Feasibility of Refactoring Unique ID Algorithms in Elixir for GPU Execution

By: Matheus de Camargo Marques, Web-Engenharia

1. Introduction

1.1. The Imperative of Unique Identifiers in Modern Systems

Unique identifiers (IDs) are fundamental in contemporary software architecture, underpinning a myriad of functionalities in distributed systems, microservices architectures, as primary keys in databases, and for tracking and logging activities.[1] In large-scale environments, the efficient and reliable generation of these IDs presents significant challenges, including generation speed, minimizing the probability of collisions, and, in many cases, the need for sortability to optimize data queries and indexing.[1] The demand for IDs that meet these criteria grows exponentially with the complexity and scale of systems.

1.2. GPU Acceleration: A New Frontier for High-Throughput ID Generation?

General-purpose computing on graphics processing units (GPGPU) has emerged as a transformative technology, capable of massively accelerating inherently parallelizable workloads. Traditionally applied in domains such as scientific computing, machine learning, and image processing, GPGPU offers largely untapped potential in other areas. This report investigates one such area: the possibility of leveraging the massive parallelism of GPUs to generate unique identifiers at a substantially higher throughput than traditional CPU-bound methods, specifically within the Elixir ecosystem. Exploring GPU for ID generation in Elixir transcends the mere pursuit of raw speed; it's about potentially enabling new scales of operation for Elixir applications or offloading CPU-intensive tasks in highly concurrent systems, thereby improving overall system responsiveness. Elixir, with its actor-based concurrency model and execution on the BEAM (Erlang Virtual Machine), is optimized for I/O-bound tasks and concurrency. However, computationally intensive tasks, such as generating IDs in extreme volumes, can become bottlenecks, consuming CPU cycles that could be used by other BEAM processes. Offloading such tasks to a GPU, if feasible and efficient, would free up CPU resources, resulting in a systemic benefit beyond the micro-optimization of ID generation.

1.3. Elixir's Journey into Numerical and GPU Computing

Elixir, known for its robust concurrency and fault tolerance capabilities inherited from the BEAM, has traditionally excelled in web applications and distributed systems. Recently, the Elixir ecosystem has expanded its frontiers towards numerical computing and, by extension, GPU computing. The emergence of the Nx (Numerical Elixir) library and associated projects marks an inflection point, providing the foundations for Elixir to explore domains requiring intensive data processing.[2] This evolution paves the way for investigating the use of GPUs in tasks not traditionally associated with Elixir, such as high-performance ID generation.

1.4. Purpose and Structure of the Report

The primary objective of this report is to investigate the feasibility of refactoring or reimplementing common unique ID generation algorithms for GPU execution, using the Elixir language and its associated tools, such as GPotion and Nx, or alternative approaches like Native Implemented Functions (NIFs) and XLA Custom Calls. The analysis will delve into the computational characteristics of ID algorithms, the capabilities of Elixir's GPU tools to handle the core components of these algorithms (such as timestamps, random number generation, atomic operations, and bitwise manipulation), and will evaluate potential performance benefits against development complexity and inherent challenges (data transfer, kernel latency). The maturity of Elixir's GPU ecosystem, including Nx, EXLA (the XLA backend for Nx), and GPotion, will be a direct indicator of the practicality of this endeavor. High-level tools like Nx aim to simplify GPU programming.[3] However, ID generation may require specific low-level primitives (atomics, precise timers, cryptographically secure pseudo-random number generators - CSPRNGs). If these primitives are not easily accessible through Elixir's high-level GPU libraries, the reliance on NIFs will increase, elevating complexity and security risks.[4] Thus, the "ease" promised by Elixir's GPU tools is conditional on their feature set matching the problem's demands.

The report structure comprises the following sections:

- **Section 2:** Analysis of unique ID algorithms from the perspective of GPU parallelization.
- **Section 3:** Evaluation of the Elixir ecosystem for GPU execution, including Nx/EXLA, GPotion, NIFs, and XLA Custom Calls.
- **Section 4:** Feasibility assessment of implementing ID algorithm components on GPU with Elixir.
- **Section 5:** Performance considerations, benchmarking insights, and trade-offs.
- **Section 6:** Recommendations and future directions.
- **Section 7:** Conclusion, summarizing findings and practical implications.

2. Analysis of Unique ID Algorithms for GPU Parallelization

The feasibility of accelerating unique ID generation on GPUs fundamentally depends on the computational structure of each algorithm and how well its components lend themselves to massively parallel execution.

2.1. Common Computational "Kernels" in ID Generation

Several ID algorithms share computational building blocks that can be analyzed from the perspective of GPU parallelization:

- **Timestamp Acquisition and Manipulation:** Many IDs incorporate a timestamp to ensure temporal uniqueness and, in some cases, sortability. UUIDv1/v6/v7, ULID, and Snowflake are prominent examples.[1] Precision varies: UUIDv1 requires 100-nanosecond intervals, while ULID and Snowflake use milliseconds. A central challenge on the GPU is obtaining synchronized, high-precision timestamps, both among the thousands of parallel threads and with the host system (CPU).
- **Random/Pseudorandom Number Generation (PRNG/CSPRNG):** Randomness is crucial for global uniqueness, especially in IDs like UUIDv4 (which ideally uses 122 random bits, although [1] mentions 80 bits for UUIDv4, which is more characteristic of ULID's random portion) and ULID (80 bits of randomness).[1] It is vital to distinguish between PRNGs, which focus on speed and good statistical properties (like Threefry used in Nx.Random [7]), and CSPRNGs, which are necessary for IDs with security or unpredictability requirements, and typically involve mixing with operating system entropy (like Erlang's :crypto.strong_rand_bytes/1 [8]).
- **Bitwise Operations and Encoding:** Operations like bit shifts, AND, OR, and XOR are ubiquitous for packing timestamps, random bits, and sequence numbers into the final ID structure. Additionally, encoding schemes like Base32 (used by ULID [1]) or Base58/Base64 are often applied to produce compact and URL-friendly textual representations, each with its computational cost.
- **Sequential Number Generation (Atomic Operations):** Algorithms like Snowflake rely on counters that increment monotonically within a specific time unit (e.g., per millisecond).[1] In a highly parallel environment like a GPU, this requires the use of atomic operations (like atomic increment) to prevent race conditions and ensure sequence uniqueness.
- **Cryptographic Hashing:** Certain ID types, notably Content Identifiers (CIDs), require calculating cryptographic hashes (e.g., SHA256) over some content or metadata. These hashes are computationally intensive and can benefit from GPU parallelization.

