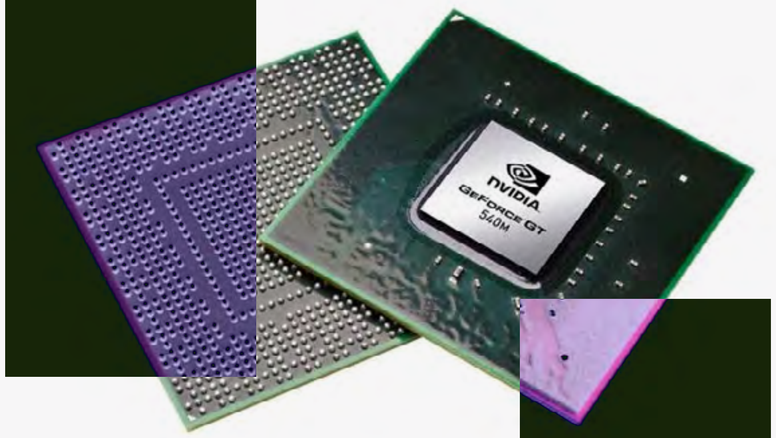
Qt Quality Assurance

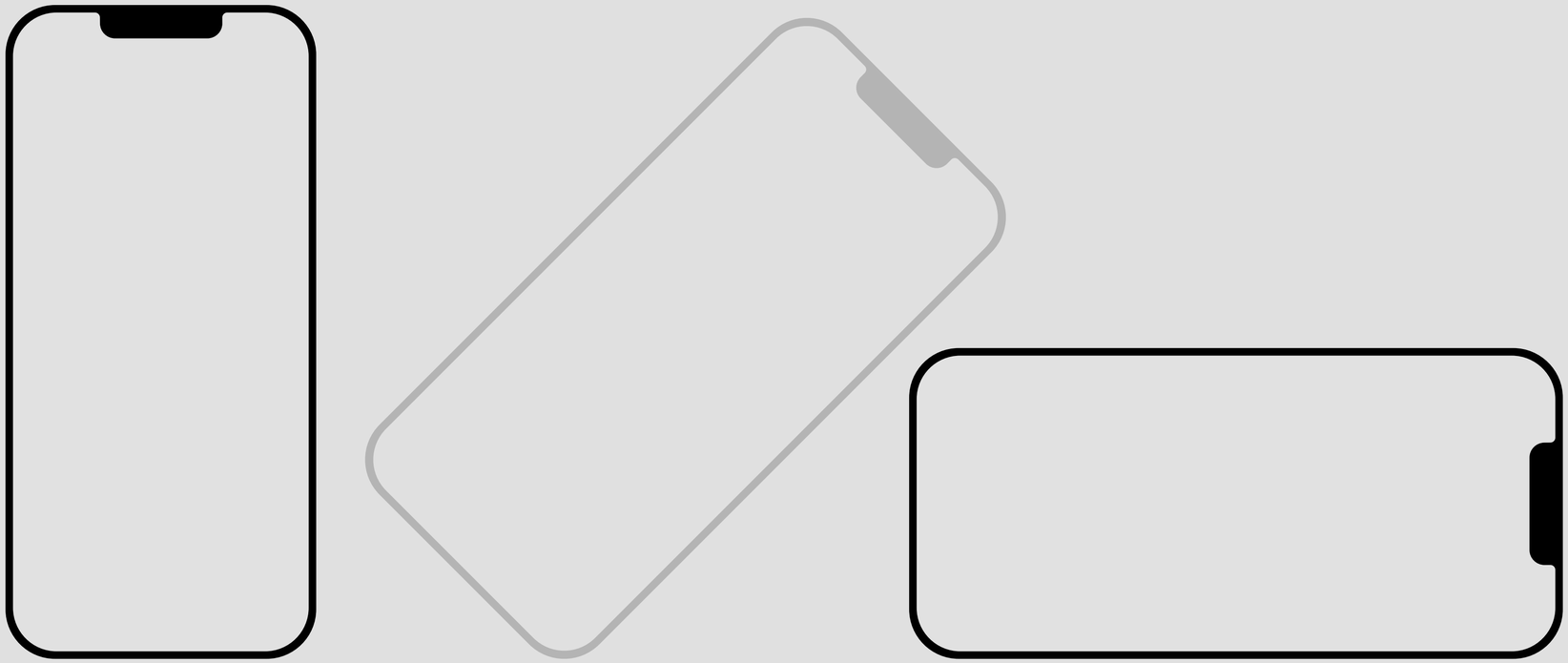
CUDAコード品質を極める

ゲームを変えるCUDAアプリケーションを
開発するためのプレイブック



NVIDIA
CUDA

モバイル端末で読んでいますか？
横向きにすると読みやすくなります。



はじめに

このガイドは長いです — 長すぎると思う人もいるかもしれません。

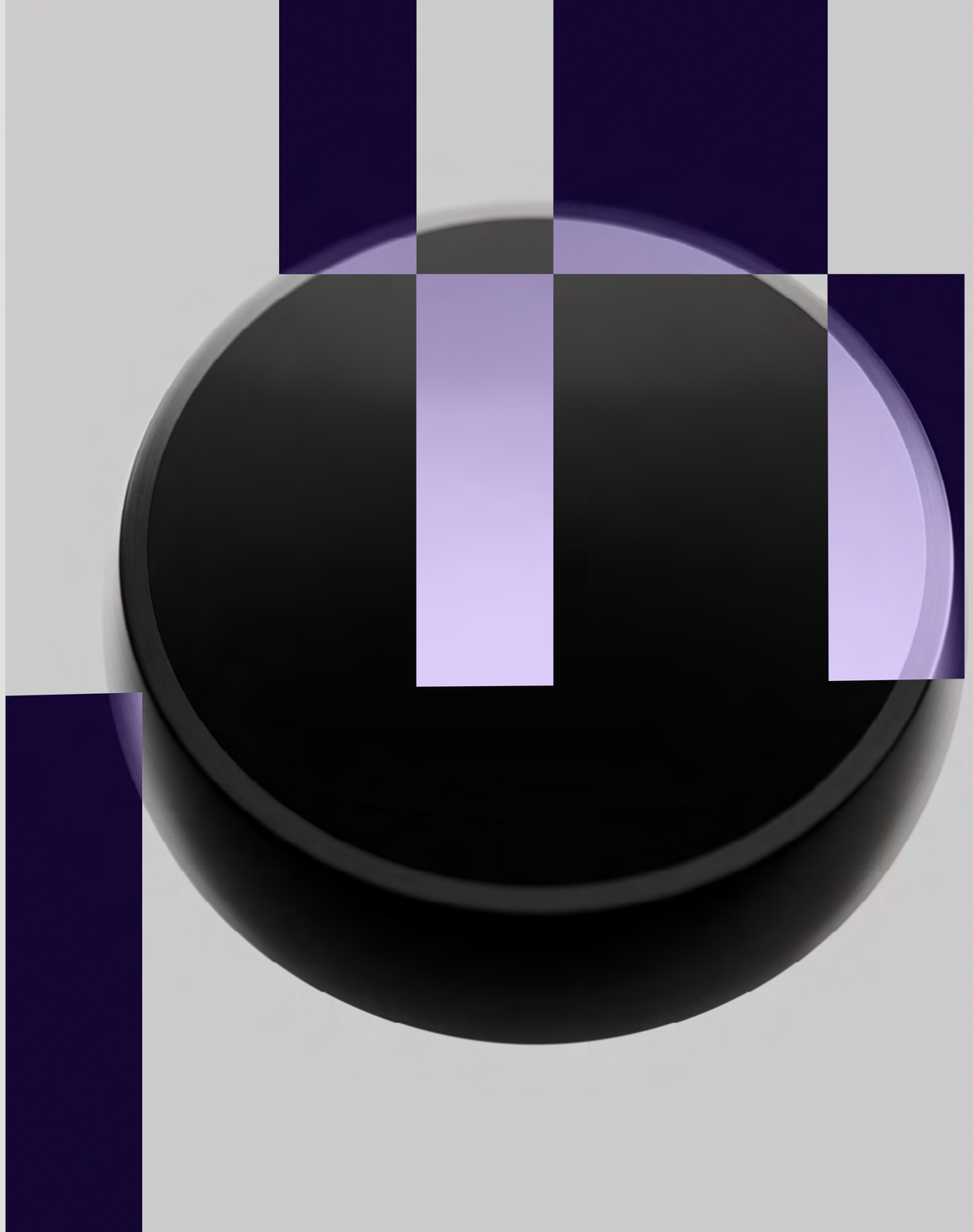
しかし、ソフトウェア品質は取るに足らない話題ではありません。本質的に、製品の命運を左右し、安全性が要求される環境では生死に関わります。だからこそ、これについて話す（あなたの場合は読む）時間には価値があります。

本ガイドは、ジュニアからシニアまでの開発者を対象に、CUDA C++プロジェクトにおけるソフトウェア品質を扱います。ソフトウェア品質の重要性は使用する言語に関係なく同じですが、CUDA開発者が望む品質レベルを実現するためのツールは多くありません。

Axivion for CUDAは、CUDA C++アプリケーションの開発を支援するために特別に設計されました。本ガイドではその内容についてご紹介します。安全性が要求される環境で使われるソフトウェアを中心にしていますが、ここで説明する手法はあらゆる用途に適用できます。

