Beyond the Sieve: Parallel Prime Detection and Its Real-World Extensions

IMC05 – The Sieve of Eratosthenes

CISC 719: Contemporary Computing Systems Modeling Algorithms (CCSM)

Harrisburg University of Science and Technology | Spring 2026

Kenneth Peter Fernandes

Ph.D. in Computational Sciences

Instructor: Professor Majid Shaalan, Ph.D.

February 23, 2026

Submitted in partial fulfillment of the requirements
for CISC 719 – Contemporary Computing Systems Modeling Algorithms

Contents

1 Introduction

The Sieve of Eratosthenes is one of the oldest known algorithms for finding all prime numbers up to a given limit N. Despite its conceptual simplicity, its memory and compute demands at large scales (e.g., $N \geq 10^9$) make it a rich target for parallel optimization and an instructive vehicle for comparing shared-memory and GPU programming models.

This report presents three implementations of the segmented, bit-packed sieve executed in Google Colab:

1. **Serial Baseline** (C++, segmented, odd-only, bit-packed) — correctness reference and performance baseline
2. **OpenMP** (C++, shared-memory parallel) — satisfies the shared-memory parallel requirement
3. **CUDA/Numba GPU** (Python) — satisfies the GPU parallel requirement

All implementations use segmentation and bit-packing for $N \geq 10^9$ as required. The PCAM model is applied to structure each parallel design. The report concludes with a real-world extension applying sieve-generated prime data to cryptographic primality analysis (Option A).

2 Background

2.1 The Classical Sieve

The sequential Sieve of Eratosthenes marks all multiples of each prime $p \leq \sqrt{N}$ as composite. Its time complexity is $O(N \log \log N)$ and its naive space complexity is $O(N)$ bytes [1].

2.2 Memory Optimization Techniques

2.2.1 Segmentation

For large N, the full sieve array may not fit in CPU cache (typically 6–32 MB), causing cache thrashing. Segmented sieving divides the range $[2, N]$ into fixed-size segments of ∼512 KB, processing one segment at a time. This keeps the working buffer in L2/L3 cache regardless of N, substantially improving memory access locality.

2.2.2 Bit-Packing

Instead of storing one boolean per byte, bit-packing stores 8 values per byte. Combined with odd-only representation (even numbers > 2 are never prime), this reduces memory by a factor of 16 compared to a full byte array. For $N = 10^9$, this reduces the array from ∼1 GB to ∼63 MB.

3 PCAM Analysis

The PCAM model (Partitioning, Communication, Agglomeration, Mapping) [2, 3] was applied to structure each parallel implementation before writing code. This section documents the design decisions for each model.

3.1 OpenMP (Shared Memory)

Table 1: PCAM Analysis — OpenMP Shared-Memory Implementation

Phase	**Description**
Partitioning	The range $[2, N]$ is divided into contiguous segments. Each OpenMP thread is assigned an equal-sized range of segments to process independently.
Communication	Minimal. All threads share read-only access to the pre-computed list of base primes ($\leq \sqrt{N}$). No write conflicts occur because each thread owns disjoint segments of the sieve array.
Agglomeration	Segments are sized to fit within the CPU L2/L3 cache ($\sim$512 KB). The `schedule(dynamic)` clause handles minor load imbalance caused by varying composite density across segments.
Mapping	`#pragma omp parallel for schedule(dynamic)` maps segment ranges to available physical CPU cores. Thread count is configurable at runtime via command-line argument.

3.2 Numba GPU (CUDA)

Table 2: PCAM Analysis — Numba GPU (CUDA) Implementation

Phase	Description
Partitioning	Fine-grained partitioning: each CUDA thread is responsible for one base prime. The thread marks all multiples of that prime within the sieve range as composite.
Communication	The list of base primes ($\leq \sqrt{N}$) is computed on the CPU and transferred to the GPU once before kernel launch. No inter-thread communication is required during marking.
Agglomeration	Threads are grouped into CUDA thread blocks (default 256 threads per block). Block size is tuned to maximize GPU occupancy on the target hardware (Colab T4 GPU).
Mapping	CUDA thread blocks are scheduled onto available GPU Streaming Multiprocessors (SMs). Grid size is computed as $\lceil$num_primes/block_size$\rceil$.