2.2. Specific Algorithm Suitability and GPU Considerations

Evaluating each algorithm individually:

- **UUID (Universally Unique Identifier):**
  - **UUIDv1/v6 (Timestamp-based):** Composed of a 60-bit timestamp (100ns precision), a clock sequence, and a node ID.[1] Timestamp generation and bit packing are parallelizable. However, managing the node ID in a GPU context (where there isn't a traditional physical "node" for each thread) is challenging. The clock sequence might require atomic operations if generated rapidly by multiple threads for the same simulated "node." Algorithms relying on pre-configured "node IDs" or "datacenter IDs," like Snowflake [1] or UUIDv1, introduce a dependency on external state that needs to be efficiently distributed or made accessible to GPU kernels. This presents a data organization and kernel configuration challenge. If a single GPU is treated as one "node," passing the node ID as a kernel argument is simple. However, if the goal is to simulate multiple "workers" (e.g., each Streaming Multiprocessor - SM - or GPU thread block acting as a worker), then each of these units would need its own unique worker ID.
  - **UUIDv4 (Random):** Primarily consists of random bits (typically 122 random bits, with 6 fixed bits for version and variant). It is highly parallelizable, making batch generation of random numbers on the GPU a strong use case.
  - **UUIDv7 (Timestamp + Random):** Combines a Unix timestamp (millisecond precision) with random bits, designed to be k-sortable. Its GPU suitability is similar to ULID; timestamp and random number generation are parallelizable.
- **ULID (Universally Unique Lexicographically Sortable Identifier):** Formed by a 48-bit timestamp (millisecond precision) and 80 bits of randomness, encoded in Base32.[1] Timestamp and random number generation, as well as Base32 encoding, can be performed in parallel per ID, making it a good candidate for batch generation.
- **KSUID (K-Sortable Unique Identifier):** Similar to ULID, but with a 32-bit timestamp (second precision relative to an epoch) and a 128-bit random payload. Random payload generation is highly parallelizable.
- **NanoID:** A NanoID is a fixed-length string composed of random characters from a custom alphabet, without a timestamp.[1] It is an excellent candidate for GPU acceleration if large batches are needed, as it purely involves random number generation and character mapping.
- **Snowflake ID:** Structured with a sign bit, 41-bit timestamp (milliseconds), 5-bit datacenter ID, 5-bit machine ID, and a 12-bit sequence number that resets every millisecond.[1] This is the most challenging algorithm for GPU. While the timestamp is manageable, datacenter/machine IDs require careful management (global constants or passed per thread group). The 12-bit sequence number, allowing 4096 IDs per millisecond per "machine," demands high-performance atomic increment operations and reset logic synchronized with the millisecond tick. This is the main difficulty.
- **CID (Content Identifier):** Composed of a multibase prefix, a multicodec ID, and a multihash (e.g., SHA256). The multihash calculation is the most computationally intensive part and can be significantly accelerated on GPU for bulk data. The remaining parts are metadata assembly. Feasibility depends on the ability to perform cryptographic hashes efficiently on the GPU.
- **CUID (Collision-Resistant Unique Identifier):** Includes the letter 'c', timestamp, a counter, a client "fingerprint," and a random block. The timestamp and random block are parallelizable. The counter requires atomic operations. The client fingerprint, if derived from host or GPU properties, needs specific handling.

The "sortability" feature of IDs like ULID, KSUID, and UUIDv7, while primarily a function of their structure, implies a subtle consideration for GPU generation. If large batches are generated, the creation order *within the same timestamp unit* (e.g., the same millisecond) can become non-deterministic unless explicitly managed. This could affect strict lexicographical sorting if not addressed. GPUs execute thousands of threads in parallel. If multiple threads generate an ID within the same timestamp "tick," their relative order is not guaranteed. The random part

of these IDs typically ensures uniqueness even within the same millisecond.[1] However, if a consumer expects IDs generated "simultaneously" by different GPU threads to have a subtle ordering based on thread ID or some other internal GPU factor (which is not part of the ID specification), this expectation might be violated. The specification only guarantees sortability based on the timestamp. For most use cases, randomness ensures distinction, and timestamp-based ordering is primary.

2.3. Comparative Table of Unique ID Algorithms for GPU Suitability

The following table summarizes the characteristics of ID algorithms and their potential suitability for GPU parallelization.

Algorithm	Primary Components	Timestamp Precision	Random Bits	Sequential/ Atomic Need	Cryptographic Need	GPU Parallelism Potential	Key GPU Challenges
UUIDv1/v6	Timestamp (60-bit, 100ns), Clock Seq., Node ID	100 nanoseconds	N/A (Node ID)	Medium (Clock Seq.)	Low	Medium	Node ID management, high-precision timestamp synchronization, atomic Clock Seq.
UUIDv4	122 random bits	N/A	~122	Low	Low (PRNG)	High	High-quality batch PRNG generation.
UUIDv7	Unix Timestamp (ms), random bits	Milliseconds	Variable	Low	Low (PRNG)	High	Timestamp synchronization, batch PRNG generation.
ULID	Timestamp (48-bit, ms), 80 random bits	Milliseconds	80	Low	Low (PRNG)	High	Timestamp synchronization, batch PRNG generation, parallel Base32 encoding.
KSUID	Timestamp (32-bit, s), 128 random bits	Seconds	128	Low	Low (PRNG)	High	Batch PRNG generation.
NanoID	Random chars from custom alphabet	N/A	Variable	Low	Low (PRNG)	High	Batch PRNG generation, character mapping.
Snowflake	Timestamp (41-bit, ms), Datacenter/Machine IDs, Seq (12-bit)	Milliseconds	N/A	Very High (Atomic Seq.)	Low	Medium	High-rate atomic sequence counter with synchronized reset, machine/DC ID management.
CID	Multibase, Multicodec, Multihash (e.g., SHA256)	N/A	N/A	Low	High (Crypto Hash)	Medium to High (Hash)	Efficient cryptographic hash (e.g., SHA256) implementation on GPU.
CUID	'c', Timestamp, Counter, Client Fingerprint, Random Block	Variable	Variable	Medium (Counter)	Low	Medium	Atomic counter, fingerprint management.

This table provides a concise and comparative overview, allowing a quick understanding of which algorithms are inherently more suitable for GPU acceleration based on their computational structure and dependencies on specific GPU primitives, which may be more or less challenging to implement.

3. Elixir Ecosystem for GPU Execution

Exploring GPU execution in Elixir is facilitated by an evolving set of tools and libraries, each with different levels of abstraction and control.