ガイドでは、高品質なコードがもたらす経済的メリットにも触れます。これらが開発者にとって主な動機ではないとしても、非技術系スタッフに対して、ソフトウェア品質への投資がリソースの無駄遣いとは正反対である理由を説明する助けになります。

このガイドを最初から最後まで読むにせよ、興味のある章だけを選ぶにせよ、あなたのコードが価値を生むための助けになれば幸いです。



目次

Qt Quality Assurance

序章

今日の高性能コンピューティング (HPC: High-Performance Computing) の世界では、CUDAはさまざまな業界でアプリケーションを加速する基盤技術となっています。自動運転やロボティクスから科学シミュレーション、AIまで幅広く使われています。しかし、CUDAアプリケーションが複雑化・大規模化するにつれ、性能だけでは不十分になりました。安全性、セキュリティ、信頼性、保守性が成功するソフトウェア開発の柱となっています。イノベーションと規制のバランスを見つけることで、最先端のCUDAアプリケーションを高速かつスケーラブルにするだけでなく、安全でセキュア、そして監査に耐えられるものにできます。

多くの静的解析ソリューションは、CUDA特有の構文を無視するか、汎用的なC/C++拡張として扱い、GPUプログラミングのニュアンスを見落とします。一方、Axivion for CUDAはCUDAアプリケーションのために設計されており、CUDA構文の包括的サポート、アーキテクチャ検証、業界標準やコーディングガイドラインへの準拠（NVIDIAのCUDA C++ガイドラインを含む）を提供します。

AxivionはCI/DevOpsワークフローにシームレスに統合され、クリーンでコンプライアンスに沿った将来性のあるコードを維持できます。さらに重要なのは、安全性が要求される環境で求められる厳格な規則や規制を順守する助けとなることです。

このガイドでは、高品質なCUDAアプリケーションを開発するために何が必要か、そしてAxivion for CUDAがソフトウェアエロージョン（ソフトウェアの劣化）にどう対処し、技術的負債を防ぎ、イノベーションの力を失うことなくアーキテクチャの整合性を保つ手助けをするかを説明します。

まず、ソフトウェア品質をどう維持するか、その経済的メリットに焦点を当てます。そのうえで、安全性が要求される環境でCUDAアプリケーションを開発する実践的側面を見ていきます。

1

ソフトウェアエロージョン（劣化）と技術的負債を理解する

ソフトウェア品質—維持と保護の方法

GPUアクセラレーテッドコンピューティングの急速に発展する世界では、高性能が究極の目標ですが、それは方程式の半分に過ぎません。信頼性、保守性、スケーラビリティも同じく重要です。CUDAアプリケーションが複雑化・大規模化するにつれ、ソフトウェア品質はイノベーションを支える基盤となります。

NVIDIA CUDA C++ Guidelines for Robust and Safety-Critical Programmingなど、内部品質と外部品質を保証するためのコーディングガイドラインや業界標準が存在します。これらを守ることは選択ではなく必須です。規則や規制が変わるたびに、品質を損なわずに素早く採用できるか判断する必要があり、時間的制約のもとでは難しくなります。

では、期限がある中でどのようにソフトウェア品質を確保すればよいのでしょうか？

答えは、**ソフトウェアエロージョンを止め、技術的負債の増加を防ぐことです。**

ソフトウェア品質は時間がかかり、とても無理だと考える企業は多いものです。しかし実際には、高い品質基準を保つために必要な多くのステップは、Axivion for CUDAのような高度なコード解析ツールを導入することで容易に自動化できます。

他のソフトウェア品質ツールと異なり、Axivion for CUDAはC/C++にキーワードやAPIを追加して、CPUコードと並行してNVIDIA GPU上で直接実行するコードを書けるようにするアプリケーション向けに特別に開発されています。一度セットアップすれば、Axivion for CUDA は CI/DevOps 環境にシームレスに統合され、コードをクリーンな状態に保ち続けます。

さらにAxivionは、CUDA上でのMISRAなど、コーディングガイドラインや各種業界標準への準拠もチェックできます。これはソフトウェアに品質のレイヤーを追加するだけでなく、評価や監査の準備・実施にも役立ちます。

Axivion for CUDA ↗

ソフトウェア品質は | 次の2つに分けられます：

内部品質

保守性、テスト容易性、進化容易性など、内部品質の側面は開発者にしか見えません。

外部品質

ユーザビリティ、正しい機能、信頼性、安全性・セキュリティなど。外部品質はエンドユーザーから見える部分です。

ソフトウェアエロージョンの定義と原因

ソフトウェアは無生物ですが、生物のように時間とともに成長し進化します。コードへの変更は痕跡を残します。

ソフトウェアエロージョン、ソフトウェアの腐敗、ソフトウェアの腐朽、技術的負債――これらはすべて、ソースコードの構造が徐々に劣化することを指す同義の言葉です。劣化は、性能低下、新しい要件に合わせた変更の難易度増加、プログラムエラーの増加といった形で現れます。