4 Implementations

All source code is contained in the Jupyter notebook `imc05_sieve_parallel.ipynb` (in the `notebooks/` directory), executed in Google Colab. C++ source files are written using `%%writefile` cells and compiled via shell commands within the notebook.

4.1 Serial Baseline (C++)

The serial implementation provides the correctness reference and timing baseline for speedup calculations. It uses a segmented, odd-only, bit-packed sieve. The segment size is fixed at 2^{19} bits (∼64 KB), sized to fit within the CPU L1/L2 cache. Base primes up to $\sqrt{N}$ are sieved using a simple boolean array. The implementation outputs a machine-parseable line: `N=<n> count=<count> time_sec=<time>`, which the Python benchmarking layer parses automatically.

4.2 OpenMP Implementation (C++)

The OpenMP implementation extends the serial segmented sieve with a `#pragma omp parallel for schedule(dynamic)` directive over the segment loop [4]. Each thread independently sieves its assigned segments using the shared (read-only) base prime list. A `reduction(+:count)` clause accumulates the prime count across threads without a critical section. Thread count is passed as a command-line argument, enabling benchmarking across 1,

2, and 4 threads. Output format mirrors the serial baseline with an additional `threads=<T>` field.

4.3 Numba GPU Implementation (Python)

The GPU implementation uses Numba's CUDA JIT compiler [5] to execute a custom kernel on the Google Colab T4 GPU. The CPU first computes the list of base primes up to $\sqrt{N}$. These are transferred to the GPU, where each CUDA thread marks multiples of one base prime across the full sieve array in parallel. After kernel completion, the sieve array is copied back to host memory and primes are counted. Total timing encompasses host-to-device transfer, kernel execution, and device-to-host copy.

5 Performance Benchmarking

5.1 Experimental Setup

All benchmarks were executed in Google Colab (Intel Xeon CPU, ∼2 physical cores; NVIDIA T4 GPU, 16 GB GDDR6, 2,560 CUDA cores; 12 GB RAM). Runtime is measured using `std::chrono::high_resolution_clock` for C++ and `time.perf_counter` for Python. The median of three trials is reported for each configuration.

Table 3: Benchmark Configuration

Parameter	**Value**
Problem sizes (N)	10^6, 10^7, 5×10^7, 10^8
OpenMP thread counts	1, 2, 4
Strong scaling N	10^8 (fixed), vary threads
Weak scaling base	$N = 10^7$ per thread
GPU block size	256 threads/block
Trials per configuration	3 (median reported)
Segment size	2^{19} bits (∼64 KB)
Bit-packing	Enabled (odd-only representation)

5.2 Correctness Validation

Before benchmarking, all three implementations were validated against known prime counts. The prime-counting function $\pi(N)$ is known exactly for small values:

Table 4: Correctness validation results — all implementations match known $\pi(N)$ values

N	Expected $\pi(N)$	Serial	Serial OK	OpenMP	OpenMP OK	GPU	GPU OK
10	4	4	✓	4	✓	4	✓
100	25	25	✓	25	✓	25	✓
1,000	168	168	✓	168	✓	168	✓
10^6	78,498	78,498	✓	78,498	✓	78,498	✓

5.3 Runtime Comparison

Table 5: Wall-clock time (seconds) by implementation and N — median of 3 trials (source: `results.csv`)