3.1. Nx and EXLA (XLA Backend): The Foundation for Numerical Elixir

Nx is Elixir's central library for numerical computing and multidimensional arrays, known as tensors, drawing parallels with Python's NumPy.[2] A key feature of Nx is its immutable tensor model.[2] To enable optimized compilation and execution of numerical code, Nx introduces Nx.Defn, a mechanism for writing Elixir code that can be Just-In-Time (JIT) compiled for various backends.[3]

The EXLA backend is crucial for GPU computing, as it uses Google's XLA (Accelerated Linear Algebra) to compile Nx.Defn code for execution on GPUs (NVIDIA CUDA and AMD ROCm) and TPUs.[11] Compilation target configuration is typically done via the XLA_TARGET environment

variable and specific client settings.[11]

In the context of ID generation, Nx and EXLA offer the following capabilities:

- **Numerical and Bitwise Operations:** Nx.Defn supports standard Elixir mathematical and bitwise operators (like &&& for bitwise AND and <<< for left shift) on tensors.[14] These are essential for packing and unpacking ID components.
- **Random Number Generation (Nx.Random):** This library, based on the Threefry PRNG, is designed for reproducibility and parallelization, making it suitable for the random parts of UUIDs, ULIDs, etc.[7] It's important to note that Nx.Random is a PRNG, not a CSPRNG, which has security implications for certain ID types.
- **Conditional Logic and Loops:** Nx.Defn.Kernel supports constructs like if/cond and while.[14] Limitations exist, such as the need for scalar predicates for if/cond, using while to simulate recursion, and the behavior of raise (which always triggers during compilation if present in any conditional branch [16]). The case statement is expanded at compile time.[16] These are crucial for implementing ID logics that may have branches, like Snowflake's counter reset.
- **Data Manipulation:** Functions like Nx.gather allow indexing and creating new tensors from existing ones [14], which could be useful for lookup table-like behaviors if parts of IDs are pre-computed or fixed.

Despite these capabilities, Nx/EXLA abstracts away direct kernel writing. Access to GPU-specific functionalities, like CUDA's clock64() or atomic operations beyond what XLA's HLO (High-Level Optimizer) exposes, is generally not available directly through Nx.Defn. The XLA HLO specification does not list general-purpose RMW (Read-Modify-Write) atomic operations like atomicAdd or atomicCAS.[18] While it mentions the Scatter operation, this exhibits non-deterministic behavior for overlapping writes if unique_indices is false, which is a significant concern for Snowflake's sequence counter. Similarly, XLA HLO does not natively support cryptographic hashes like SHA256.[18]

3.2. GPotion: An Embedded DSL for GPU Kernels in Elixir

GPotion is presented as an embedded Domain-Specific Language (DSL) for GPU programming in Elixir [19], aiming to simplify this process by allowing kernel-like definitions directly in Elixir.[20]

GPotion's main abstractions include:

- GMatrex: Represents an array residing in GPU memory, manipulable only by GPotion kernels.[21]
- GPotion Kernels: Functions that operate on GMatrex data on the GPU.
- Hok (Higher-Order Kernels): An extension to GPotion allowing GPU kernels to receive functions as arguments, promoting separation between coordination and computation code [21], aligning well with Elixir's functional paradigm.

GPotion aims to enable writing low-level GPU kernels [21], potentially offering more direct control than Nx/EXLA. The gpotion GitHub repository lists Elixir and CUDA among its languages [19], suggesting it compiles Elixir-like syntax to CUDA. Research indicates GPU execution requires memory management, kernel writing in CUDA/OpenCL, and optimizations [20], tasks GPotion seeks to abstract.

The critical question, largely unanswered by research materials due to scarce detailed documentation [19], is whether GPotion kernels can access:

- GPU-specific timers (e.g., CUDA's clock64()).
- GPU atomic operations.
- GPU-accelerated PRNG/CSPRNG libraries or primitives.
- Shared memory and thread synchronization primitives beyond what Hok might provide at a higher level.

3.3. Native Implemented Functions (NIFs) with Rustler or C: The Path to Full Control

NIFs are Elixir's mechanism for calling C/C++/Rust code directly, executing in the same BEAM process space.[4] This offers the lowest latency for interoperability. Through NIFs, it's possible to link CUDA/OpenCL runtime libraries, allowing direct calls to:

- Launch custom CUDA/OpenCL kernels.
- Utilize GPU libraries like cuRAND, cuBLAS, or custom cryptographic libraries.
- Explicitly manage GPU memory allocation and data transfers.
- Utilize CUDA atomic functions, clock64(), etc.

Rustler is a prominent library for writing safer NIFs in Rust, reducing the risk of crashing the BEAM.[5] The candlex project [25], for example, uses Rust and CUDA, indicating this is a practiced approach. Other examples of Rustler usage include rhai_rustler [26] and elixir-arrays_rrb_vector.[27]

NIF development is inherently more complex than pure Elixir programming. It requires knowledge of C/Rust and the GPU platform API. Memory management, error handling, and ensuring BEAM scheduler cooperation are critical.[4] Data marshalling between Elixir terms and C/Rust types suitable for the GPU is a key task; ErlNifBinary is mentioned for zero-copy of large binaries.[28] Examples like clex [29] (involving an Erlang NIF for OpenCL) and futlixir [30] (integrating Futhark with Elixir via NIFs) demonstrate this approach's viability. The rust-cuda-shim-example [31] shows Rust

calling CUDA kernels via a C shim, a pattern adaptable for Rustler NIFs.

3.4. Alternative: XLA Custom Calls (via EXLA)

Theoretically, XLA provides a CustomCall mechanism [32] to invoke external, pre-compiled C++ functions (and thus, potentially CUDA/OpenCL) from an HLO graph. The XLA FFI library [32] defines how these custom calls are linked and how data (buffers, attributes) are passed, supporting GPU stream passing for GPU backends.

Available documentation does not explicitly state whether EXLA currently exposes a way for Elixir developers to define and register XLA Custom Calls. This is a significant unknown. If EXLA could support this, it might offer a more integrated way than NIFs to inject custom GPU code into an Nx.Defn computation graph. The complexity of implementing XLA Custom Calls is comparable to NIFs, involving C++ and understanding XLA's buffer management and FFI specifics.[32] They could be used for operations not native to HLO, like specific cryptographic functions or custom atomic sequences. PyTorch/XLA, for instance, uses custom calls for Triton kernels.[34]

The choice between Nx/EXLA, GPotion, and NIFs represents a fundamental trade-off triangle: Ease of Use (Elixir-native, higher abstraction) versus Performance (granular control, direct hardware access) versus Feature Completeness (availability of specialized GPU primitives). Nx/EXLA offers high ease of use for tensor computations within the Elixir paradigm [3], but is limited by XLA HLO's expressiveness for specialized operations.[18] GPotion aims to bridge this gap by allowing Elixir-defined kernels [19], but its maturity and depth of features for low-level primitives are unclear. NIFs provide maximum performance and feature access [4], but with the highest development complexity and safety concerns.[4]

The success of JAX Pallas in providing Python access to GPU atomics [35] and Triton's ability to compile Python-like scripts to efficient GPU code, including atomics [37], set a precedent. If EXLA's XLA backend or a future Elixir GPU tool could adopt similar strategies (e.g., MLIR-based lowering that can target GPU atomic instructions, or integrating a Triton-like compiler), it could significantly lower the barrier to implementing complex ID algorithms without resorting to manual NIFs. Community interest in advanced GPU features, akin to Triton/Pallas, is suggested by related discussions and projects.[38] Furthermore, the general difficulty of GPU programming [20] means that the quality of documentation and community support for these Elixir GPU tools are as crucial as their technical capabilities. The sparse documentation for GPotion [19], for example, can be a significant barrier to adoption.