ソフトウェアエロージョンの厄介さは、それを引き起こす個々の欠陥が必ずしも間違いではなく、プログラムの動作を妨げない場合が多い点にあります。しかし、小さな、そして文書化されない省略や追加が積み重なると、ソフトウェアの理解のしやすさに影響します。元のアーキテクチャからのさまざまな逸脱によってコードは不可解になり、テストが難しくなり、現場での不具合も増えます。

コードを使い物にならなくさせないためには修復、つまりリファクタリングが必要です。リファクタリングで一時的にソフトウェアエロージョンを止められますが、それは症状への対処にすぎません。原因を治すには、技術的負債の発生を防ぐ必要があります。

ソフトウェアエロージョンの典型的な原因

アーキテクチャドリフト／アーキテクチャ負債	実装が意図したアーキテクチャから逸脱したときに発生します。短期的な便宜で積み重ねた結果、将来の変更が困難になります。
デッドコード	実行時に到達しないコード。リソース（メモリ、プロセッサ時間、性能）を浪費します。
クローンや重複コード	複数箇所に存在する同一コード。更新時に他の箇所のクローンを更新し忘れるリスクがあります。
メトリクス違反	行数やトークン数、McCabe複雑度、NPath数などのメトリクスはコードをクリーンに保つために使われます。これらのメトリクスが満たされないと、ソフトウェアの劣化が加速します。
循環依存	関数が互いに呼び合うことで無限再帰の危険を伴います。直接循環依存があれば、システムがいつでもクラッシュするリスクがあります。

ソフトウェアエロージョン vs. 技術的負債

「ソフトウェアエロージョン」と「技術的負債」という言葉は同義で使われることが多いですが、小さいながらも重要な違いがあります。

技術的負債は金融負債に例えられ、意識的な決断と、誰かの過失または責任というニュアンスを含むことがあります。利点と欠点を秤にかけ、ひとまず短期の修正で済ませ、後から必要な是正を行うと決めたときに技術的負債が発生します。こうしたやり方には正当な理由があることも多く、意図的な決断はいつかこの負債を返済するという前提を伴います

しかし多くの場合、その「いつか」は訪れません。ソフトウェア品質は会社にとって主眼ではなく、必要な注意やリソースを割いてもらえないからです。ある時点で負債が膨れ上がり破綻します。コードはリファクタリング不能になり、手の施しようがなく、新しく作り直すしかなくなります。

一方のソフトウェアエロージョンは技術的負債と同義に使われながらも、時間の経過とともに進行し技術的負債に至るプロセスそのものです。自然浸食に例えるとわかりやすいでしょう。望むと望まざるとにかかわらず起こります。誰かが故意にソフトウェアを壊そうとしたと決めつけるべきではなく、コード中の問題は成長痛のようなものだと考えるべきだと私たちは考えています。ソフトウェア開発ではそれが自然な一部なのです。

自然界で侵食をもたらす要因は、熱、寒さ、風、水、氷などで、最初は目に見えない損傷でも断崖や岩、海岸線を絶えず蝕みます。ソフトウェア開発における侵食要因は時間的制約とリソース不足です。コードクローン、コーディングガイドライン違反、メトリクス違反、アーキテクチャ違反はソースコードという岩にできたひび割れで、最終的にそれを崩壊させます。

しかし「起こるものだから」といって、無防備でいてよいわけではありません。ソフトウェアエロージョンを止め、技術的負債の必要性を防ぐことで、あなたのソフトウェアは末永く健全に機能します。