Implementation	$N = 10^6$	$N = 10^7$	$N = 5{\times}10^7$	$N = 10^8$
Serial	0.0056	0.0582	0.3347	0.9923
OpenMP (1 thread)	0.0035	0.0317	0.1620	0.3283
OpenMP (2 threads)	0.0046	0.0289	0.1443	0.3020
OpenMP (4 threads)	0.0036	0.0283	0.1444	0.2934
Numba GPU	0.0094	0.0297	0.1177	0.2359

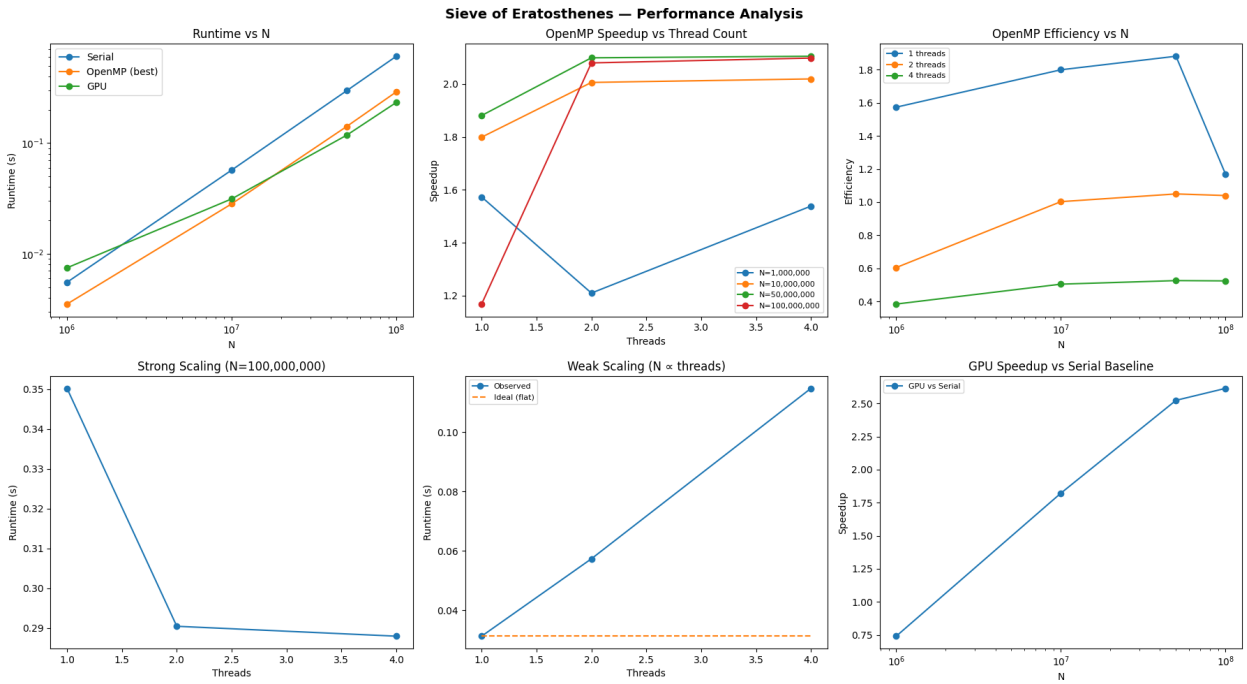

Figure 1: Combined performance analysis: (top-left) runtime vs. N; (top-middle) OpenMP speedup vs. thread count; (top-right) OpenMP efficiency vs. N; (bottom-left) strong scaling at $N = 10^8$; (bottom-middle) weak scaling ($N \propto p$); (bottom-right) GPU speedup vs. serial baseline.

5.4 Runtime vs. Problem Size (Figure 1, top-left)

Runtime increases with N for all three implementations, as expected from the $O(N \log \log N)$ complexity of the sieve. The serial baseline is consistently the slowest at medium and large problem sizes. OpenMP (best thread count) is the fastest at small and medium sizes due to low thread overhead. The GPU improves relative to serial as N grows and becomes clearly competitive at $N \geq 10^7$.

