Goal
Problem Set
Expected Values
  - Input; Output;
Plan of Attack
Solution (Executed on Device)
Distributed Solution (Multiple Devices)
Results
Conclusion
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
4. Analyze results for conclusion
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 $\sqrt{\text{input number}} \times 2$
    - 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 offset by $\text{sizeCount(number of iterations required per device)}$
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.

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>20</td>
<td>10</td>
<td>0</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Index</th>
<th>Operation</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>%N</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>%N</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>%N</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Index+SizeCount</td>
<td>/N</td>
<td>0</td>
</tr>
<tr>
<td>Index+SizeCount</td>
<td>/N</td>
<td>0</td>
</tr>
</tbody>
</table>
__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;
    }
}
- Host `cudaMalloc`'s on all CUDA devices
- Split Array by Device then between each process
  - Amount of Iterations = \( \frac{\sqrt{\text{NUM}} + 1}{\text{devCount}} \)
Results

Effect on Varying Calculations Per Thread

Time in Seconds

Number of Integer Calculations per Thread
Results

Varying Integer Input Size

Time in Seconds

Input Integer

100000000 1000000000 1E+09 1E+10 1E+11 1E+12 1E+13 1E+14

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