3.5. Evaluation Table of Elixir GPU Approaches for ID Generation Components

Essential GPU Primitive	Nx/EXLA (Direct Support)	GPotion (Potential Support)	NIFs (Rustler/C) (Capability)	XLA Custom Call (via EXLA) (Potential)
High-Precision Timestamping	Limited (likely via CPU)	Unknown (depends on ability to call clock64(), etc.)	High (direct access to clock64(), globaltimer)	High (if custom kernel can access GPU timers)
PRNG on GPU	High (Nx.Random)	Likely (could involve custom PRNG kernels)	High (cuRAND, OpenCL PRNG kernels, custom kernels)	High (custom kernel with PRNG)
CSPRNG on GPU	Low/None (Nx.Random is not CSPRNG)	Low (unlikely without external libraries or complex NIFs)	Medium to High (requires careful CPU seeding, or use of GPU CSPRNG libraries if they exist and are secure)	Medium to High (custom kernel, same seeding challenges as NIFs)
Bitwise Operations	High (supported by Nx.Defn)	High (expected in a kernel DSL)	High (full C/Rust/CUDA/OpenCL capability)	High (custom kernel)
Atomics (Increment, CAS)	Low (XLA HLO lacks generic RMW atomics)	Unknown (depends on DSL expressiveness and compiler)	High (direct access to CUDA/OpenCL atomicAdd, atomicCAS)	High (if custom kernel can use GPU atomics)
Cryptographic Hash (e.g. SHA256)	Low/None (not a standard HLO)	Low (unlikely without external libraries)	High (GPU crypto libraries, custom kernels)	High (custom kernel with SHA256)

This table maps the fundamental building blocks for ID algorithms to the capabilities and limitations of each Elixir GPU approach, clearly showing where Nx/EXLA or GPotion might suffice and where NIFs or XLA Custom Calls are likely indispensable.

4. Feasibility Assessment: Implementing ID Algorithm Components on GPU with Elixir

Effectively implementing ID algorithms on GPU using Elixir requires a detailed analysis of the feasibility of each key computational component.

4.1. Timestamp Generation on GPU

Obtaining high-precision, synchronized timestamps is challenging. GPU clocks, like clock() and clock64() in CUDA [40], are often cycle counters relative to GPU start, not a standard epoch like Unix epoch. The globaltimer register is another option in CUDA.[42]

- **Nx/EXLA:** Offers no direct access to GPU hardware timers. Timestamps would likely be passed from the CPU, limiting precision for in-kernel timing.
- **GPotion:** Feasibility depends on whether its kernel language can compile to include access to CUDA's clock64() or globaltimer and return these values. This is unclear from available materials.
- **NIFs:** Allow direct calls to CUDA's clock64() or using inline assembly for globaltimer.[42] The host can get the GPU's current clock rate (via cudaGetDeviceProperties() [41], though clockRate is deprecated) or perform calibration. Conversion to wall-clock time requires a CPU reference point and careful synchronization. torch.cuda.Event(enable_timing=True) [43] is used for measuring elapsed time, not absolute timestamps for ID generation.
- **Synchronization:** For sortable IDs (ULID, Snowflake), if timestamps are generated on the GPU, ensuring they are consistent with host time or a global time source is complex. __syncthreads() is block-local [44]; global synchronization is harder. Passing a host timestamp at kernel launch might be more practical for batch operations, with the GPU handling sub-millisecond components if needed.
- **Precision:** UUIDv1 requires 100ns precision.[1] CUDA's clock64() offers cycle-level, convertible to nanoseconds if the clock rate is known.

4.2. Random Number Generation on GPU

- **PRNG with Nx/EXLA:** Nx.Random.uniform/normal/randint [7] uses the Threefry algorithm and can generate large batches of random numbers efficiently on GPU via EXLA. It's suitable for the non-cryptographic parts of UUIDs, ULIDs, KSUIDs, and NanoIDs. Nx.Random.key/1 and split/2 are used to manage PRNG state.[7]
- **CSPRNGs:**
  - **Challenge:** CSPRNGs need a good entropy source and mixing, which is hard to replicate on GPU. Erlang's :crypto.strong_rand_bytes/1 [8] depends on the OS and OpenSSL.
  - **Nx/EXLA:** Nx.Random is not a CSPRNG.
  - **GPotion:** Unlikely to provide CSPRNGs directly, unless it links to a custom library.
  - **NIFs:** Best approach for CSPRNGs. Could:
    1. Generate CSPRNG bytes on CPU (using :crypto.strong_rand_bytes/1) and transfer to GPU (with overhead).
    2. Use a GPU-accelerated crypto library with CSPRNG features (e.g., Sobol or other quasi-random generators from cuRAND, if suitable, or if cuRAND has CSPRNG modes not detailed in materials). The MWC64X generator is mentioned for OpenCL, being a PRNG.[46]
    3. Implement a known CSPRNG algorithm in a custom kernel, carefully seeded with CPU-generated entropy. This is complex.
  - **Security Implication:** If an ID requires truly cryptographic randomness, relying solely on typical GPU PRNGs is insufficient. The feasibility of implementing CSPRNGs on GPU via NIFs depends on how entropy is managed. If NIFs call cuRAND, its internal seeding and quality determine security. If a custom CSPRNG kernel is written, securely seeding it from the CPU (e.g., with bytes from :crypto.strong_rand_bytes/1) and ensuring periodic reseeding without compromising performance or security is a non-trivial distributed systems problem mapped to a single device.
- **Throughput:** GPU PRNGs (like cuRAND) are known for very high throughput (GB/s of random numbers).[47]

4.3. Bitwise Operations and Data Encoding/Decoding on GPU

- **Nx/EXLA:** Nx.Defn supports all standard Elixir bitwise operators (&&&, |||, ^^^, <<<, >>>, ~~~), which are translated by EXLA for GPU execution.[14] This is well-suited for packing/unpacking timestamp, counter, and random components of IDs.
- **GPotion/NIFs:** Custom kernels in GPotion (if sufficiently expressive) or NIFs (CUDA/OpenCL) can implement any bitwise logic. This would be necessary for complex custom encoding/decoding that isn't easily vectorizable with Nx operations (e.g., efficient Base32/Base58 for a large batch of IDs).
- **Parallelism:** Bitwise operations and encoding are typically highly parallelizable per ID.