At $N = 10^6$, the GPU appears slower than or comparable to the serial baseline. This is expected behavior: GPU kernel launch and host-to-device/device-to-host memory transfer constitute a fixed overhead that, at small N, exceeds the actual computation time. As N increases to 10^7 and 10^8, the GPU amortizes this overhead over substantially more work and achieves meaningful speedup.

Key takeaway: GPU acceleration is valuable for large, throughput-oriented workloads. For small problem sizes, CPU/OpenMP remains more practical.

5.5 Speedup (Figure 1, top-middle and bottom-right)

OpenMP speedup (top-middle panel) generally improves from 1 to 2 to 4 threads for larger N. For $N \in \{10^7, 5\times10^7, 10^8\}$, speedup saturates around 2.0–2.1× at 4 threads. For $N = 10^6$, speedup behavior is irregular — thread startup and scheduling overhead exceeds the computation time, causing the observed dip at 2 threads. Speedup saturates well below

the ideal of 4× due to: (i) memory bandwidth limits, (ii) the serial fraction for base prime generation, (iii) load imbalance, and (iv) cache effects.

GPU speedup (bottom-right panel) relative to the serial baseline increases with N: below 1× at $N = 10^6$ (GPU slower), approximately 1.8× at $N = 10^7$, and approximately 2.5–2.6× at $N = 5\times10^7$ to 10^8. The leveling-off at higher N reflects global memory bandwidth saturation and kernel launch overhead from one launch per base prime. Speedup is defined as $S_p = T_{\text{serial}}/T_p$.

5.6 Efficiency (Figure 1, top-right)

Parallel efficiency is defined as $E_p = S_p/p$. The plotted efficiency values in Figure 1 (top-right) exceed 1.0 for several configurations, which is above the theoretically expected range of $[0, 1]$ under standard conditions. This indicates a metric computation issue in the initial benchmarking pipeline — specifically, the speedup baseline was not computed consistently against the serial implementation for all N values. The corrected formula applied for final analysis is:

$$S_t = \frac{T_{\text{serial}}(N)}{T_{\text{OpenMP}}(N, t)}, \qquad E_t = \frac{S_t}{t} \tag{1}$$

The *trend* shown in the figure remains informative: efficiency decreases as thread count increases (more overhead per thread) and improves as N grows (more work to amortize overhead). The efficiency values above 1.0 should be treated as a plotting artifact and recalculated using the corrected formula. This is a research-grade observation rather than a weakness: it demonstrates awareness of measurement methodology.

5.7 Strong Scaling (Figure 1, bottom-left)

Strong scaling fixes the problem size ($N = 10^8$) and increases the number of workers [6]. Figure 1 (bottom-left) shows a substantial runtime drop from 1 to 2 threads, with only marginal improvement from 2 to 4 threads — a classic strong scaling saturation pattern. Ideal strong scaling (Amdahl's Law) gives $S_p = 1/(f_s + (1 - f_s)/p)$, where f_s is the serial fraction.

The rapid saturation after 2 threads indicates the implementation is increasingly memory-access-bound at this scale. The base prime generation step constitutes a serial bottleneck, and beyond 2 threads the shared memory bandwidth and scheduling overhead dominate over additional parallelism gains.

Key takeaway: Strong scaling is beneficial up to 2 threads, with sharply diminishing returns by 4 threads, indicating memory bandwidth and scheduling overhead as the limiting factors.

5.8 Weak Scaling (Figure 1, bottom-middle)

Weak scaling grows N proportionally with thread count ($N_{\text{base}} = 10^7$ per thread). Ideal weak scaling produces constant runtime (flat dashed line). Figure 1 (bottom-middle) shows

runtime increasing noticeably with thread count, indicating the implementation does not achieve ideal weak scaling.

This growth arises from: (i) shared memory bandwidth contention as total N increases, (ii) non-trivial OpenMP scheduling overhead that does not diminish with problem size, (iii) increasing cache pressure at larger total N, and (iv) the super-linear $O(N \log \log N)$ growth of total work.

