# 高度なAI計算基盤におけるAOTritonの処理原理とFlashAttentionの比較、およびRyzen 7 8700G環境でのソースビルドの有効性に関する詳細解析

## 1. 序論

現代のオープンソースを基盤とする深層学習エコシステムは、歴史的な経緯からNVIDIAのCUDAアーキテクチャを第一の標的として発展してきた。PyTorch、xFormers、FlashAttention、そして近年急速に普及しているTriton言語といった中核的なAIパッケージ群は、主としてNVIDIAのハードウェア上で記述され、テストされているのが現状である[1]。このハードウェアとソフトウェア間の技術的非対称性を解消し、AMD製GPU（CDNAおよびRDNAアーキテクチャ）上で同等のパフォーマンスとスケーラビリティを実現するため、AMDはROCm（Radeon Open Compute）プラットフォームを展開してきた。しかしながら、ROCmプラットフォームにおける巨大なソフトウェア依存関係ツリーの管理は、極めて複雑な課題を伴う。PyTorchの実行時、特に動的コンパイル（Just-In-Time: JIT）を前提とするTritonカーネルの実行においては、実行時のコンパイルレイテンシ（ジッター）の増加や、コンパイラキャッシュの不確実性に起因する「グラフブレイク（Graph Breaks）」が頻発し、安定運用に対する重大な懸念材料となっていた[1]。さらに、PyTorch、Tritonコンパイラ、そしてROCmドライバ間の厳格なバージョン結合は「依存関係の地獄（Dependency Hell）」を引き起こし、わずかな上流の変更がシステム全体を機能不全に陥らせるリスクを内包している[1]。

これらの課題を根本的に解決し、ROCm環境での推論および学習パフォーマンスを安定化させるための橋渡しとして開発されたのが「AOTriton（Ahead of Time Triton Math Library）」である[2]。本レポートでは、深層学習コンパイラの視点からAOTritonの内部的な処理原理を解剖し、それが実装・提供の対象とする画期的なアルゴリズムである「FlashAttention」との根本的な概念の相違について体系的に解析する。さらに、公式のバイナリサポート対象外である統合型GPU（iGPU）搭載のAPU「AMD Ryzen 7 8700G（Radeon 780M / gfx1103）」環境での利用を想定し、AOTritonをソースコードからビルドした場合に得られるパフォーマンス上の有効性、最適化のメカニズム、およびシステム全体に及ぼす影響について詳細な検証を行う。

## 2. AOTritonのアーキテクチャと内部処理原理

### 2.1 AOT（事前コンパイル）アプローチへの転換とその背景

通常、PyTorchなどのフレームワーク上でTriton言語を使用する場合、カーネルは実行環境においてJITコンパイラによって動的にコンパイルされ、GPUの実行バイナリへと変換される。この手法はハードウェアの柔軟な抽象化に寄与する一方で、ROCm環境においては前述したコンパイルジッターや予期せぬ実行遅延を引き起こす要因となっていた[2]。

AOTritonは、この問題を解決するために設計されたC++ベースの数学ライブラリである。頻繁に使用される極めて重要な数学カーネル（現在はTri Dao氏のアルゴリズムに基づくFlashAttentionがその

筆頭である）を、実行時ではなくビルド時（Ahead of Time）にTritonコンパイラを用いて事前にコンパイルし、静的ライブラリ（.a）または動的ライブラリ（.so）の形式でパッケージ化する[3]。この手法により、実行環境からTritonコンパイラ本体への依存が切り離され、PyTorchのScaled Dot-Product Attention（SDPA）バックエンドから直接、事前コンパイル済みのバイナリを呼び出すことが可能となる[3]。結果として、実行時のオーバーヘッドが完全に排除される。

## 2.2 ビルドパイプラインとディスパッチ機構のメカニズム

AOTritonの処理原理は、単なるコードの静的翻訳ではなく、対象となるハードウェアごとに最適なアセンブリを選択するための高度な自動生成とディスパッチャ（実行経路の振り分け）機構に依存している。

