# NGrams with CUDA

Alessandro Bianco
alessandro.bianco1@edu.unifi.it

## Abstract

*N-gram counting is a fundamental operation in NLP, yet it poses significant challenges for GPU architectures by introducing a trade-off between atomic contention and memory complexity. In this work, we design and implement three distinct parallel strategies to address this problem. The first two (V1 and V2) are histogram-based: a 'Global Atomic' (V1) approach, which risks high contention, and a 'Private' (V2) approach, designed to mitigate it. Both, however, exhibit an exponential memory complexity of O(C^n). To overcome this limitation, we implement a third 'Map-Sort-Reduce' (B) strategy, which trades time complexity (O(N log N)) for a scalable O(N) memory footprint. This paper presents a performance analysis of these different approaches, testing their bottlenecks against varying data size ('N') and problem complexity ('n').*

## Future Distribution Permission

## 1. Introduction

N-gram counting is a fundamental operation in Natural Language Processing, used in tasks such as language modeling, machine translation, and sentiment analysis. Given the typical size of modern text, calculating the frequency of billions of N-grams becomes a computational bottleneck. The massively parallel architecture of modern GPUs offers a promising solution.

Harnessing this parallelism, however, is not trivial and introduces a fundamental trade-off. The most intuitive approach, based on histograms, collides with two distinct bottlenecks: memory contention, where thousands of threads attempt to atomically update the same counters, and an exponential memory complexity ($O(C^n)$), which makes the histogram unmanageable for high values of 'n'.

To overcome these limitations, an alternative paradigm could be using a Map-Sort-Reduce. This approach bypasses the $O(C^n)$ memory problem by adopting a scalable $O(N)$ memory footprint, at the cost of a theoretically worse time complexity ($O(N \log N)$) due to the sorting phase.

In this report, we implement and analyze three GPU algorithms that embody this conflict: **V1** (Global Atomic Histogram), **V2** (Private Histograms, to mitigate contention), and **B** (Map-Sort-Reduce). The final objective is to demonstrate that no single solution prevails, but rather an optimal hybrid strategy that selects the most suitable algorithm for each specific scenario.

## 2. Algorithmic Methodologies

All three implemented parallel algorithms share a fundamental core concept: the method of mapping a character N-gram to a unique numerical index.

### 2.1. Flat Indexing (Base 256)

A character N-gram is a contiguous sequence of $n$ characters. The most common approach to count them is to use a sliding window of size $n$ that moves across the entire corpus, one character at a time.

To parallelize this process on a GPU, each thread is responsible for one N-gram. To efficiently count occurrences, each unique N-gram must be mapped to an integer index (a "flat index"). By treating the corpus as an array of bytes (values 0-255), we can interpret an N-gram of $n$

characters as a number in base 256.

The kernel implements this calculation by iterating over the N-gram from right to left (from character index `i = n-1` down to `i = 0`). The rightmost character (`text[idx + n - 1]`) is treated as the Least Significant Byte (LSB) and multiplied by $256^0$, while the leftmost character (`text[idx + 0]`) is treated as the Most Significant Byte (MSB) and multiplied by $256^{n-1}$.

The formula implemented by the kernel is as follows:

$$idx = \sum_{i=0}^{n-1} \text{char}[idx + i] \times 256^{(n-1-i)} \tag{1}$$

This unique index is fundamental:

- **In Algorithms V1 and V2**, it is used as the index into the histogram array to (atomically) increment the counter.
- **In Algorithm B**, it is the ID that is written to the intermediate array, which is later sorted and reduced.

## 2.2. Algorithm V1: Global Histogram (Atomic)

Algorithm V1 represents the most direct approach to parallelizing the problem. The fundamental idea is to map every single N-gram in the corpus to a GPU thread. Each thread is then responsible for calculating the unique index of its N-gram and incrementing the corresponding counter in a single, shared, global histogram.

The `_compute_v1` host function handles data preparation and kernel launch. Its critical step is allocating the GPU histogram, whose exponential size ($256^n$) is the primary bottleneck. After transferring the corpus to VRAM, the function calculates the grid to launch one thread per N-gram. It then launches the `kernel_v1_cupy` kernel and, after synchronization, copies the completed histogram back to the CPU.