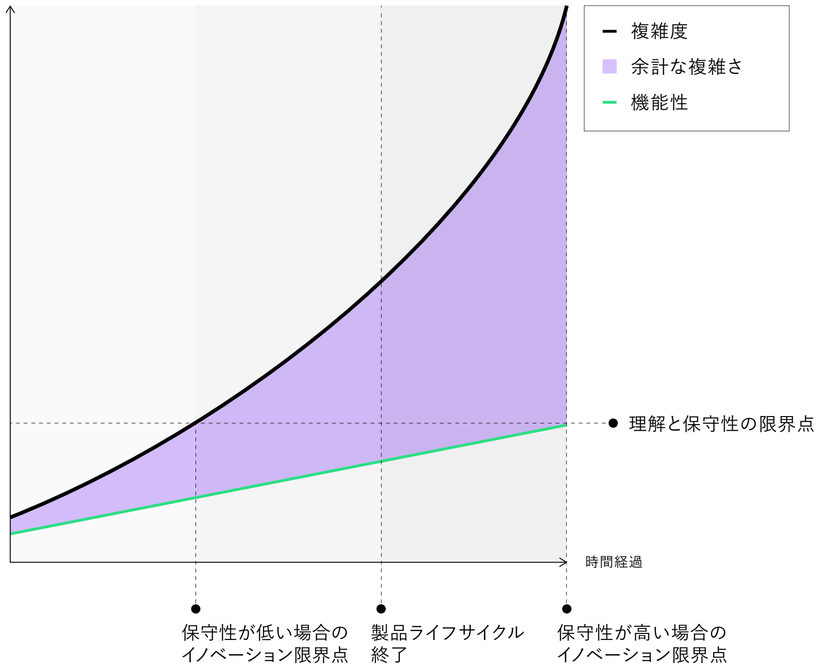
ソフトウェアエロージョンは品質を蝕む

既に述べたように、ソフトウェアエロージョンは性能低下やコード変更の難易度増大を招きます。つまり何を意味するのでしょうか。端的に言えば、イノベーションの可能性が制限されるということです。

初期段階では、機能と複雑性は並行して成長し、ソフトウェアエロージョンは保守性や機能追加能力にほとんど影響しません。この能力は、ソフトウェアを理解し、変更がどのような影響を及ぼすかを知っていることに基づきます。

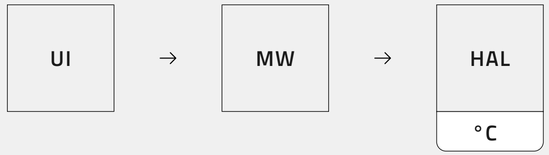
しかし開発が進むにつれ状況は急速に変化し、複雑性は指数関数的に増加します。理解の限界を超えると、ソフトウェアは保守がより困難になり、コード行あたりのエラー数が劇的に増えます。ソフトウェアエロージョンを無視すると、機能追加の代わりに何を直すべきか探る時間ばかりが増え、イノベーションが止まります。リファクタリングはほぼ不可能になり、開発速度は信頼性が損なわれるほどまで低下します。

一方、ソフトウェアエロージョンを止め、コードをクリーンに保てば、開発を続けられます。複雑性が下がることで、バグ修正ではなくイノベーションに時間を割けます。

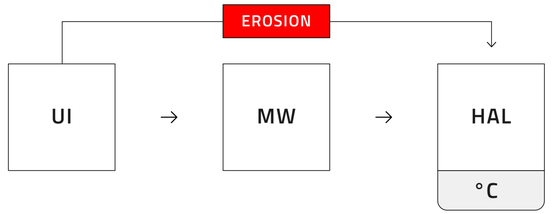


ソフトウェアエロージョンは想像以上に早く進み、初期には問題に見えません。

簡単な例として、°Cを使う車載の温度センサーを考え、次の構造を仮定します。ユーザーインターフェース（UI）がミドルウェア（MW）にアクセスし、MWがハードウェア抽象化層（HAL）の温度センサーにアクセスします。ユーザーインターフェースは本来HALへ直接アクセスしてはいけません。

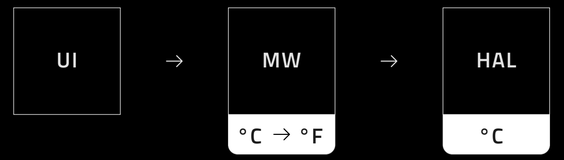


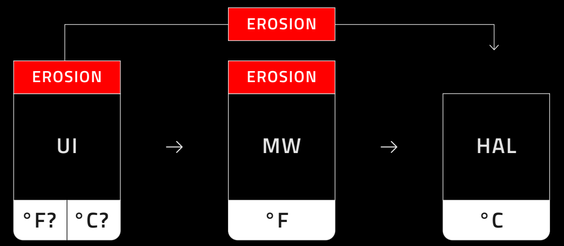
ところが開発者がこのルールを無視してUIがHALから直接情報を取得したらどうなるでしょうか。最初は何も起きないように見えます。問題がなければ誰も違いに気づきません。しかし実際には、意図したアーキテクチャと実装されたコードが一致していないため、ソフトウェアエロージョンが発生しています。見えない不要な複雑さの層を追加したことになります。



次に、温度を摂氏ではなく華氏で表示したいとします。そのためにソフトウェアアーキテクトは単純にミドルウェアに変換処理を追加します。MWの変換はテストされ、意図したとおり動作します。