ビルドプロセスは非常に厳格な環境要件を持つ。Python 3.10以上（Python 3.14以降ではast.Numエラーに対処するためのパッチが必要）、GCC 8以上またはClang 10以上（C++のDesignated initializersを利用するため）、CMake 3.26以上、Ninja 1.11以上、およびLZMA圧縮を処理するためのliblzmaパッケージなどの依存関係が必須となる[3]。コンパイルパイプラインは以下のような多層的なプロセスを経て進行する。

第一の段階はカーネルの定義と生成である。rules.pyと呼ばれる汎用的なフレームワークファイルにカーネルの定義が記述されており、ビルドシステムはこれを読み込んで、サポート対象となるGPUアーキテクチャ（target_gpusフラグで指定される）や、データ型（FP16、BF16、FP32）、マスクの有無（Causal等）のあらゆる組み合わせに対応する膨大な数のC++シム（Shim）コードを自動生成する[3]。

第二の段階は、ハイパーパラメータの最適化とデータベース化である。Tritonカーネルの実行性能は、スレッドブロックのサイズやステージ数といったハードウェアごとの物理的制約に強く依存する。AOTritonの開発フローにおいては、クリーンな環境でtune_flash.pyなどのチューニングスクリプトを実行し、最適なパラメータ構成を探索する[5]。極端なチューニングパラメータはGPUのセグメンテーションフォールトやシステムリセットを引き起こす危険性があるため、保守的かつ精緻な探索が行われ、その結果はtuning_database.sqlite3というSQLiteデータベースに保存される[5]。ビルド時には、このデータベースの最適解が純粋なC++のルックアップテーブル（LUT）として埋め込まれるため、実行時に再計算を行う必要がない[3]。

第三の段階が、実行時のC++ディスパッチャによる解決である。PyTorchのバックエンドから関数の呼び出し（例：attn_fwd API）が行われると、生成されたC++コードであるshim.attn_fwd.cc内のディスパッチャが、現在のGPUストリーム、テンソルの形状、指定されたデータ型などの条件を評価する[5]。そして、ハードコードされたLUTを参照し、条件に完全に合致する最適なアセンブリ（hsacoオブジェクト）を瞬時に選択してGPUへと送信する[5]。

## 2.3 PyTorchスタックとの厳密な互換性マトリクス

AOTritonは独立して機能するライブラリではなく、PyTorchのコンパイルプロセス中に自動的にダウンロードされ、SDPAカーネルのバックエンドとしてPyTorchのバイナリ（Wheel）に直接組み込まれる形で提供される[3]。しかし、AOTritonのFlashAttention APIは進化のスピードが速く、シグネチャや機能が頻繁に変更されるため、特定のPyTorchリリースは特定のAOTritonのバージョンとしか互換性を持たないという強固なバージョン結合が存在する[1]。

以下の表は、PyTorchの公式アップストリームリリースと、ROCm固有のフォークリリースにおけるAOTritonの互換性マトリクスを示している。

| PyTorchバージョン | AOTritonバージョン (Upstream) | AOTritonバージョン (ROCm Fork) | 運用上の特筆事項および制約 |
|---|---|---|---|
| **2.9 (Upcoming)** | 0.11b, 0.10b | 0.11b | PyTorch 2.9環境で0.10bを使用すると、コンテキスト並列化に関するバグ（Issue 156012）が再発するリスクが指摘されている[3]。 |
| **2.8** | 0.10b, 0.9b | 0.10b (backported) | 0.9bでビルドした場合、新機能であるSliding Window Attention (SWA) のサポートが失われる[3]。 |
| **2.7** | 0.9b, 0.10b (drop-in) | 0.9b (backported) | 0.10bはAPIの下位互換性を持つため、共有ライブラリのシンボリックリンクによるドロップイン置換が可能だが、統合コードの整合性チェックによりPyTorch 2.7のソースコードからのネイティブビルドには使用できない[3]。 |
| **2.6** | 0.8b | 0.9b (backported) | 0.7bとのAPI互換性は保たれているが、0.8b専用の機能を必要とするため、0.7bでコンパイルすると実行時エラーが発生する[3]。 |
| **2.4 / 2.3** | 0.6b / 0.4b | 0.10b / 0.4b | ROCmフォーク版では旧バージョンのPyTorchに対しても新しいAOTritonの機 |