The `char_ngram_kernel` executes the parallel work. Each thread first calculates its global index and performs a bounds check to prevent out-of-range access. It then computes the unique flat index using the Base-256 logic. Finally, it uses the core `atomicAdd(&histogram[flat_idx], 1)` instruction to safely increment the correct counter, guaranteeing mutual exclusion and preventing race conditions among the thousands of concurrent threads.

This approach presents a clear trade-off. Its primary advantage is an $O(N)$ time complexity, as each N-gram is processed in a single pass (excluding allocation overhead). However, it suffers from two potential bottlenecks: contention on the `atomicAdd` instruction and exponential memory complexity. Our tests will show that contention is negligible for $n = 2$ (when the histogram fits in the L2 cache) but becomes a factor for $n >= 3$, while the second disadvantage, its $O(C^n)$ memory requirement, is fatal for larger $n$ sizes.

## 2.3. Algorithm V2: Private Histograms

Algorithm V2 (Private Histograms) is a direct attempt to solve the primary bottleneck of V1: `atomicAdd` contention. The strategy is to replace the single, highly contended global histogram with $K$ smaller, private histograms. Threads are distributed among these, reducing contention. This, however, requires a two-kernel process: a first kernel to populate the private histograms and a second kernel to sum them into a final result.

The `_compute_v2` host function orchestrates this two-stage process. First, it allocates a large, contiguous block of memory for the private histogram gpu, a critical step as its size is $O(K \times C^n)$. This makes its memory footprint significantly larger than V1's, causing OOM failures to occur even sooner (as our tests will show). It then launches the `kernel_v2_private_cupy` to populate these histograms. After synchronizing, it allocates the final global histogram gpu and launches the `kernel_v2_reduce_cupy` kernel to sum the partial results. Finally, it copies the global histogram back to the CPU.

The device implementation is split into two kernels. The first, is similar to V1's, but instead of writing to a single histogram, each thread block writes to one of the $K$ private histograms based on its `blockIdx.x`. This effectively eliminates V1's global atomic contention. The second kernel, performs a parallel reduction where each thread is responsible for a single bin index. Each thread then loops through all $K$ private histograms, sums the value for its specific bin, and writes the final total to the global histogram.

This method brings in a new and more complicated trade-off. The main benefit is that it effectively removes the global atomic contention seen in V1 because threads are spread across multiple ($K$) histograms. However, this does come with a significant downside. Firstly, it adds the extra work of launching a second kernel and requires a large amount of intermediate memory. More importantly, it worsens the memory bottleneck issue.

### 2.4. Algorithm B: Map-Sort-Reduce

Algorithm B represents a complete paradigm shift, designed to overcome the $O(C^n)$ exponential memory bottleneck of the histogram-based approaches. Instead of creating an enormous histogram, this algorithm uses a Map-Sort-Reduce strategy, where the memory footprint is ruled by the number of N-grams ($O(N)$), not their combinatorial space.

The `_compute_B` host function orchestrates a pipeline of GPU operations. First, it allocates an array, with a size equal to the total number of N-grams in the text ($O(N)$). It then launches the first kernel to perform the Map phase, populating this array with the ID for each N-gram. After the map kernel is synchronized, the host calls `cp.sort()`, a highly optimized parallel sort, on the generated array. This Sort phase groups all identical N-gram IDs contiguously. Finally, the host performs the Reduce phase by calling `cp.unique(return_counts=True)`, which efficiently counts the occurrences of each unique ID. The resulting arrays of unique IDs and their counts are then copied back to the CPU for final dictionary construction.

This mapping kernel logic phase is simple: each thread computes its index and calculates the flat index (the N-gram ID) using the same Base-256 logic as V1. However, instead of an atomic add, it performs a single, contention-free write of this ID into the output array: `ngram_ids_output[idx] = flat_idx`. The subsequent Sort and Reduce phases do not use custom kernels but they leverage CuPy's built-in, which are highly optimized parallel primitives.