4.4. Atomic Operations for Sequencing (e.g., Snowflake Counters) on GPU

Snowflake's 12-bit sequence number, resetting every millisecond, needs an atomic increment and a coordinated reset mechanism among threads/blocks working on the same "machine ID".[1]

- **Nx/EXLA (XLA HLO):**
  - Direct HLO support for generic RMW (Read-Modify-Write) atomic operations like atomicAdd or atomicCAS is not apparent in XLA's operation semantics.[18]
  - The HLO Scatter can perform updates with a computation, but the "order in which updates are applied is non-deterministic" for

overlapping indices [18], making it unsuitable for reliable counters without additional guarantees. This is a major limitation for implementing Snowflake purely in Nx.Defn if EXLA relies only on standard HLOs.

- **JAX Pallas and Triton Precedent:**
  - JAX Pallas exposes atomic operations like atomic_add and atomic_cas for GPU.[35] This is crucial as it implies XLA *can* compile to GPU atomics, or Pallas generates lower-level code (e.g., Triton or direct MLIR with atomic ops).
  - Triton also supports atomics like tl.atomic_add(), which map to CUDA atomics.[37]
  - If EXLA could adopt a Pallas-like strategy (e.g., allowing certain Nx.Defn patterns to be lowered to XLA representations that then become GPU atomics, or integrating a Triton-like path), this would be transformative. Community discussions [38] suggest interest in advanced GPU features like Triton/Pallas.
- **GPotion:** Feasibility depends on whether GPotion's DSL can express or compile to GPU atomic instructions. Unknown from materials.
- **NIFs:** The most direct way to ensure access to CUDA (atomicAdd(), atomicCAS(), atomicInc() [45]) or OpenCL atomics. This gives full control but adds NIF complexity.
- **Synchronization for Reset:** Resetting the counter every millisecond across potentially many GPU blocks requires careful synchronization. This might involve a global flag in GPU memory, checked by blocks, and potentially a dedicated kernel or master block to manage the timestamp and signal resets. __syncthreads() is block-local; true global synchronization is difficult.[44]
- **Performance:** Native CUDA atomics are fast but can become contention points if many threads hit the same counter.[50]

The "atomicity" requirement for Snowflake's sequence numbers isn't just about atomicAdd. It's about a *conditional atomic sequence*: increment if current_ms matches, otherwise reset and increment. This compound operation, if not directly supported as a single atomic HLO or GPU intrinsic, might require a loop with atomicCAS in a NIF or custom kernel, potentially impacting performance under high contention. A GPU kernel would need to read the current GPU time, compare it to the timestamp associated with the counter's current value, and if it's the same millisecond, use atomicAdd; if it's a new millisecond, use atomicCAS to reset the counter and store the new millisecond, or a master thread would signal the reset.

4.5. Cryptographic Hashing on GPU (e.g., SHA256 for CIDs)

- **Nx/EXLA (XLA HLO):** Standard XLA HLOs do not include cryptographic hash functions like SHA256.[18]
- **GPotion:** Unlikely to have built-in SHA256, unless it links to a custom library or allows embedding raw CUDA for it.
- **NIFs:** The most viable route. Link against existing CUDA SHA256 libraries or implement SHA256 in a custom CUDA kernel. This is a common GPGPU task.
- **XLA Custom Calls:** A strong alternative if EXLA supports them. A C++ function implementing SHA256 (possibly calling a CUDA library) could be registered as a custom call.[32]
- **Throughput:** GPUs are very efficient at SHA256 for bulk data.[48]

If an ID algorithm can be decomposed such that only a small, critical part requires NIFs (e.g., just the atomic counter for Snowflake), while the rest (timestamping, bit manipulation, PRNG) can be handled by Nx/EXLA or GPotion, a hybrid approach might offer the best balance of development effort and performance. This would minimize the NIF code surface, leveraging Elixir's high-level tools for most of the logic.

4.6. Feasibility Matrix for GPU-Accelerated ID Components with Elixir Approaches

Essential GPU Primitive	Nx/EXLA (Direct)	Nx/EXLA (via hypothetical advanced XLA features)	GPotion	NIFs (CUDA/OpenCL)	XLA Custom Call (via EXLA)
High-Precision Timestamp	Low (CPU-dependent)	Medium (if XLA exposes timers)	Unknown	High (clock64())	High (custom kernel)
PRNG on GPU	High (Nx.Random)	High	Probable	High (cuRAND, etc.)	High (custom kernel)
CSPRNG on GPU	None	Low	Low	Medium (requires CPU seeding, complex)	Medium (same challenges as NIFs)
Bitwise Operations	High	High	High	High	High (custom kernel)
Atomic Increment	Low (XLA HLO limited)	Medium (if Pallas/Triton-inspired)	Unknown	High (atomicAdd)	High (custom kernel)
Atomic CAS	Low (XLA HLO limited)	Medium (if Pallas/Triton-inspired)	Unknown	High (atomicCAS)	High (custom kernel)
SHA256 Hash	None	Low (via Custom Call)	Low	High (libs/custom kernels)	High (custom kernel)
Overall Feasibility	**Low to Medium** (for simple, random IDs)	**Medium to High** (if features evolve)	**Unknown to Medium** (depends on maturity & features)	**High** (for all components, but high complexity)	**Medium to High** (if EXLA supported, complexity similar to NIFs for kernel itself)
Key Considerations/Challenges	Limited by XLA HLO; ideal for vectorizable parts.	Depends on EXLA/XLA evolution.	Sparse documentation ; low-level capabilities uncertain.	Higher development complexity, BEAM safety risks, data marshalling.	Requires EXLA support; C++ and XLA FFI complexity.

This matrix provides a granular assessment of how each Elixir GPU approach can handle the fundamental building blocks of ID algorithms, directly addressing the "feasibility" part of the user's query at a component level and identifying where Elixir's GPU ecosystem is strong and where it has gaps for this specific domain.

5. Performance Considerations, Benchmarking Insights, and Trade-offs

Accelerating ID generation with GPUs involves a series of performance considerations that go beyond the simple parallelizability of algorithms.