| | | | |
|---|---|---|---|
| | | | 能がバックポートされている場合がある[3]。 |

この互換性の複雑さは、ROCm環境でAIソフトウェアスタックをソースから構築する際、あるいは依存関係を解決する際の最大の障壁となっている。特定のバージョンのPyTorchを構築するためには、適合するTritonコミットのサブモジュールとAOTritonのバージョンを正確に一致させる必要があり、不適合な組み合わせは不可解なHIPエラーやサイレントクラッシュを引き起こす原因となる[1]。

# 3. FlashAttentionの理論的基盤とAOTritonとの概念的差異

ユーザーからの照会において頻出する混乱の一つが、「AOTriton」と「FlashAttention」の混同である。これらは競合するアルゴリズムではなく、「実装・デリバリー手段」と「数学的アルゴリズム」という直交する概念である。

## 3.1 FlashAttentionのアルゴリズム的本質

FlashAttentionは、Tri Dao氏らによって提唱された、ハードウェアのメモリ階層（IO）を意識したExact Attention（厳密なアテンション）の計算アルゴリズムである[3]。標準的なトランスフォーマーモデルのアテンション機構は、クエリ（Q）とキー（K）の行列積を計算してスコア行列を生成し、ソフトマックス関数を適用した後にバリュー（V）との行列積を計算する（$\text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$）。この過程で、シーケンス長 $N$ に対して $O(N^2)$ のメモリ空間を必要とする巨大な中間テンソルが生成され、これを大容量だが比較的低速なHBM（High Bandwidth Memory / VRAM）に読み書きするオーバーヘッドが推論および学習における最大のボトルネックとなっていた[7]。

FlashAttentionは、この巨大な中間テンソルをHBMに書き出すことなく、GPUダイ上の極めて高速なSRAM（オンチップメモリ）上でタイリング（分割）計算を行い、オンラインでのソフトマックス還元アルゴリズムを適用することで、数学的には標準アテンションと完全に同一の結果を出力しながら、メモリ計算量を $O(N)$ へと劇的に削減する手法である[3]。これにより、8Kやそれ以上の長文脈（Long-Context）LLMの学習および推論が物理的なVRAM容量に収まるようになり、現在ではvLLM、SGLang、llama.cppなど、ほぼすべての主要な推論エンジンで標準的に採用されている[7]。

アルゴリズムは継続的に進化しており、FlashAttention-2では並列化とワークパーティショニングの改善により約2倍の高速化が図られ、FlashAttention-3ではNVIDIAのHopperアーキテクチャ（H100等）特有のTMA（Tensor Memory Accelerator）やWGMMA命令を利用した非同期データ転送と計算のオーバーラップ、さらにはFP8演算がサポートされている[2]。

## 3.2 AOTritonとの関係性とハードウェア・エコシステムの差異

FlashAttentionが「計算アルゴリズムの仕様」であるのに対し、AOTritonはそれをAMDのGPU上で「Triton言語を用いて記述・コンパイルし、提供するためのパッケージングシステム」である[3]。NVIDIA環境とAMD環境では、このアルゴリズムをハードウェアに実装し提供するレイヤー構造が根本的に異なる。