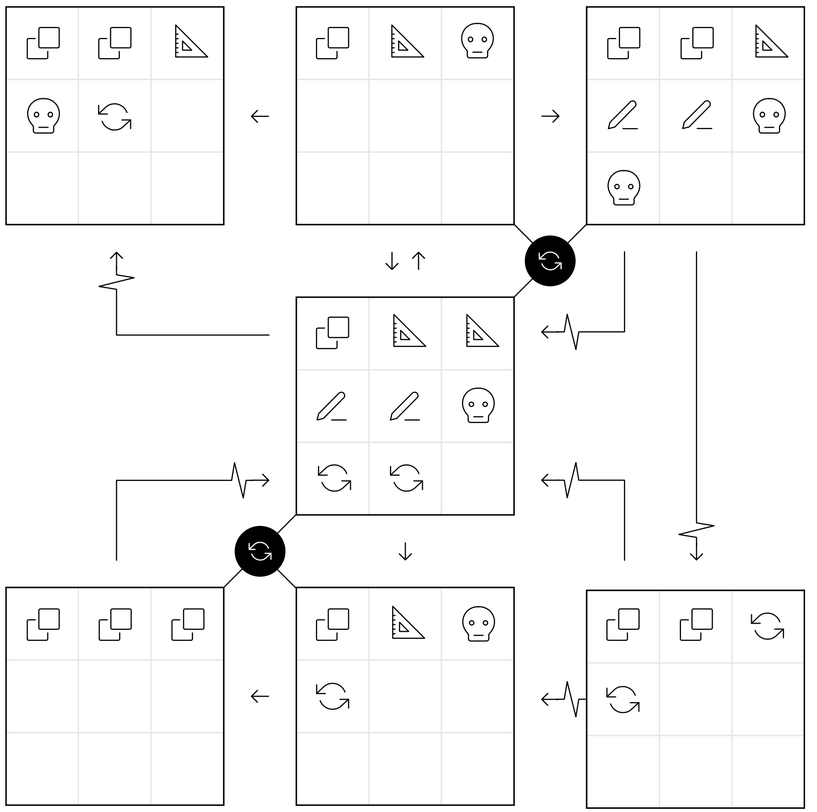
ところが最終のシステム統合テストで、UIが32° Fではなく32° Cで凍結警告を表示することに気づきました。リリースまで数日しかなく、すぐ実装できる解決策が必要です。ミドルウェアでの変換が正しく動いていることは分かっているので、それをUIにコピーしてしまえばよいでしょう。うまく動作し、問題は解決します。しかしここでクローンを追加し、さらにソフトウェアエロージョンを増やしてしまいました。





こうした小さな変更が積み重なり、かつてはクリーンで分かりやすかったソフトウェアが、やがて理解不能な混沌へと変わります。

これは悪循環です。時間の不足から開発者は内部品質をおろそかにし、その結果としてソフトウェアが侵食され、理解が難しくなります。手間のかかる回避策が必要になり、コードはさらに劣化します。開発が遅れ、時間的プレッシャーはいっそう高まります。



アーキテクチャ違反／隠れた依存関係
クローンや重複コード
メトリクス違反
スタイル違反
デッドコード
循環依存

ソフトウェアエロージョンの検出

悪循環を断ち、理解や保守が困難になる前にコードの劣化を防ぐには、できるだけ早期に問題を修正、できれば予防する必要があります。そうすれば問題が引き起こすダメージを減らし、必要なリソースも少なくて済みます。

しかし、初期には目立たないソフトウェアエロージョンをどうやって検出すればよいのでしょうか。たとえ定期的にコードを確認したくても、人間には事実上検出不可能な問題もあります。だからこそ、最初から自動コード解析を導入することが重要です。

では、既にソフトウェアエロージョンや技術的負債に悩まされているコードはどうでしょうか。

レガシーコードやサードパーティコードを扱う場合がそうかもしれません。このコードを、これまでコントロールできなかったとしても、ソフトウェアの残りの部分と同じ品質レベルにする必要があります。ここでAxivionが役立ちます。

ソフトウェア開発をコントロールしましょう。
コードの出どころにかかわらず、主導権を握りましょう。

そのために、ソフトウェアアーキテクチャを検証し、開発中に静的コード解析を使い、コーディングガイドラインや標準への適合を確認してください。

2

CUDA向け Axivion静的コード解析

Axivion for CUDAの始め方

静的コード解析とは？

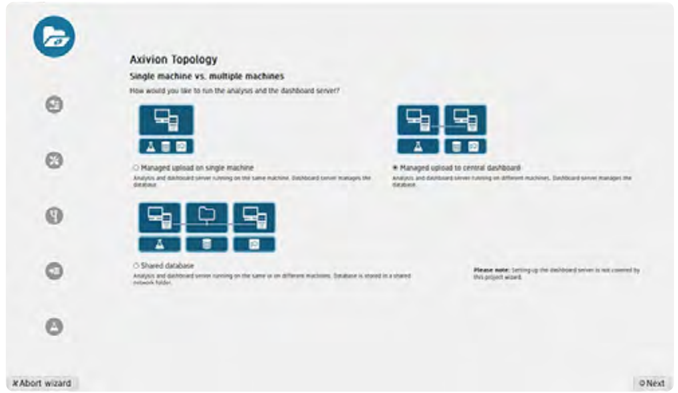
静的コード解析は、バグ、セキュリティ脆弱性、コーディングガイドラインや業界標準からの逸脱を検出する方法です。ソフトウェアテストと異なり、コードを実行する必要がありません。これにより、コミット前にコードの一部だけをチェックでき、ソフトウェアが完成している必要もありません。

