1 Primality

Definition 1.1. a divides b if and only if $\exists n \in \mathbb{N}$ such that $b = na$. We write $a \mid b$.

Definition 1.2. $P \in \mathbb{N}$ is prime if and only if $\forall a \in \mathbb{N}$ with $n \neq 1, P$ then $a \nmid P$.

Proposition 1.1. *$P \in \mathbb{N}$ is prime if $\forall a \in \mathbb{N}$ such that $1 < a \geq \sqrt{P}$ then $a \nmid P$.*

Proof. By way of contradiction we suppose that both P is not prime and $\forall a \in \mathbb{N}$ st $a < \sqrt{N}$ then $a \nmid P$. Since P is not prime we have some $b \nmid P$ so $\exists n$ such that $bn = P$. Either, b or n are less than $\sqrt{P}$ so we can let $a = b$ and we have a contradiction. Or then must both be greater than $\sqrt{P}$. In this case we have $bn > \sqrt{P}n > \sqrt{P}\sqrt{P} = P$ so $bn > P$, this means $bn \neq P$ which is a contradiction. Therefore, P must be prime. □

Definition 1.3. For $a, b \in \mathbb{Z}$, we say $a \equiv b \mod n$ if and only if $a - b \mid n$.

Corollary 1.1.1. *P is prime if $\forall a \in \mathbb{N}$ such that $1 < a \geq \sqrt{P}$ implies $P \not\equiv 0 \mod a$.*

Proof. We suppose $\forall a \in \mathbb{N}$ such that $1 < a \leq \sqrt{P}$ then $P \not\equiv 0 \mod a$. Then by the definition of modular arithmetic $a - 0 = a \nmid P$. Therefore by proposition 1.1 P is prime. □

Algorithm 1 Divisibility Test

```
1:  function PRIME(n)
2:      if n ≤ 1 then
3:          return FALSE                    ▷ 1 and numbers ≤ 1 are not prime
4:      end if
5:      for i ∈ ℕ, 1 < i < √n do
6:          if n ≡ 0 mod i then
7:              return FALSE                ▷ n is divisible by i and therefore not prime
8:          end if
9:      end for
10:     return TRUE
11: end function
```

Remark. For each n the algorithm tests divisibility on $\sqrt{n}$ numbers hence this is an O(n) time-complexity algorithm.

Definition 1.4. A composite number is a number $a \in \mathbb{N}$ that is not prime.

Proposition 1.2. *Composite numbers can be factored into primes.*

Proof. Base case: For $k = 4$ and primes $p_1 = 2$, $p_2 = 2$. It is clear $p_1 p_2 = k$. Inductive case: Suppose composite numbers less that k can be factored into primes. Consider $k + 1$ is not prime so there exists $a \in \mathbb{N}$ such that $a \mid k + 1$ so

Algorithm 2 Divisibility Test Sieve

```
1: function PRIME-SIEVE(n):
2:     Array<bool> prime[n]
3:     for i ∈ {0, ..., n} do
4:         prime[i] ← Prime(i)
5:     end for
6:     return primes
7: end function
```

$k+1 = an$. since $a \geq 1$ clearly both $a < k$ and $n < k$ so by the hypothesis a and n can be factored into primes $a = p_1 p_2 ... p_m$ and $n = q_1 q_2 ... q_w$. Therefore, $k+1 = p_1 p_2 ... p_m q_1 q_2 ... q_w$ can be factored into primes. □

Remark. So we see that we need only check the previous prime numbers to see if a number is prime or composite.

Algorithm 3 Sieve Of Eratosthenes

```
1: function ERATOSTHENES(n):
2:     Array<bool> prime[n] = True
3:     prime[0] ← False
4:     prime[1] ← False
5:     for i ∈ {2, 3, ..., n} do
6:         for p ∈ {2, 3, ..., √i} do
7:             if p = True then                ▷ Boolean logic omitted for clarity.
8:                 if i ≡ 0 mod p then
9:                     prime[i] ← False        ▷ i is not prime.
10:                end if
11:            end if
12:        end for
13:    end for
14:    return prime
15: end function
```

Remark. This is $O(n \log(\log(n))$ time complexity. This comes from a result first conjectured by Fermat that the density of prime numbers approach $\frac{1}{\log(n)}$ which evaluated in a series results in this complexity.

2 Linear Sieves

The sieve of Eratosthenes remained the fastest way to find primes numbers for over 2000 years until Sundaram found his sieve. Instead we'll be considering the Sieve of Atkin.