以下の表は、両者のエコシステムにおけるFlashAttentionの実装と提供経路の相違を示している。

| 比較項目 | **NVIDIA (CUDA / Hopper アーキテクチャ等)** | **AMD (ROCm / CDNA・RDNAアーキテクチャ)** |
| --- | --- | --- |
| ネイティブ実装基盤 | CUDAおよびNVIDIAのCUTLASSライブラリを用いて直接記述されており、Day 0からハードウェア固有の最適化（TMA等）が施されている[2]。 | 従来はComposable Kernel (CK) に依存していたが、現在はAOTritonを介したTritonベースの実装へと移行しつつある[1]。 |
| **PyTorch**のフォールバック機構 | SDPA経由でネイティブなflash_attn バックエンドが優先的に自動選択され、サポート外の形状の場合はxFormersやMathバックエンドにフォールバックする[7]。 | check_gpu(stream) 関数が事前コンパイル済みのAOTritonイメージの有無を確認し、存在しない場合はCKまたは標準のMathバックエンドへダウングレードする[4]。 |
| コンパイルと導入の複雑性 | 事前ビルドされたPython Wheel (pip) を通じて極めて容易に導入可能。または実行環境での単純なJITコンパイルが行われる[2]。 | AOTritonによる完全なAOT（事前コンパイル）による静的ライブラリリンクが必要。互換性のマトリクス管理が厳格であり、「依存関係の地獄」を引き起こしやすい[1]。 |

AMD環境において、従来FlashAttentionは「Composable Kernel (CK)」と呼ばれるC++テンプレートライブラリによって実装されていた[2]。しかし、CKはMFMA（Matrix-Fused Multiply-Add）命令に強く依存しており、これがハードウェア側に実装されているCDNAアーキテクチャ（MI200やMI300Xなど）では機能するものの、一部の古いアーキテクチャ（Vega世代のgfx900やgfx906）やコンシューマ向けのRDNAアーキテクチャではサポートが限定的であったり、コンパイル時に除外されたりする問題があった[8]。

AOTritonは、このCKの脆さとハードウェア対応の偏りを克服するための戦略的ソリューションである。複雑なC++テンプレートの代わりに、より抽象度の高いTriton言語でFlashAttentionを記述し、それをAOTritonというパイプラインを通じて各ハードウェア向けのアセンブリ（hsaco）に事前変換して配布することで、AMD製GPUの多様なラインナップ全体に最適化されたAttentionカーネルを安定して供給することが可能となった[1]。

# 4. Ryzen 7 8700G (Radeon 780M / gfx1103) 環境の特性と運用上の課題

本レポートの主題の一つである、AMD Ryzen 7 8700G環境での利用を想定した場合、AIワークロードの実行にはアーキテクチャ特有の根本的な制約とソフトウェア的な障壁が立ちはだかる。

## 4.1 UMA（Unified Memory Architecture）の物理的特性と影響

Ryzen 7 8700Gは、Zen 4アーキテクチャのCPUとRDNA 3アーキテクチャの統合型GPU（Radeon 780M、アーキテクチャコード：gfx1103）を単一のダイに収めたAPU（Accelerated Processing Unit）である[10]。データセンター向けのGPU（HBM搭載）やディスクリートGPU（GDDR6搭載）とは異なり、APUは専用のVRAMを持たず、システムのメインメモリ（DDR5）をCPUとGPUで共有するUMA（Unified Memory Architecture）を採用している[10]。
この構造は、巨大なLLMの重みデータをPCI-Expressバスを介さずにシステムRAMから直接GPUへマッピングできるという利点をもたらす一方で、メモリの絶対的な帯域幅が専用VRAMに比べて著しく狭いという致命的な弱点を持つ。深層学習、特にトランスフォーマーモデルの推論は高度なメモリバウンド（メモリ帯域幅律速）のワークロードであるため、データの読み書き回数の増加はそのまま全体的なパフォーマンスのボトルネックへと直結する。