5.1. Data Transfer Overheads (CPU-GPU-CPU)

Moving data between host (CPU) memory and device (GPU) memory is often a primary bottleneck for GPU acceleration, especially for small, latency-sensitive tasks like generating a single ID.[39] The Nx backend has transfer mechanisms [14], but the inherent latency of these transfers must be considered.

5.2. Kernel Launch Latency

Launching a kernel on the GPU incurs an intrinsic latency.[39] If IDs are generated one by one, this latency can nullify any computational gain.

5.3. Batching Strategies for Amortization

Generating IDs in large batches is crucial to amortize data transfer and kernel launch overheads.[39] The optimal batch size will depend on the specific ID algorithm, GPU hardware, and application requirements. The "sweet spot" for GPU-accelerated ID generation will likely be in applications needing large *batches* of IDs with relaxed latency per individual ID, rather than systems needing single IDs with extremely low

latency. If an application can request, say, 1 million IDs to be generated at once, the per-ID cost of overheads becomes negligible.

5.4. Throughput Analysis of Essential GPU Primitives

- **Atomic Operations:** Native CUDA atomic operations, like atomicAdd and atomicCAS, are highly optimized but can become contention points if many threads access the same counter simultaneously.[5048]
- **Cryptographic Hashing (SHA256):** GPUs excel at parallel hashing. Typical throughput rates can be on the order of GB/s or millions of hashes per second.[48]
- **Random Number Generation (PRNG/CSPRNG):** Libraries like cuRAND offer very high throughput for PRNGs.[47] CSPRNG generation might be slower due to more complex algorithms or the need to fetch entropy.
- **Timestamp Acquisition:** Reading clock64() is a very low-latency operation (a few cycles). The main cost is in organizing this data and synchronizing it.

5.5. Development Complexity vs. Performance Gain

It's essential to evaluate the trade-off: Nx/EXLA (easier, potentially less performant for complex logic) versus GPotion (medium complexity, performance dependent on its capabilities) versus NIFs (harder, highest performance potential and feature access). The learning curve for Elixir developers venturing into GPU-specific NIFs or XLA Custom Calls is a significant factor.

5.6. Resource Management

- **GPU Memory Usage:** Storing state for many parallel ID generators (e.g., individual PRNG states, counters).
- **GPU Core Utilization:** Ensuring sufficient parallel work to keep the GPU busy.[57] Batching aids this.
- **Impact on BEAM Schedulers:** If NIFs are long-running (though GPU NIFs typically offload work and return quickly, the NIF call itself should be fast [4]).

The choice of ID algorithm strongly influences the batching strategy and potential for GPU saturation. Simpler, stateless IDs like NanoID or UUIDv4 are easier to batch and can achieve higher parallelism with less complex kernel logic than stateful, multi-component IDs like Snowflake. The latter requires a shared atomic counter and coordination around millisecond ticks [1], which can create contention and limit scalability within the batch. A failed attempt to port a complex algorithm (like Snowflake) to GPU due to insurmountable atomicity/synchronization challenges with current Elixir tools could lead to incorrectly dismissing GPUs for *all* ID generation. A nuanced understanding of which algorithms fit the GPU model (and Elixir's access to it) is crucial.

6. Recommendations and Future Directions

Based on the analysis of algorithm suitability, Elixir ecosystem capabilities, and performance considerations, the following recommendations and future directions can be outlined.

6.1. Algorithm-Specific Recommendations

- **High Potential (with Batching):**
  - **NanoID, UUIDv4 (random parts):** These are the strongest candidates for GPU acceleration using Nx/EXLA, specifically Nx.Random for bulk pseudorandom number generation and Nx.Defn for bitwise operations and character mapping. The absence of complex state or high-precision GPU timestamp requirements simplifies implementation.
  - **ULID, KSUID, UUIDv7 (random parts and timestamping if host-provided or with simple GPU timers):** Similarly, random generation and bitwise manipulation portions are suitable for Nx/EXLA. If the timestamp is managed by the CPU and passed to the kernel in batch, or if millisecond precision obtained simply on the GPU is sufficient, feasibility is high. Encoding (e.g., Base32 for ULID) can be parallelized.
- **Medium Potential (Complex, NIFs/CustomCalls likely needed for parts):**
  - **Full ULID/KSUID/UUIDv7 (if GPU timestamping and custom encoding are needed):** If high-precision timestamps directly on the GPU or highly optimized encoding algorithms non-trivial with Nx are required, NIFs or XLA Custom Calls might be necessary for these parts.
  - **CUID:** If the counter and "client fingerprint" logic require complex atomic operations or access to system information unavailable via Nx, NIFs would be the approach.
- **Challenging (Significant NIF/CustomCall effort for core logic):**
  - **Snowflake:** The high update rate of the atomic sequence counter (4096/ms) and millisecond-synchronized reset logic make this the most challenging. Currently, this would require NIFs for direct access to GPU atomics (atomicAdd/atomicCAS) and precise time management.
  - **CID:** Cryptographic hash calculation (e.g., SHA256) is not natively supported by Nx/EXLA. This would require a NIF utilizing a GPU crypto library or a custom CUDA/OpenCL kernel for hashing, or an XLA Custom Call if EXLA supports it.

6.2. Recommended Elixir GPU Strategies

- **For highly parallel, array-oriented parts (PRNG, bitwise ops):** Nx/EXLA is the first choice due to ease of use and Elixir integration.
- **For custom kernel logic with potential low-level access (if mature):** GPotion could be an option, but its current capabilities for atomics, CSPRNGs, and precise timers need verification beyond available materials. Lack of detailed documentation is an impediment.[19]
- **For indispensable low-level primitives (atomics, CSPRNGs, crypto, direct CUDA intrinsic access):** NIFs (with Rustler for Rust safety, or C) are currently the most reliable and highest-control path.
- **For integrating C++/CUDA operations into the XLA flow (if EXLA supports):** XLA Custom Calls are a powerful, albeit complex, alternative to NIFs.[32]

The "best" approach is highly context-dependent: the specific ID algorithm, required throughput, latency tolerance, security needs, and development team experience with GPU programming and NIFs. There is no single answer.

6.3. Hybrid CPU + GPU Approaches

It is suggested to offload only the most compute-intensive and parallelizable parts to the GPU (e.g., bulk random number generation, bulk hashing for CIDs). Other parts (e.g., complex state management for Snowflake, final ID assembly if trivial) could remain on the CPU, executed in Elixir. This can offer a good balance between performance and development complexity.

