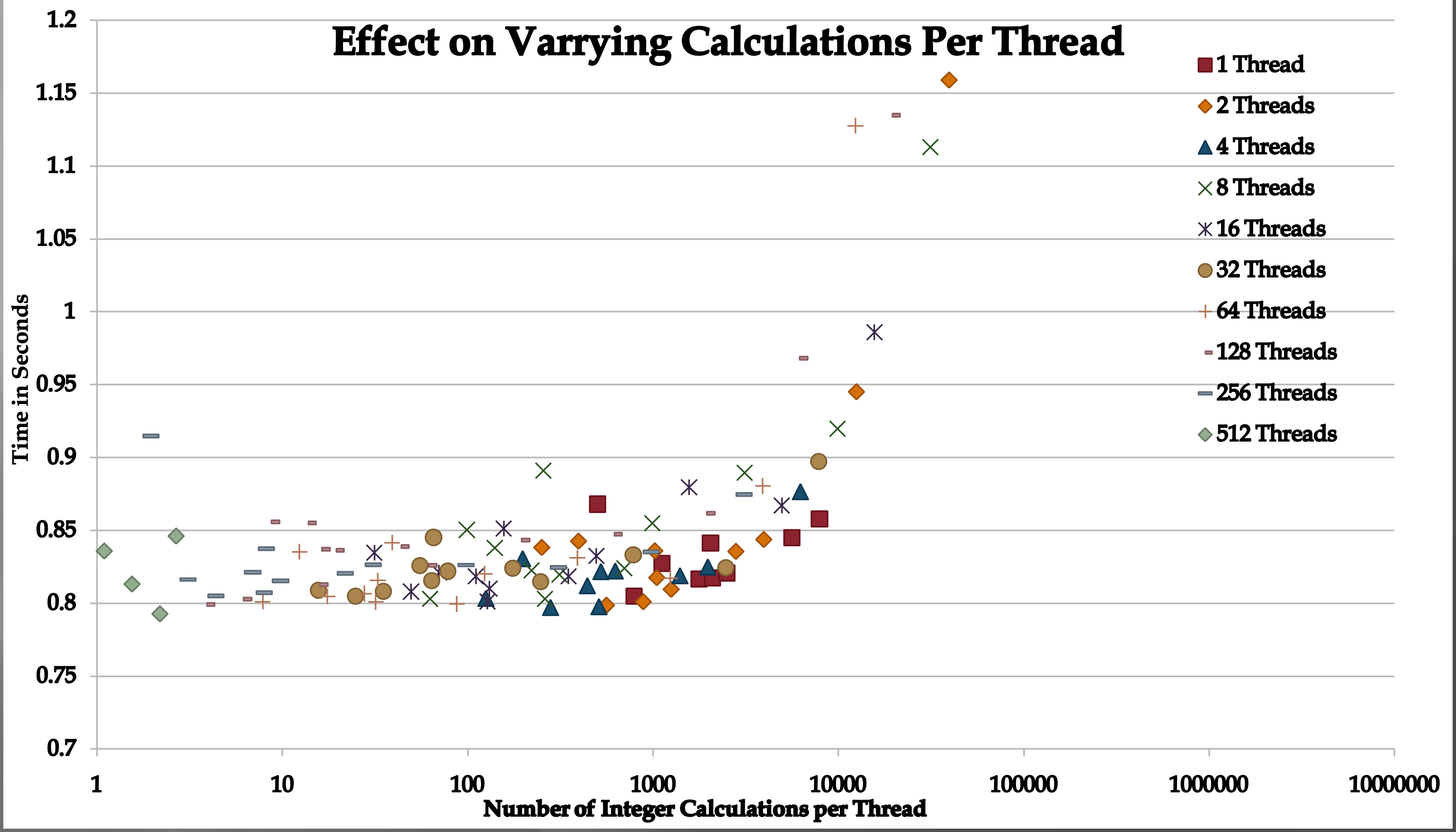
```
__global__ static void kernel(int num,int start,int sizeCount,int numThreads,int
  *value)
 {
   int tx = threadIdx.x;
   int insertPosition = tx;

   int tempNum = tx + start;

   while( insertPosition < sizeCount ){
    if((num%tempNum) == 0){
           value[insertPosition] = tempNum;
           value[insertPosition+sizeCount] = num/tempNum;
    }
    insertPosition = insertPosition + numThreads;
    tempNum = tempNum + numThreads;
   }

 }
```

Distributed Solution (Multiple Devices)

- Host cudaMalloc's on all CUDA devices
- Split Array by Device then between each process
  - Amount of Iterations = (sqrt(NUM) +1)/devCount