## 4.2 公式サポートの欠如と環境変数によるスプーフィング

ROCmおよびAOTritonの公式に事前ビルドされたバイナリ（Python Wheelや共有ライブラリ）は、主としてMI300Xなどのデータセンター向け（CDNAアーキテクチャ）や、コンシューマ向けでもハイエンドのRadeon RX 7900 XTX（gfx1100）等を対象としてコンパイルされている[6]。APUに内蔵されるgfx1103は、多くの場合ROCmの公式なサポートマトリクスから外れているか、将来のリリースでの対応予定（PRマージ待ち）にとどまっており、そのままではAOTritonの事前コンパイルバイナリ群を利用することができない[10]。
通常、このような非公式環境でPyTorchやvLLM、ComfyUIといったフレームワークを動作させるためには、環境変数 HSA_OVERRIDE_GFX_VERSION=11.0.0 を設定し、ROCmドライバに対して自らがサポート対象のハイエンドGPUである gfx1100 であると偽装（スプーフィング）させるワークアラウンドが広く用いられている[10]。

## 4.3 未最適化フォールバックによる破滅的影響とシステムクラッシュ

スプーフィングは一時的な解決策としては機能するものの、PyTorchのSDPAが呼び出すAOTritonのディスパッチャ（check_gpu(stream)）は、カーネルイメージ（aotriton.images）の有無を厳密にチェックする。環境変数を偽装しても、実際にロードされたバイナリの中にハードウェアアーキテクチャの完全な合致が見られない場合、ディスパッチャは処理を拒否し、PyTorchはFlashAttentionを使用せずに標準のアテンション（Mathフォールバック）へと静かにダウングレードする[4]。

前述の通り、標準アテンションのメモリ空間計算量は $O(N^2)$ である。シーケンス長 $N$ が長くなった瞬間、あるいは高解像度の画像生成処理を開始した瞬間、巨大な中間テンソルが生成され、これが帯域幅の細いシステムのDDR5メモリに一気にフラッシュされる。結果として、以下のような深刻な現象が引き起こされる。

- メモリスワップと**OOM**（**Out of Memory**）の誘発**:** 限られた共有メモリ空間が即座に枯渇し、システム全体の深刻な遅延や、OSのOOM Killerによるプロセスの強制終了が頻発する[8]。
- 計算速度の著しい低下**:** メモリ律速により、推論速度（Tokens per Second）や画像生成のイテレーション速度が実用不可能なレベルまで落ち込む[2]。
- ドライバレベルの致命的なクラッシュ**:** 報告によれば、Windows 11環境下でComfyUIなどを用いてVRAM（共有メモリ）の限界を超過した場合、グラフィックスドライバが共有システムメモリへのページング処理に失敗し、画面のブラックアウト、マザーボードのVGAフォールトLEDの点灯、そしてハードウェアの強制再起動を伴う破滅的なシステムクラッシュを引き起こす事例が確認されている[20]。これを回避するためには、アプリケーション起動時に

--disable-smart-memory --reserve-vram 8 といった引数を与え、メモリの枯渇を未然に防ぐ防御的な運用を余儀なくされる[20]。

# 5. 8700G環境におけるAOTritonソースビルドの有効性と最適化の深層

上記の制約と脆弱性を踏まえると、Ryzen 7 8700G（gfx1103）向けにAOTritonおよび依存するPyTorchスタックを「ソースコードからビルド（コンパイル）」するアプローチは、極めて高い技術的ハードルを伴うものの、APUの潜在的なAI計算能力を安全かつ最大限に解放するための最も有効な手段となる。