Axivion for CUDAの静的コード解析で検出できるもの

- クローンや重複コード
- 循環依存
- デッドコード
- メトリクス違反
- コーディングガイドラインおよび安全・セキュリティ規則の問題

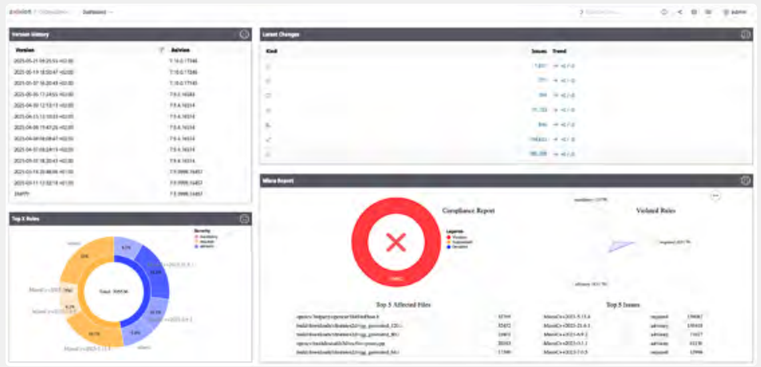
これらは第1章で述べた、ソフトウェアエロージョンを引き起こしコード品質を低下させる要因です。

Axivion for CUDAをインストールしライセンスを有効化すると、セットアップウィザードがプロジェクト設定の必要ステップを案内します。

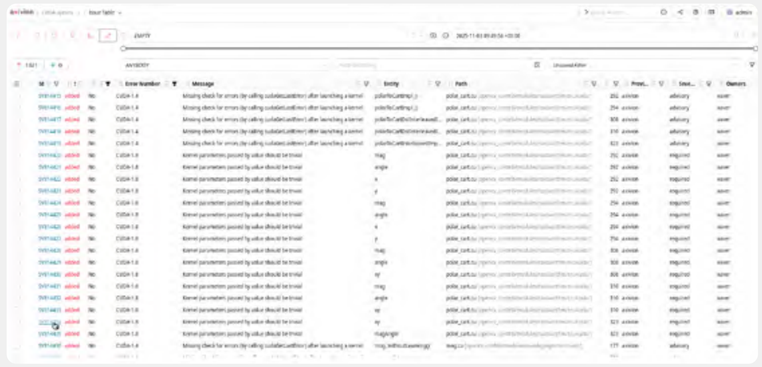


初期設定が終わると、ベースラインを確立するためにコードの全解析が実行されます。

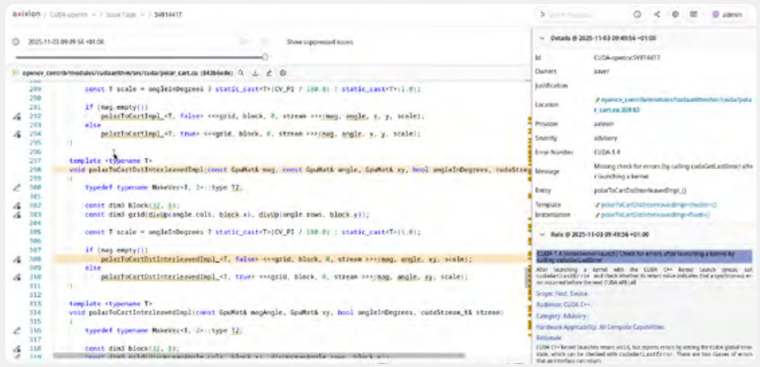
1. プロジェクトのルートディレクトリとソースコードパスを定義し、ビルド環境を指定します。
2. プロジェクトに合ったコーディングガイドラインとメトリクスを選択します。
3. ルールセットとしきい値を設定します。このステップでは、会社固有の要件に合わせたカスタムルールを追加できます。
4. CI/DevOps環境との統合を設定します。
5. 各ビルド後に生成するレポートを設定します。



開発者はダッシュボードから詳細付きのイシューテーブルにアクセスでき、そこからローカルIDEのコードへ直接移動して修正できます。なぜ問題なのかを理解するためのルール詳細も確認できます。



解析結果はカスタマイズ可能なダッシュボードに表示され、2回目以降のコードチェック後は差分解析を含みます。差分解析は時間経過による課題数を追跡し、前回チェックからの変化を示します。