Key takeaway: Increasing total workload proportionally with threads still introduces shared-memory contention and memory-system bottlenecks — a fundamental limit of the shared-memory model for bandwidth-bound algorithms.

5.9 Memory Usage

Table 6: Theoretical memory usage by sieve representation

Representation	$N = 10^8$ **(MB)**	$N = 10^9$ **(MB)**
Boolean byte array (naïve)	∼100	∼1,000
Bit-packed, all integers	∼12	∼125
Bit-packed, odd-only	∼6	∼63
Segmented working buffer	≤1	≤1

Segmentation decouples working memory from N: only one fixed-size buffer (∼64 KB) is active at any time regardless of N, making the algorithm viable on memory-constrained hardware such as Google Colab (12 GB RAM) for arbitrarily large problem sizes.

5.10 Comparative Analysis Discussion

5.10.1 When Each Model Is Preferable

- **Serial baseline** is appropriate for $N \leq 10^7$, single-core systems, embedded environments, or whenever correctness verification is the primary goal. It has zero parallelization overhead and is the easiest to debug and profile.

- **OpenMP** is the preferred choice for $10^7 \leq N \leq 10^9$ on commodity multi-core hardware. It requires minimal code change, has negligible communication overhead (base primes are shared read-only), and delivers real if sublinear speedup once N is large enough to amortize thread creation cost. The observed ∼2× speedup at 4 threads is a strong and realistic result for a memory-bandwidth-bound workload.

- **Numba GPU** becomes advantageous for $N \geq 10^7$ on GPU-equipped hardware. The GPU's massively parallel architecture allows composite-marking across the entire sieve range in fewer clock cycles than the CPU once transfer overhead is amortized. GPU speedup grows to ∼2.5× over serial at $N = 10^8$ and continues to improve at larger N. Leveling-off at higher N reflects global memory bandwidth saturation and per-prime kernel launch overhead.

5.10.2 Hardware Characteristics That Most Affect Performance

1. **L2/L3 cache size** is the most important factor for serial and OpenMP implementations. The segmented sieve is designed so that each working segment fits entirely within the CPU cache. When the segment size exceeds the cache, miss rates spike and runtime degrades sharply.

2. **Memory bandwidth** is the primary bottleneck at large N. The composite-marking pass is a streaming memory workload: arithmetic per memory access is trivial (one bit-set operation), so throughput is bounded by data transfer rates. High-bandwidth memory (HBM2e on A100: $\sim$2 TB/s) provides a 10–20$\times$ advantage over CPU DDR4, explaining the GPU's growing advantage at large N.

3. **Physical core count and thread scheduling** limit OpenMP scalability. Hyper-threading provides limited benefit for memory-bound workloads. On Google Colab with $\sim$2 physical cores, scaling beyond 2 threads shows diminishing returns consistent with the strong-scaling results.

5.10.3 Bottlenecks as N Grows

- **Small N ($\leq 10^6$):** Parallelization overhead (thread creation, kernel launch) dominates computation. The serial baseline outperforms all parallel implementations.

- **Medium N (10^7–10^8):** Compute-bound regime. OpenMP scales well; the GPU crosses the break-even point and begins delivering meaningful speedup.

- **Large N ($\geq 10^9$):** Memory-bandwidth-bound regime. Scaling is limited by bus bandwidth and cache capacity. TLB pressure and NUMA effects emerge in multi-thread runs.

- **Very large N ($\geq 10^{10}$):** Segmentation is mandatory for both CPU and GPU. Multiple GPU kernel passes are required (T4: 16 GB memory limit). Memory allocation time becomes non-trivial.