6.4. Areas for Further Research and Elixir Ecosystem Development

- **EXLA:** Investigate exposing more XLA capabilities, potentially inspired by JAX Pallas [35], for atomic operations or easier integration of custom GPU code/Triton kernels.[37] This could bridge the gap between Nx.Defn and NIFs. Community interest in such features is notable.[38]
- **GPotion:** Clarify documentation and present examples for accessing low-level GPU primitives (timers, atomics, shared memory, PRNG/CSPRNG interfaces).
- **Standard Libraries:** Development of higher-level Elixir libraries encapsulating NIF-based GPU primitives for common tasks, like CSPRNGs or atomic counters, would make these features easier to consume. Investing in NIFs for GPU-accelerated ID generation could create reusable components (e.g., a generic GPU atomic counter service, a GPU CSPRNG NIF) that would benefit other parts of an Elixir application or the broader Elixir ecosystem.
- **Benchmarking:** More comprehensive benchmarks of these approaches within the Elixir ecosystem are needed to quantify actual performance gains.

Investigating GPU for IDs can highlight current limitations in Elixir's GPU tooling that, if addressed by the community or core maintainers (e.g., better atomic support in EXLA, more mature GPotion), could unlock GPU acceleration for a wider range of Elixir applications, fostering a richer HPC ecosystem around Elixir.

7. Conclusion

7.1. Summary of Findings

The feasibility of refactoring or reimplementing unique ID generation algorithms in Elixir for GPU execution is multifaceted and critically depends on the specific algorithm and the current capabilities of the Elixir GPU ecosystem.

- **Algorithms with a high random component and simple bitwise operations (NanoID, UUIDv4, parts of ULID/KSUID/UUIDv7)** present the highest potential for acceleration using Nx/EXLA, primarily due to Nx.Random and bitwise operation support in Nx.Defn. Batch generation is essential to amortize overheads.
- **Algorithms requiring complex state, high-frequency atomic counters (Snowflake), or cryptographic hashes (CID)** face significant challenges with current high-level Elixir tools. Nx/EXLA, in its current form based on standard XLA HLO, lacks direct support for robust RMW atomic operations and cryptographic hashes.
- **GPotion** remains an unknown in terms of its ability to provide access to low-level GPU primitives needed for more complex algorithms, due to limited documentation.
- **NIFs (using C or Rust with Rustler)** emerge as the most flexible and powerful approach for accessing the full range of GPU functionalities, including atomics, precise timers, and crypto libraries. However, they introduce greater development complexity and potential risks to BEAM stability.
- **XLA Custom Calls** represent a theoretically promising avenue for integrating custom GPU code within the EXLA flow, but their support and usability from Elixir are not clearly documented.

Key challenges identified include CPU-GPU data transfer overhead, kernel launch latency, and the critical need for access to low-level GPU primitives, especially atomic operations for algorithms like Snowflake and cryptographically secure pseudorandom number generators (CSPRNGs) for IDs requiring high security.

7.2. Final Thoughts on Practical Implications

The decision to use GPUs for ID generation in Elixir should be guided by a careful analysis of specific application requirements and rigorous benchmarking of a prototype against an optimized CPU-based solution in Elixir. The allure of GPU parallelism should not overshadow practical overheads and development costs.

Currently, the benefits of GPU acceleration for IDs in Elixir are most likely to be realized in scenarios involving **generating large batch volumes of simpler, random-heavy IDs**. For complex, stateful, or crypto-dependent algorithms, implementation hurdles are considerable and may require substantial investment in NIF development.

This investigation serves as a case study for Elixir's broader applicability in high-performance computing (HPC). The challenges and solutions identified for ID generation – such as state management, access to low-level hardware features, and the balance between abstraction and control – are relevant to other domains considering Elixir for GPU computing. The Elixir GPU ecosystem is evolving, and future developments in tools like EXLA (potentially incorporating features inspired by JAX Pallas or Triton) and GPotion may, in the future, reduce reliance on NIFs and make GPU acceleration more accessible for a wider range of computationally intensive tasks in Elixir.

Referências citadas

1. UUID, ULID, NanoIDs and Snowflake IDs , What's the difference? - Hewi's Blog, acessado em maio 29, 2025, https://hewi.blog/uuid-ulid-nanoids-and-snowflake-ids-whats-the-difference
2. Nx for Absolute Beginners - DockYard, acessado em maio 29, 2025, https://dockyard.com/blog/2022/03/15/nx-for-absolute-beginners
3. Numerical Elixir (Nx) - GitHub, acessado em maio 29, 2025, https://github.com/elixir-nx
4. Speeding up Elixir: integration with native code (NIF, Ports, etc.) - DEV Community, acessado em maio 29, 2025, https://dev.to/adamanq/speeding-up-elixir-integration-with-native-code-nif-ports-etc-5ajd
5. Embedding Python in Elixir, it's fine | Hacker News, acessado em maio 29, 2025, https://news.ycombinator.com/item?id=43171239
6. The Unique Features of Snowflake ID and its Comparison to UUID - Software Mind, acessado em maio 29, 2025, https://softwaremind.com/blog/the-unique-features-of-snowflake-id-and-its-comparison-to-uuid/
7. Nx.Random — Nx v0.9.2 - HexDocs, acessado em maio 29, 2025, https://hexdocs.pm/nx/Nx.Random.html
8. Cryptographically secure pseudorandom number generator - Wikipedia, acessado em maio 29, 2025, https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator
9. crypto v5.6 - Erlang, acessado em maio 29, 2025, https://www.erlang.org/doc/apps/crypto/crypto.html
10. crypto — crypto v5.6 - Erlang, acessado em maio 29, 2025, https://www.erlang.org/doc/man/crypto.html
11. EXLA v0.9.2 - HexDocs, acessado em maio 29, 2025, https://hexdocs.pm/exla/
12. Understanding the Elixir Machine Learning Ecosystem - The Stack Canary, acessado em maio 29, 2025, https://www.thestackcanary.com/understanding-the-elixir-machine-learning-ecosystem/
13. EXLA - 0.9.2 - HexDocs, acessado em maio 29, 2025, https://hexdocs.pm/exla/EXLA.epub
14. Nx.Defn — Nx v0.9.2 - HexDocs, acessado em maio 29, 2025, https://hexdocs.pm/nx/Nx.Defn.html
15. Generating random unit vectors in Elixir Nx | jyc - Jonathan Y. Chan, acessado em maio 29, 2025, https://www.jonathanychan.com/blog/generating-random-unit-vectors-in-elixir-nx/
16. Nx.Defn.Kernel — Nx v0.9.2 - HexDocs, acessado em maio 29, 2025, https://hexdocs.pm/nx/Nx.Defn.Kernel.html
17. Nx — Nx v0.9.2 - HexDocs, acessado em maio 29, 2025, https://hexdocs.pm/nx/Nx.html#gather/3
18. Operation semantics | OpenXLA Project, acessado em maio 29, 2025, https://www.tensorflow.org/xla/operation_semantics
19. GPotion - An embedded DSL for GPU programming in Elixir - GitHub, acessado em maio 29, 2025, https://github.com/researchanon/gpotion
20. GPotion: Embedding GPU programming in Elixir | Request PDF - ResearchGate, acessado em maio 29, 2025, https://www.researchgate.net/publication/388822174_GPotion_Embedding_GPU_programming_in_Elixir
21. GPotion: An embedded DSL for GPU programming in Elixir | Request PDF - ResearchGate, acessado em maio 29, 2025, https://www.researchgate.net/publication/375245367_GPotion_An_embedded_DSL_for_GPU_programming_in_Elixir
22. acessado em dezembro 31, 1969, https.researchgate.net/publication/375245367_GPotion_An_embedded_DSL_for_GPU_programming_in_Elixir
23. acessado em dezembro 31, 1969, https://hexdocs.pm/gpotion/GPotion.html
24. at main · mikeroyal/Developer-Handbook - GitHub, acessado em maio 29, 2025, https://github.com/mikeroyal/Developer-Handbook?search=1
25. mimiquate/candlex: An Nx backend for candle machine learning framework - GitHub, acessado em maio 29, 2025, https://github.com/mimiquate/candlex
26. rhaiscript/rhai_rustler: Elixir NIF bindings for Rhai, an embedded scripting language and engine for Rust - GitHub, acessado em maio 29, 2025, https://github.com/rhaiscript/rhai_rustler
27. elixir-arrays_rrb_vector/lib/arrays_rrb_vector.ex at main · Qqwy/elixir-arrays_rrb_vector - GitHub, acessado em maio 29, 2025, https://github.com/Qqwy/elixir-arrays_rrb_vector/blob/master/lib/arrays_rrb_vector.ex
28. elixir-nx/fine: C++ library enabling more ergonomic NIFs, tailored to Elixir - GitHub, acessado em maio 29, 2025, https://github.com/elixir-nx/fine
29. The Clex Elixir package wraps the excellent Erlang NIF provided by tonyrog/cl - GitHub, acessado em maio 29, 2025, https://github.com/arpieb/clex
30. Munksgaard/futlixir - GitHub, acessado em maio 29, 2025, https://github.com/Munksgaard/futlixir
31. An example of a rust binary that calls CUDA kernels via a C shim - GitHub, acessado em maio 29, 2025, https://github.com/boustrophedon/rust-cuda-shim-example
32. XLA Custom Calls | OpenXLA Project, acessado em maio 29, 2025, https://openxla.org/xla/custom_call
33. acessado em dezembro 31, 1969, https://openxla.org/xla/custom_calls
34. Custom GPU Kernels via Triton — PyTorch/XLA master documentation, acessado em maio 29, 2025, https://docs.pytorch.org/xla/release/r2.7/features/triton.html