The primary advantage of Algorithm B is its memory complexity. The $O(N)$ memory footprint is independent of 'n', making it the only algorithm in our study that can successfully run for $n \geq 3$. Its performance is also highly predictable, as the time is dominated by the `cp.sort` operation. The main disadvantage is its time complexity, $O(N \log N)$, due to this sort.

## 3. Experimental Setup

To simulate workloads of varying sizes and test 'N' scalability, the corpus is read and duplicated in memory using an amplification factor ($k$). This operation is handled by the `amplify_corpus` function.

Given that the GPU algorithms operate at the byte level to calculate indices (as described in Section 2.1), the text corpus is converted into a `numpy.uint8` array using `latin-1` encoding. This encoding ensures a 1:1 mapping between characters and 8-bit values, preserving data integrity for the CUDA kernels.

### 3.1. Benchmark Methodology

The testing infrastructure is designed to run two main experiments:

- **Experiment A ('n' Scalability):** Tests performance by varying the N-gram size while keeping the corpus size fixed ($k = 10$).
- **Experiment B ('N' Scalability):** Tests performance by varying the corpus size while keeping $n$ fixed.

Each test run is managed by the `NgramBenchmark` class. To ensure accurate measurements and to exclude the compilation overhead of the, a warmup call is performed before each experiment.

To achieve statistically stable results, each test configuration is executed `NUM_RUNS` times. The final reported execution time is the mean of these runs, accompanied by the standard deviation.

### 3.2. Evaluation Metrics

The performance of each algorithm is evaluated using two primary metrics, managed by the `performance_timer.py` file:

1. **Correctness:** The GPU results are compared against those of a sequential CPU implementation. The `verify_results` function performs a rigorous comparison to ensure the counts for every N-gram are identical.

2. **Performance (Speedup):** The GPU execution time is measured using `time.perf_counter`. Speedup is calculated as:

$$\text{Speedup} = \frac{\text{Sequential CPU Time}}{\text{Parallel GPU Time}}$$

## 4. Results and Analysis

We present the results from our two primary experiments. First, we analyze the impact of problem complexity ('n') on a fixed-size corpus. Second, we analyze the impact of data size ('N') for the key scenarios where the algorithms are viable.

### 4.1. Experiment A: Scalability vs. Complexity (n)

This experiment tests the exponential memory complexity of the histogram-based algorithms. We held the corpus size fixed at 10x amplification and varied the N-gram size $n = [1, 2, 3, 4]$.

#### 4.1.1 Results

As shown in Table 1, at $n = 1$ and $n = 2$, all algorithms perform correctly. V1 (Global Atomics) is the clear winner, achieving a remarkable 126.7x speedup at $n = 2$. This is due to the small histogram size ($256^2 \times 4\text{B} \approx 256\text{KB}$) fitting entirely within the GPU's L2 cache, which makes atomic contention negligible. V2 (Private) is also fast (112.5x) but slower than V1, as its overhead (a second kernel launch and larger memory footprint) is slowing the performances.

The critical failure points occur as $n$ increases:

- $n = 3$: V2 fails with an Out-of-Memory (OOM) error, as it attempts to allocate $128 \times 67\text{MB} \approx 8.5\text{GB}$. V1's performance plummets from 126.7x to 22.5x. This is the effect of the 67MB histogram no longer fitting in the L2 cache, causing severe VRAM contention. Algorithm B, now faster than V1, remains stable at 27.3x.

- $n = 4$: V1 now also fails (OOM), as it attempts to allocate a 16GB histogram.