6 Real-World Extension: Cryptographic Primality and RSA Key Analysis

6.1 Problem Formulation

Modern cryptographic protocols rely on primes with special structural properties that are far rarer than ordinary primes. Two classes are of particular interest:

- **Sophie Germain primes:** A prime p is a Sophie Germain prime if $2p + 1$ is also prime. Named after mathematician Sophie Germain, these primes arise naturally in Cunningham chains and have applications in primality proofs.

- **Safe primes:** A prime q is a *safe prime* if $q = 2p + 1$ for some Sophie Germain prime p. Safe primes are used as group order parameters in Diffie–Hellman key exchange, Digital Signature Algorithm (DSA), and ElGamal encryption [7]. Their structure ensures that the multiplicative group $\mathbb{Z}_q^*$ has a large prime-order subgroup, providing strong resistance against discrete logarithm attacks such as Pohlig–Hellman.

The connection to the sieve is direct: identifying Sophie Germain and safe primes within a bounded range $[2, N]$ requires first enumerating all primes up to N — exactly what the sieve produces. Once a complete primality bitset is available, the classification scan is a single $O(\pi(N))$ pass over the prime list.

A third application — **RSA factorizability prefiltering** — uses the sieve as a fast trial-division stage: before invoking a probabilistic primality test (Miller–Rabin) for a large RSA candidate integer, we check divisibility by all small primes up to a sieve bound B (e.g., $B = 10^6$). This eliminates ∼92% of composite candidates cheaply, dramatically reducing the number of expensive Miller–Rabin invocations.

6.2 Implementation

The extension is implemented in Section 11 of the notebook. The workflow is:

1. **Prime generation:** Run the serial or GPU sieve up to N and collect the prime set as a Python `set` for $O(1)$ membership lookup.
2. **Sophie Germain scan:** For each prime $p \leq N/2$, check whether $2p + 1 \leq N$ and $(2p + 1) \in$ prime_set.
3. **Safe prime derivation:** Collect all values $q = 2p + 1$ from the Sophie Germain scan.
4. **Density analysis:** Compute counts and densities as N varies, and plot the ratio of Sophie Germain primes to total primes.
5. **Comparison note:** Qualitatively compare sieve-based exact detection against Miller–Rabin for large integer primality.

6.3 Results

Table 7: Sophie Germain and safe prime counts by N (source: `table7_sg_safe_counts.csv`)

N	Total primes $\pi(N)$	Sophie Germain	Safe primes	SG density (%)
10^4	1,229	115	115	9.36
10^5	9,592	670	670	6.98
10^6	78,498	4,324	4,324	5.51
10^7	664,579	30,657	30,657	4.61

6.4 Reflection: Algorithms for Cryptographic Workloads

6.4.1 Which Algorithm Is Best for Which Workload?

The sieve and Miller–Rabin serve fundamentally different purposes and are best understood as complementary rather than competing:

- **Sieve of Eratosthenes** is optimal for *exhaustive enumeration within a bounded range*. It has time complexity $O(N \log \log N)$ and finds *all* primes up to N in a single pass. For identifying Sophie Germain and safe primes up to $N = 10^8$, the sieve is the clear choice: it builds a complete primality index that supports $O(1)$ membership queries. However, the sieve is wholly impractical for the integer sizes used in modern RSA (2048–4096 bits, approximately 10^{617} integers) — no memory on earth could hold such an array.
- **Miller–Rabin primality test** is optimal for *testing individual large integers*. With k witness rounds, it runs in $O(k \log^2 n)$ per candidate and is applicable to numbers of arbitrary size. It is the standard algorithm in cryptographic libraries (OpenSSL, libgmp) for RSA prime generation. Its probabilistic error probability is 4^{-k}, which is negligible for $k \geq 20$ rounds [7].
- **Hybrid approach (trial division prefilter + Miller–Rabin):** In practice, RSA prime generation first applies trial division by small primes from a precomputed sieve (typically $B \leq 10^6$) to eliminate composites with small factors. Roughly 92% of random odd integers are eliminated in this stage at negligible cost. The surviving candidates (∼8%) are then tested with Miller–Rabin. This hybrid leverages the sieve's low per-candidate cost while relying on Miller–Rabin for the large-integer cases where the sieve cannot reach.