Proposition 2.1. *$P \in \mathbb{N}$ is prime or has a perfect square factor if the following conditions hold:*

- *$4x^2+y^2 = p$ has an odd number of solutions when $p \equiv 1 \mod 12$ or $z \equiv 5 \mod 12$.*
- *$3x^2 + y^2 = p$ has an odd number of solutions when $p \equiv 7 \mod 12$.*
- *$P \not\equiv 3 \mod 4$ and $3x^2 - y^2 = p$ has an odd number of solutions when $z \equiv 11 \mod 12$.*

Proof. Omitted for brevity. □

Algorithm 4 Sieve of Atkin

```
function ATKINS(n)
    Array<bool> prime[n] = False
    for x ∈ {1, 2, .., √n} do
        for y ∈ {1, 2, .., √n} do
            z ← 4x² + y²
            if z ≤ n & (z ≡ 1 mod 12 or z ≡ 5 mod 12) then
                prime[z] = ¬prime[z]
            end if
            z ← 3x² + y²
            if z ≤ n & z ≡ 7 mod 12 then
                prime[z] = ¬prime[z]
            end if
            z ← 3x² − y²
            if x > y & z ≤ n & z ≡ 11 mod 12 then
                prime[z] = ¬prime[z]
            end if
        end for
    end for
    prime[2] ← True
    prime[3] ← True
    prime[5] ← True
    return prime
end function
```

However, this doesn't remove the numbers which have a perfect square as a factor.

Algorithm 5 Remove Perfect Squares

```
function REMOVE(prime)
    for i ∈ {k²|k ∈ {1, 2, ..., n}} do
        if prime[i] = True then
            for j ∈ {k ∈ ℕ|ki² < n} do
                prime[ji²] ← False
            end for
        end if
    end for
    return prime
end function
```

3 Optimisations

The algorithm involves a pair of nested for loops, this makes it ideal for parallelisation. We're generally computing primes for value of n above 10^6. We clearly don't have sufficient number of threads in a contemporary GPU to initialise a thread for each value of x and y. Of course when the number of operations exceeds the number of cores we can do each of the operations in batches. We would however like to avoid this as we're just doing the operations in series again. Instead our implementation is going to only use $\sqrt{n}$ threads. There's two options here either computing the inner or outer loop in parallel. We will be paralleling the inner loop so that each thread has less computation to perform before needing. The threads will all end at a similar time so that the program can continue to the next batch.

Algorithm 6 Atkins Sieve Kernel

```
function ATKINSKER(x,y, n)
    z ← 4x² + y²
    if z ≤ n & (z ≡ 1 mod 12 or z ≡ 5 mod 12) then
        prime[z] = ¬prime[z]
    end if
    z ← 3x² + y²
    if z ≤ n & z ≡ 7 mod 12 then
        prime[z] = ¬prime[z]
    end if
    z ← 3x² − y²
    if x > y & z ≤ n & z ≡ 11 mod 12 then
        prime[z] = ¬prime[z]
    end if
end function
```

Here note that notationally we're notating both x and y as inputs to the kernel. However during implementation one of the variables we get from an identifier for the thread. In CUDA this is written int x = threadIdx.x.

And similar for the removal of perfect squares.

Algorithm 7 Remove Perfect Squares Kernel

```
function RemoveKer(i, j, n)
    i_squared ← i^2
    if i_squared × j < n then
        prime[i_squared × j] ← False
    end if
end function
```

Then we can move the prime list into the GPU memory then we need only access the *prime* array using the GPU memory then load it all back at the end. Syntax for this is very much language specific so is omitted from Pseudocode.

Algorithm 8 Multi-threaded Sieve of Atkin

```
function MultiAtkin(n)
    Array<bool> prime[n] = False          ▷ Allocate in GPU cache
    for x ∈ {1, 2, .., √n} do
        Dispatch √n AtkinsKer
    end for
    for i ∈ {k^2 | k ∈ {1, 2, ..., √n}} do
        Dispatch √n RemoveKer
    end for
    return prime          ▷ After loading the array back into the CPU cache
end function
```

My work has focused on improving time complexity, for this case going to $O(\sqrt{n})$ for sufficient number of GPU cores, although in practice this becomes closer to $O(n)$ as seen in figure 1.

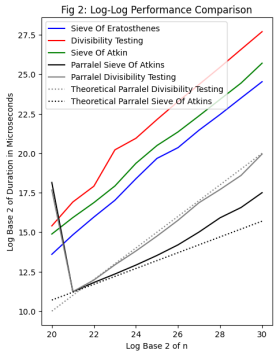

Figure 1: log-log plot of the duration of different sieve methods.

Further, improvements can also be made to memory complexity. Segmentation turns memory complexity from $O(n)$ to $O(\sqrt{n})$.