Algorithm B is the only strategy that successfully completes all tests, proving its $O(N)$ memory is robust to complexity 'n'.

| Alg | N | Hists | CPU Time (s) | GPU Time (s) | Speedup |
|---|---|---|---|---|---|
| V1 | 1 | - | 2.686±0.015 | 0.031±0.004 | 88.0±10.8x |
| V1 | 2 | - | 3.727±0.550 | 0.029±0.000 | 126.7±20.1x |
| V1 | 3 | - | 3.676±0.081 | 0.174±0.050 | 22.5±7.1x |
| V1 | 4 | - | 4.003±0.519 | $\infty$ (FAIL) | 0.0±0.0x |
| V2 | 1 | 128 | 2.818±0.007 | 0.073±0.081 | 75.9±51.1x |
| V2 | 2 | 128 | 4.555±0.622 | 0.040±0.000 | 112.5±15.6x |
| V2 | 3 | 128 | 3.516±0.020 | $\infty$ (FAIL) | 0.0±0.0x |
| V2 | 4 | 128 | 3.692±0.022 | $\infty$ (FAIL) | 0.0±0.0x |
| B | 1 | - | 3.095±0.478 | 0.155±0.017 | 20.2±4.4x |
| B | 2 | - | 4.783±0.533 | 0.144±0.001 | 33.1±3.7x |
| B | 3 | - | 4.259±0.531 | 0.156±0.001 | 27.3±3.5x |
| B | 4 | - | 4.362±0.540 | 0.203±0.000 | 21.5±2.7x |

Table 1. Experiment A results (Corpus 10x, $n = [1; 4]$). "FAIL" indicates an Out-of-Memory (OOM) error.

### 4.2. Experiment B: Scalability vs. Data Size (N)

#### 4.2.1 Objective

This experiment tests how the algorithms scale with increasing data size ('N'). We analyze the two key cases: $n = 2$ (L2 cache) and $n = 3$ (VRAM).

### 4.2.2 Case n=2 (L2 Cache Scenario)

As shown in Table 2, for $n = 2$, all three algorithms scale linearly with $N$. V1 (Global Atomics) is the undisputed winner, peaking at 142.6x speedup. Its $O(N)$ performance, combined with the L2 cache efficiency, makes it the optimal choice. V2 is also very fast (peaking at 107.8x) but is consistently slower than V1 due to its private allocation and reduction kernel overhead. Algorithm B (Sort) scales well, but its $O(N \log N)$ complexity is not competitive in this scenario, plateauing at $\sim$ 29x.

| Amplify | Alg | GPU Time (s) | Speedup |
|---|---|---|---|
| 1x | V1 | 0.005±0.001 | 71.0±9.6x |
| 5x | V1 | 0.021±0.005 | 90.9±22.2x |
| 10x | V1 | 0.029±0.000 | 142.6±17.5x |
| 20x | V1 | 0.062±0.010 | 128.4±19.5x |
| 50x | V1 | 0.171±0.017 | 101.9±6.5x |
| 1x | V2 | 0.007±0.000 | 46.4±0.7x |
| 5x | V2 | 0.023±0.000 | 90.4±21.5x |
| 10x | V2 | 0.041±0.000 | 107.4±1.9x |
| 20x | V2 | 0.077±0.000 | 107.8±7.5x |
| 50x | V2 | 0.228±0.027 | 88.1±11.7x |
| 1x | B | 0.017±0.000 | 21.7±0.1x |
| 5x | B | 0.075±0.000 | 24.4±0.1x |
| 10x | B | 0.146±0.004 | 29.4±4.1x |
| 20x | B | 0.405±0.208 | 24.4±9.8x |
| 50x | B | 0.718±0.004 | 27.6±1.8x |

Table 2. Experiment B results for n=2 (GPU Time & Speedup).

### 4.2.3 Case n=3 (VRAM Contention Scenario)

The results for $n = 3$, shown in Table 3, are the most complex. V2 fails in all tests (OOM), confirming it is unusable for $n \geq 3$. The race is between V1 and B. For small datasets (1x, 5x), B is faster as V1's VRAM contention overhead dominates. However, as the data size increases (10x, 50x), V1's superior $O(N)$ complexity overtakes B's $O(N \log N)$, making V1 the clear winner for large datasets. V1's speedup (from 2.3x to 48.9x) grows with $N$ as the GPU time scales much better than the linear $O(N)$ CPU time. B's speedup is also stable and strong (peaking at 29.7x).