## 5.1 ソースビルドによる有効性（メリット）

ソースコードからビルドすることによって得られる主なメリットは以下の三点に集約される。

**(1) アーキテクチャのネイティブ対応とMathフォールバックの回避**

ビルド時のCMake構成パラメータにおいて、ターゲットアーキテクチャとしてCMAKE_HIP_ARCHITECTURES="gfx1103" または --target_gpus gfx1103 を明示的に指定することで、環境変数による不安定なスプーフィングを排し、Radeon 780Mのハードウェアプロファイルに完全に合致するAOTritonカーネルアセンブリ（hsaco）を生成することが可能になる[10]。これにより、PyTorch実行時のランタイムディスパッチャ（check_gpu）がカーネルイメージを正しく検知し、標準のMathバックエンドへのフォールバックを回避して、ネイティブにFlashAttentionのC++パスを確実に選択・実行するようになる[4]。

**(2) 劇的なパフォーマンス向上とメモリフットプリントの削減**

ネイティブなFlashAttentionが8700G上で有効化された場合の効果は絶大である。未最適化状態（FAなしのMathフォールバック状態）からFlashAttentionを有効化した場合、推論フレームワーク（例：llama.cpp等の派生テスト）におけるトークンスループットは平均して約39%の向上が観測されている（574 tok/s から 799 tok/s への改善など）[10]。さらに、特定のアテンション負荷テスト（XLarge: Batch=16, Heads=16, SeqLen=4096）においては、最適化前と比較してForwardパスの処理速度が最大で20倍から30倍、Backwardパスでも8倍から20倍に達する飛躍的な高速化が報告されている[23]。これに加えて、$O(N^2)$ から $O(N)$ への計算量削減によりメモリ消費量が劇的に抑制されるため、DDR5ベースの共有RAMの消費が最小限に抑えられ、前述したWindows環境での破滅的なOOMクラッシュのリスクが根本的に解消される[7]。

**(3) UMA環境（DDR5）に最適化されたチューニングデータベースの生成**

前段で述べた通り、AOTritonはビルドの過程で tune_flash.py を実行し、最適なハイパーパラメータを探索してチューニングデータベースを構築する[5]。公式から事前提供されているバイナリのルックアップテーブルは、極めて広帯域なHBMを搭載したMI300Xや、広帯域GDDR6を搭載したRX 7900向けにチューニングされたスレッドブロックサイズ（大きなSRAMタイル）が設定されている。これを帯域幅の狭いUMA環境でそのまま実行した場合、キャッシュミスや非効率なデータフェッチが発生する懸念がある。ソースコードから8700Gの実機環境でコンパイルとチューニングパスを実行することにより、DDR5ベースのUMAシステムに最適化されたタイルサイズや並列度が導き出され、ルックアップテーブルに組み込まれるため、推論速度と安定性がさらに改善する効果が期待できる[3]。

### 5.2 ソースビルドに伴う運用上の影響とトレードオフ

ソースビルドは多大なパフォーマンスメリットと安定性を提供する一方で、ビルドプロセス自体が運用上の最大の障壁となる「過酷なシステムリソースの要求」をもたらす。

**(1)** ビルドシステムへの致命的なリソース負荷

AOTritonのビルドでは、単一のTritonカーネルからハードウェア専用のC++シムコードが約20,000ファイル以上も自動生成される[14]。これらのファイルをコンパイル（nvcc または hipcc に相当するバックエンドを使用）する際、コンパイラは極めて膨大なシステムメモリとCPUスレッドを消費する。開発者の報告によれば、128スレッドや256スレッドのハイエンドEPYCプロセッサに512GBのRAMを搭載した環境であっても、完全な並列ビルドを行うにはメモリが不足し、ロードアベレージが異常な数値に跳ね上がり、OSのOOM Killerによってビルドプロセスが強制終了される事態が頻発している[24]。また、メモリが32GB程度の環境ではコンパイル中に高確率でクラッシュが発生する[25]。したがって、8700G単体（通常、搭載メモリは32GB〜64GB程度）でセルフホストビルドを行う場合、CMake/Ninjaの並列ジョブ数（-j オプション）を意図的に極端な少数に制限する必要があり、結果としてビルドの完了までに数時間から数十時間という膨大な時間を要することになる[3]。