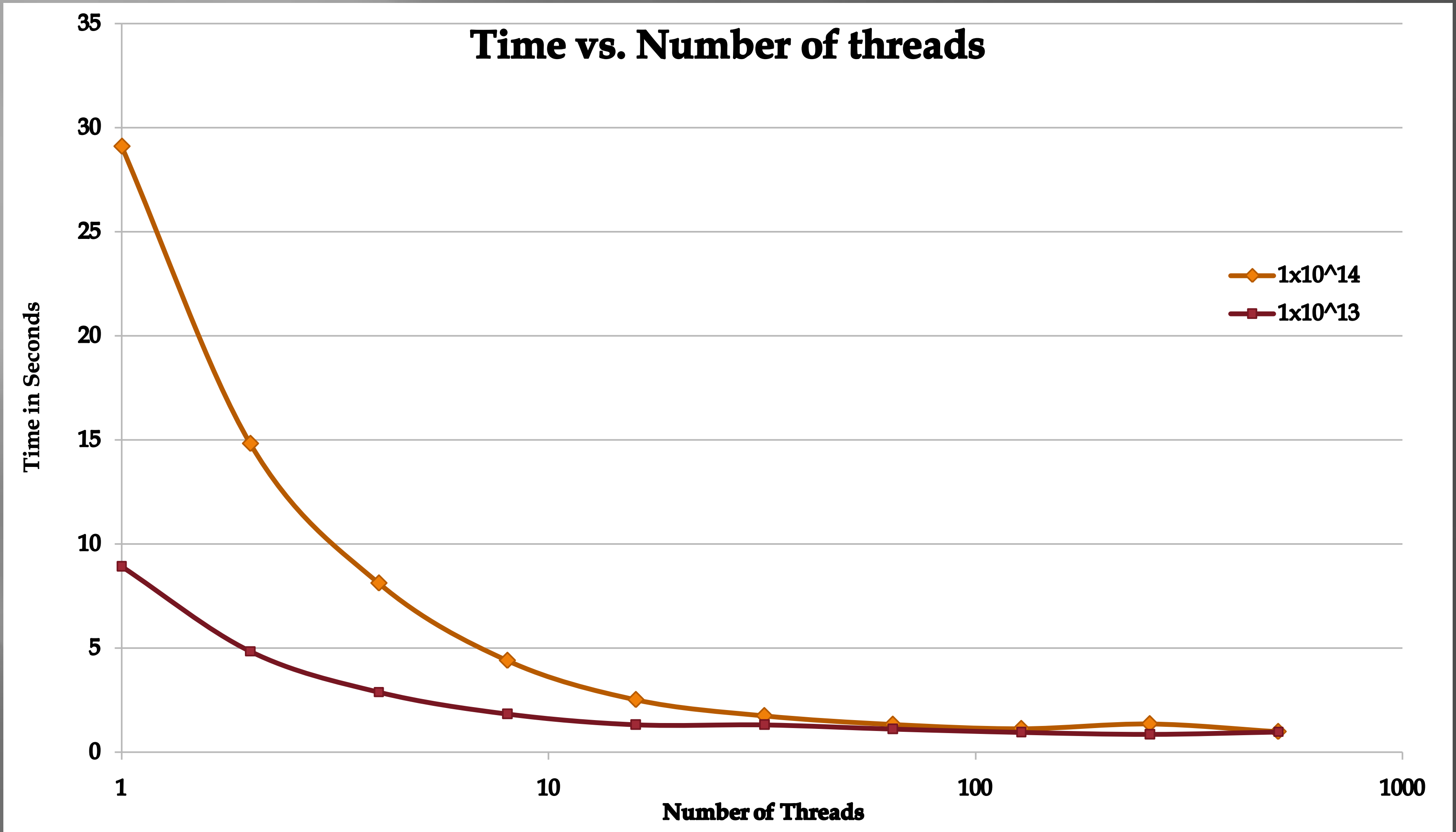


Results



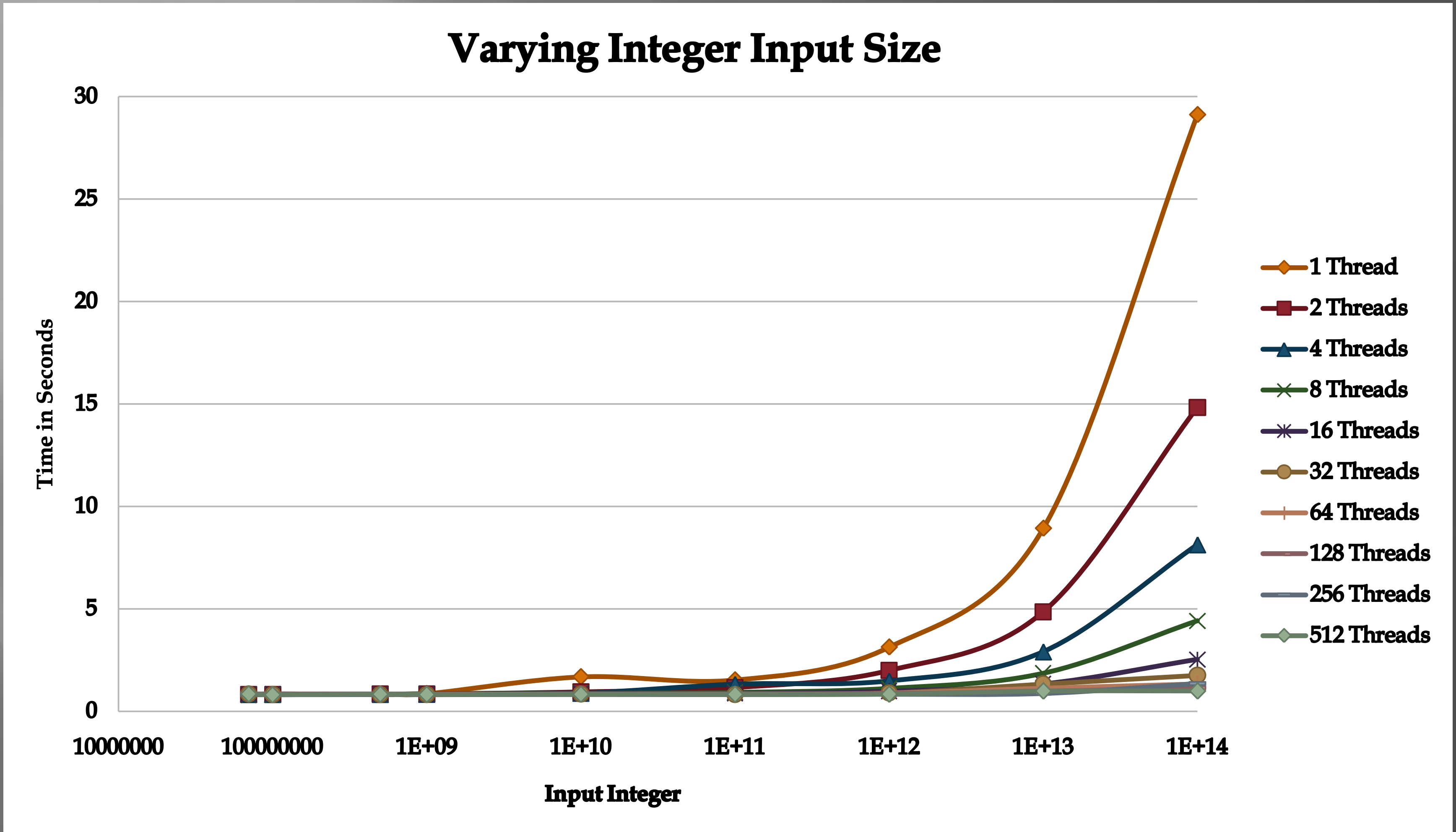
Effect on Varrying Calculations Per Thread
1.2
1.15
1.1
1.05
1
0.95
0.9
0.85
0.8
0.75
0.7
Time in Seconds
1
10
100
1000
10000
100000
1000000
10000000
Number of Integer Calculations per Thread
1 Thread
2 Threads
4 Threads
8 Threads
16 Threads
32 Threads
64 Threads
128 Threads
256 Threads
512 Threads

Results



Time vs. Number of threads
35
30
25
20
15
10
5
0
1
10
100
1000
Number of Threads
Time in Seconds
1x10^14
1x10^13

Results



Varying Integer Input Size
30
25
20
15
10
5
0
Time in Seconds
10000000
100000000
1E+09
1E+10
1E+11
1E+12
1E+13
1E+14
Input Integer
1 Thread
2 Threads
4 Threads
8 Threads
16 Threads
32 Threads
64 Threads
128 Threads
256 Threads
512 Threads

Conclusion

- Thread creation is very minimal
- Most time spent in device initialization
- Threads computing up to 1000 computations seems optimal with including device initialization for timing

- Further research
  - Time kernel execution to obtain direct relation to spawning threads with excluding device time