35. Pallas Quickstart - JAX documentation, acessado em maio 29, 2025, https://docs.jax.dev/en/latest/pallas/quickstart.html
36. jax.experimental.pallas module - JAX documentation, acessado em maio 29, 2025, https://docs.jax.dev/en/latest/jax.experimental.pallas.html
37. Demystify OpenAI Triton - fkong' tech blog, acessado em maio 29, 2025, https://fkong.tech/posts/2023-04-23-triton-cuda/
38. Pallas implementation of attention doesn't work on CloudTPU · Issue #18590 · jax-ml/jax, acessado em maio 29, 2025, https://github.com/google/jax/issues/18590
39. Axon.Serving: Model Serving with Axon and Elixir - DockYard, acessado em maio 29, 2025, https://dockyard.com/blog/2022/10/17/axon-serving-model-serving-with-axon-and-elixir
40. Clock() and Clock64() Functions - CUDA Programming and Performance - NVIDIA Developer Forums, acessado em maio 29, 2025, https://forums.developer.nvidia.com/t/clock-and-clock64-functions/285673
41. Timing Thread Execution on CUDA - Reddit, acessado em maio 29, 2025, https://www.reddit.com/r/CUDA/comments/10i1hvq/timing_thread_execution_on_cuda/
42. How to use the %%globaltimer register in CUDA? - Stack Overflow, acessado em maio 29, 2025, https://stackoverflow.com/questions/77771384/how-to-use-the-globaltimer-register-in-cuda
43. How to Accurately Time CUDA Kernels in Pytorch - Speechmatics, acessado em maio 29, 2025, https://www.speechmatics.com/company/articles-and-news/timing-operations-in-pytorch
44. Possible way to do block synchronization in CUDA kernels - Stack Overflow, acessado em maio 29, 2025, https://stackoverflow.com/questions/42696117/possible-way-to-do-block-synchronization-in-cuda-kernels
45. Synchronize all blocks in CUDA - NVIDIA Developer Forums, acessado em maio 29, 2025, https://forums.developer.nvidia.com/t/synchronize-all-blocks-in-cuda/27774
46. MWC64X - Uniform random number generator for OpenCL., acessado em maio 29, 2025, https://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html
47. Tag: cuRAND | NVIDIA Technical Blog - NVIDIA Developer, acessado em maio 29, 2025, https://developer.nvidia.com/blog/tag/curand/
48. arXiv.org e-Print archive, acessado em maio 29, 2025, https://arxiv.org/
49. acessado em dezembro 31, 1969, https://arxiv.org/search/?query=GPU+PRNG+cuRAND+clRNG+throughput+benchmark&searchtype=all&source=header
50. Reducing Synchronization and Communication Overheads in GPUs by Preyesh Dalmia A dissertation for the degree of Doctor of Philos - University of Wisconsin–Madison, acessado em maio 29, 2025, https://asset.library.wisc.edu/1711.dl/6XSM6YXB5MQSH8Z/R/file-e72a2.pdf
51. acessado em dezembro 31, 1969, https://developer.nvidia.com/blog/tag/atomic-operations/
52. NVIDIA On-Demand, acessado em maio 29, 2025, https://www.nvidia.com/gtc/on-demand/
53. NVIDIA Technical Blog - NVIDIA Developer, acessado em maio 29, 2025, https://developer.nvidia.com/blog/
54. acessado em dezembro 31, 1969, https://gist.github.com/search?q=CUDA+atomicAdd+benchmark
55. acessado em dezembro 31, 1969, https://www.researchgate.net/search.Search.html?query=GPU%20SHA256%20throughput%20benchmark
56. acessado em dezembro 31, 1969, https://arxiv.org/search/?query=GPU+SHA256+throughput+benchmark&searchtype=all&source=header
57. Tracking system resource (GPU, CPU, etc.) utilization during training with the Weights & Biases Dashboard - Lambda, acessado em maio 29, 2025, https://lambda.ai/blog/weights-and-bias-gpu-cpu-utilization