**(2) Windows**環境における制約とクロスコンパイルの必要性

8700G搭載のシステムでWindows 11を利用している場合、AOTritonはWindows上でネイティブにカーネルイメージをビルドできないという致命的な仕様上の制約がある。これは、基盤となるTritonコンパイラ自体がWindowsを完全には公式サポートしていないことに起因する[3]。Windows環境向けにビルドを完遂するには、別途Linux環境でビルドされた aotriton.images フォルダを手動で抽出し、Windows側のビルドプロセスで AOTRITON_NOIMAGE_MODE フラグを設定してイメージのビルドをスキップさせつつ、手動でリンクさせるという極めて高度で煩雑なワークアラウンドが必要となる[3]。したがって、8700Gの性能を最大限に引き出すためのソースビルドを現実的な工数で行う場合、ネイティブのUbuntu環境、あるいは最低限WSL2（Windows Subsystem for Linux 2）環境上でROCmスタックを構築することが事実上の必須条件となる[12]。

## 6. 結論

本レポートの解析を通じ、高度なAI計算基盤におけるAOTritonの処理原理、FlashAttentionとの相関性、およびAMD Ryzen 7 8700G環境における最適化メカニズムについて以下の結論が導き出される。

AOTritonは、FlashAttentionという革新的なアルゴリズムそのものではなく、複雑な依存関係やJITコンパイルの脆弱性を抱えるROCmエコシステムにおいて、そのアルゴリズムを確実かつ高速にAMDハードウェアへと届けるための「高度な事前コンパイル型のデリバリー・パイプライン」である。膨大なC++シムコードの自動生成とSQLiteベースのチューニングデータベースを組み合わせることで、実行時のオーバーヘッドを極限まで削ぎ落とす設計思想が貫かれている。

Ryzen 7 8700G（gfx1103）環境において、公式のバイナリパッケージがデータセンターやハイエンドGPUに偏重している現状では、スプーフィングを用いた場当たり的な運用はMathフォールバックを引き起こし、UMAアーキテクチャの狭い帯域幅と相まって深刻な性能低下や破滅的なシステムクラッシュ（OOM）を招く。したがって、対象アーキテクチャを明示してAOTritonをソースコードからビルドし、APU固有のDDR5メモリ特性に最適化されたネイティブなFlashAttentionを有効化することは、単なる「微調整」の範疇を超え、同プロセッサをローカルでのLLM推論や画像生成AIの実行に耐えうる

実用的なエッジデバイスへと変貌させるための「不可欠な最適化パス」であると結論付けられる。ただし、その運用には数万のファイル生成と極端なメモリ消費を伴うビルドプロセスを乗り越える必要があり、適切なOS選定（Linux推奨）とジョブ数制限による緻密なリソース管理が求められる。コンパイル時間および環境構築の初期コストは極めて高いものの、AI推論ワークロードにおける劇的なスループット向上（数十倍のForwardパス高速化など）とメモリ節約によるシステム安定性の確保という形で、その投資は遥かに上回る技術的有効性をもたらすものである。