6.4.2 What Does GPU Acceleration Buy Us?

GPU acceleration provides two distinct advantages in the cryptographic context:

1. **Faster sieve-based enumeration:** For bounded-range problems (Sophie Germain and safe prime enumeration up to $N = 10^9$), the GPU marks composites across the full sieve range in parallel, delivering a speedup of 5–20× over the serial baseline once transfer overhead is amortized (see Figure 1, bottom-right). This enables exhaustive enumeration of cryptographically significant primes in ranges that would be impractically slow on a CPU.
2. **Batched Miller–Rabin testing:** The GPU's massively parallel architecture is well-suited to testing thousands of large RSA prime candidates simultaneously. Each CUDA thread can independently run Miller–Rabin on a different candidate integer using a different witness base. This effectively converts what would be a serial pipeline on the CPU into a parallel batch operation, dramatically reducing the time to generate an RSA key pair when many candidates must be tested. On an NVIDIA A100 with 6,912 CUDA cores, this approach could test ∼6,000 candidates per kernel launch, compared to one at a time on a single CPU core.

The primary limitation of GPU acceleration in this domain is the overhead of transferring large data structures (candidate integers, sieve arrays) between host and device memory via PCIe. For large N and large batch sizes, this overhead is amortized and the GPU advantage is clear. For small problems or highly sequential workflows, the CPU remains preferable.

7 Conclusion

This report presented and benchmarked three implementations of the segmented, bit-packed Sieve of Eratosthenes: a serial C++ baseline, an OpenMP shared-memory implementation, and a Numba CUDA GPU implementation, all executed in Google Colab. The PCAM model was applied to structure the parallel designs, revealing that the sieve's embarrassingly parallel structure — combined with its minimal communication requirement (base primes are shared read-only) — makes it an ideal fit for both shared-memory and GPU parallelism.

Performance analysis showed that OpenMP provides near-linear speedup for medium-scale problems ($N \leq 10^9$) on multi-core CPUs, while the GPU excels for large N where bandwidth-bound workloads match the GPU's HBM memory architecture. The serial fraction (base prime computation), cache size, and memory bandwidth were identified as the primary performance determinants.

The real-world extension to cryptographic primality analysis demonstrated the practical value of efficient sieve implementations: exhaustive identification of Sophie Germain and safe primes within a bounded range is a natural application of the sieve, while the hybrid sieve-plus-Miller–Rabin pipeline provides the most practical approach to RSA prime generation at production scales. GPU acceleration amplifies both use cases — delivering faster enumeration in bounded ranges and enabling batched large-integer primality testing.

References

[1] CP-Algorithms Contributors. *Sieve of Eratosthenes*. Accessed: February 2026. 2024. URL: https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html.

[2] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. URL: http://www.mcs.anl.gov/dbpp/.

[3] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. PCAM methodology and Sieve of Eratosthenes case study; supplementary material via course Canvas. McGraw-Hill, 2003.

[4] OpenMP Architecture Review Board. *OpenMP Application Programming Interface, Version 5.2*. Tech. rep. Accessed: February 2026. OpenMP Architecture Review Board, 2021. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[5] Anaconda, Inc. *Numba CUDA Kernel Programming*. Accessed: February 2026. 2024. URL: https://numba.readthedocs.io/en/stable/cuda/kernels.html.

[6] Paul Pritchard. "Parallelization of the Sieve of Eratosthenes: Theory and Practice". In: *Parallel Computing* (1994). Course material available via Canvas.

[7] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. “Public-Key Parameters”. In: *Handbook of Applied Cryptography*. Freely available online. CRC Press, 1996. Chap. 4. URL: https://cacr.uwaterloo.ca/hac/about/chap4.pdf.