| Amplify | Alg | GPU Time (s) | Speedup |
|---|---|---|---|
| 1x | V1 | 0.162±0.021 | 2.3±0.3x |
| 5x | V1 | 0.146±0.002 | 14.5±3.8x |
| 10x | V1 | 0.128±0.002 | 33.1±7.8x |
| 20x | V1 | 0.331±0.029 | 23.3±1.0x |
| 50x | V1 | 0.426±0.023 | 48.9±5.3x |
| 1x | V2 | $\infty$ (FAIL) | 0.0±0.0x |
| 5x | V2 | $\infty$ (FAIL) | 0.0±0.0x |
| 10x | V2 | $\infty$ (FAIL) | 0.0±0.0x |
| 20x | V2 | $\infty$ (FAIL) | 0.0±0.0x |
| 50x | V2 | $\infty$ (FAIL) | 0.0±0.0x |
| 1x | B | 0.028±0.000 | 13.3±0.1x |
| 5x | B | 0.089±0.002 | 20.8±0.4x |
| 10x | B | 0.156±0.000 | 29.7±0.1x |
| 20x | B | 0.295±0.005 | 25.4±1.7x |
| 50x | B | 0.788±0.083 | 26.9±1.2x |

Table 3. Experiment B results for n=3 (GPU Time & Speedup).

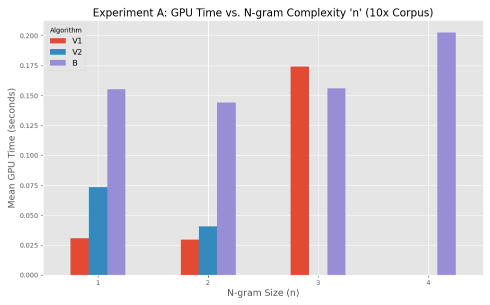

Figure 1. Experiment A: GPU Time (mean) vs. N-gram size 'n'

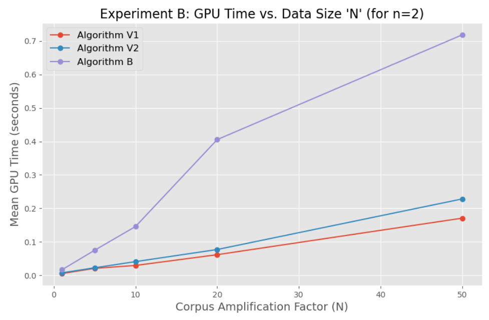

Figure 2. Experiment B (n=2): GPU Time vs. Corpus Size

## 5. Conclusions

Our analysis of GPU-based N-gram counting demonstrates that no single strategy is universally optimal; rather, the most effective approach depends strictly on the N-gram size ('n').

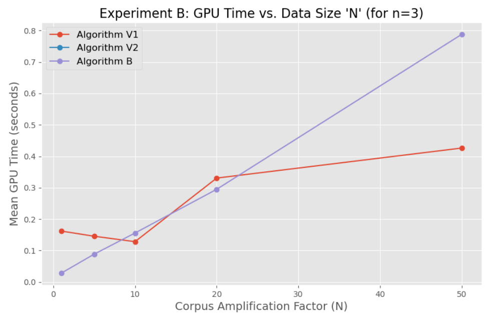

Figure 3. Experiment B (n=3): GPU Time vs. Corpus Size

The Private Histogram approach (Algorithm V2) proved ineffective, suffering from overhead in low-complexity scenarios and immediate Out-of-Memory failures at $n \geq 3$ due to its multiplicative memory footprint. Conversely, the Global Atomic approach (Algorithm V1) dominated for $n \leq 3$, where its linear time complexity $O(N)$ outperformed the sorting-based method, even with VRAM contention. However, V1 hits a hard scalability limit at $n \geq 4$ due to the exponential memory requirement of the histogram. In this high-complexity domain, the Map-Sort-Reduce strategy (Algorithm B) becomes the only viable option, leveraging linear memory scalability. Consequently, we propose a hybrid adaptive strategy: maximize throughput with Algorithm V1 for $n \leq 3$ and switch to Algorithm B for $n \geq 4$ to ensure robustness.