## 引用文献


1. pip install vllm: The iceberg under a single command | Red Hat Developer, https://developers.redhat.com/articles/2026/04/16/pip-install-vllm-single-command
2. State of PyTorch Hardware Acceleration 2025, https://tunguz.github.io/PyTorch_Hardware_2025/
3. GitHub - ROCm/aotriton: Ahead of Time (AOT) Triton Math Library, https://github.com/ROCm/aotriton
4. target_gpus: expected at least one argument` if only unsupported GPUs are selected · Issue #169 · ROCm/aotriton - GitHub, https://github.com/ROCm/aotriton/issues/169
5. [Documentation]: The overall tuning idea of aotriton · Issue #38 - GitHub, https://github.com/ROCm/aotriton/issues/38
6. [Feature]: Memory Efficient Flash Attention for gfx1100 (7900xtx) · Issue #16 · ROCm/aotriton, https://github.com/ROCm/aotriton/issues/16
7. FlashAttention Guide 2026: FA-2, FA-3, Hopper Optimizations | Local AI Master, https://localaimaster.com/blog/flash-attention-guide
8. Built a simple PyTorch flash-attention alternative for AMD GPUs that don't have it : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1s614i8/built_a_simple_pytorch_flashattention_alternative/
9. [Multi-arch] PyTorch packages build without flash attention when targets not supported by aotriton are included · Issue #4969 · ROCm/TheRock - GitHub, https://github.com/ROCm/TheRock/issues/4969
10. AMD GPUs - llm-tracker, https://llm-tracker.info/howto/AMD-GPUs
11. Windows用のAotriton - TheRock - rocm7rc : r/ROCm - Reddit, https://www.reddit.com/r/ROCm/comments/1nefttc/aotriton_for_windows_on_therock_rocm7rc/?tl=ja
12. September 2024 Update: AMD GPU (mostly RDNA3) AI/LLM Notes : r/LocalLLaMA - Reddit, https://www.reddit.com/r/LocalLLaMA/comments/1fssvbm/september_2024_update_amd_gpu_mostly_rdna3_aillm/
13. [aotriton] Enabling `gfx101X` and `gfx103X` `gfx1103` `gfx908` `gfx906` for PyTorch linux and windows · Issue #1925 · ROCm/TheRock - GitHub, https://github.com/ROCm/TheRock/issues/1925
14. sci-libs/aotriton-bin Package Details - Gentoo Browse, https://gentoobrowse.randomdan.homeip.net/packages/sci-libs/aotriton-bin

15. Why I can't use pytorch on Windows with AMD GPU? - Reddit, https://www.reddit.com/r/pytorch/comments/1jai2fq/why_i_cant_use_pytorch_on_windows_with_amd_gpu/
16. AMD ROCm Ai Applications on RDNA3 - 8700G & 7800 XT - Linux and Win11 - Reddit, https://www.reddit.com/r/ROCm/comments/1dfs0zn/amd_rocm_ai_applications_on_rdna3_8700g_7800_xt/
17. [Feature]: ROCm Support for AMD Ryzen 9 7940HS with Radeon 780M Graphics #3398, https://github.com/ROCm/ROCm/issues/3398
18. AMD Ryzen 7 8700G APUで動くPytorch/ROCmの環境を作る(未完) - errno_mmd's blog, https://errno-mmd.hatenablog.com/entry/2024/12/22/201455
19. AMD ROCm Ai Applications on RDNA3 - 8700G & 7800 XT - Linux and Win11 - Reddit, https://www.reddit.com/r/Amd/comments/1dfuehs/amd_rocm_ai_applications_on_rdna3_8700g_7800_xt/
20. Severe System Crash on VRAM Overflow with ComfyUI on Windows 11 · Issue #1320 · ROCm/TheRock - GitHub, https://github.com/ROCm/TheRock/issues/1320?timeline_page=1
21. RAM memory growth of triton server, until killed by OS · Issue #7035 - GitHub, https://github.com/triton-inference-server/server/issues/7035
22. Aotriton for Windows on TheRock - rocm7rc : r/ROCm - Reddit, https://www.reddit.com/r/ROCm/comments/1nefttc/aotriton_for_windows_on_therock_rocm7rc/
23. PyTorch w/ Flash Attention + vLLM for Strix Halo - Framework Community, https://community.frame.work/t/pytorch-w-flash-attention-vllm-for-strix-halo/74736
24. ROCm 7.1.1: you can (not) build | lunnova.dev, https://lunnova.dev/articles/rocm-711-you-can-not-build/
25. Build consumes 32GB RAM triggers OOM crash when compiling CUDA object · Issue #111526 - GitHub, https://github.com/pytorch/pytorch/issues/111526