Axivion for CUDA プロダクトツアー ↗

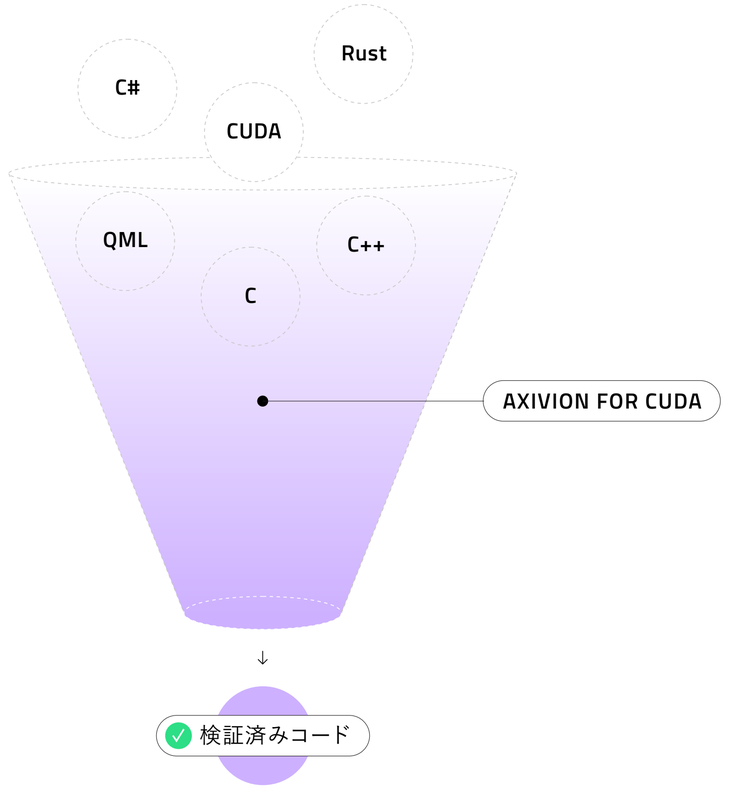
静的コード解析を超えて

Axivion for CUDA は、バグ検出だけではありません。 主要な標準・ガイドライン（MISRA、 CUDA C++ Guidelines for Robust and Safety-Critical Programming、ISO 26262 など）で定められた要件を、コード全体が満たしていることを確実にします（→ 第5章）。

包括的なレポートはソフトウェアの状態を明確に示し、開発プロセスの証跡を提供できます。レポートは自動生成・送付され、必要に応じてカスタマイズ可能です。

Tool Qualification Kitはツールチェーンの分類と資格付けを支援し、CIプロセスに統合できます。検証テストを技術的に自動化することで、アップデートやアップグレードといった変更後も効率的に再実行でき、改善やイノベーションの恩恵を受けながら適合性チェックを繰り返すことができます。提供されるテストドライバーが、関連するルールセットに対して1つ以上のテストを実行し、その結果を報告します。

Axivionの**多言語サポート**により、複数の解析を実行させたり手動レビューしたりせずにコード全体をチェックできます。MISRAルールのチェックを例にとると、多くのツールはCUDAコードを無視するか「誤検知」として扱います。その部分を別途レビューしなければなりません。Axivionは違います。MISRAチェッカーがコード全体を解析し、準拠を確認します。さらにAxivionは複数言語を同時に解析できるため、異なる言語間での呼び出しが正しく意図どおりかなど、多言語コードで起こりうる他の問題も検証できます。



3

CUDAプロジェクト向け Axivionアーキテクチャ検証

アーキテクチャ検証：高性能コンピューティングの秘訣

なぜアーキテクチャを検証するのか？

ソフトウェアアーキテクチャと設計をコードと一致させておけば、新機能の影響を議論するときの指針・基盤としてアーキテクチャを使えます。

そのとき初めて、長期的に的を絞った計画的な製品開発が可能になります。Axivion for CUDAは、コードがアーキテクチャに準拠していることを保証します。機能アーキテクチャに加え、Freedom From Interference（干渉の排除 - ソフトウェア分離とも呼ばれる）など安全・セキュリティアーキテクチャ仕様への準拠もレビュー・チェックします。

CUDAプロジェクトにおけるアーキテクチャ検証の重要性 ↗

ソフトウェアアーキテクチャとは、周囲の環境におけるソフトウェアシステムの中核概念と特性を、構成要素、接続、作成と進化を導く原則を通じて表現したものです。

「アーキテクチャ」という用語は、建設業界との類似からソフトウェア開発に借用されました。かつてはウォーターフォール開発が主流で、コーディング前に詳細な計画が求められました。これは、建築で着工前に設計を固めるのと似ています。現在では状況が変わり、柔軟で変更しやすいソフトウェア開発手法が主流になり、初期段階で厳密な計画を立てる必要性は低くなりました。それでもアーキテクチャ定義は不可欠です。ASPICEなどの標準はV字モデルを参照しており、詳細な計画を期待しています。

ソフトウェアアーキテクチャは、ソフトウェアがどのように動き、周囲とどう適合するかを示す地図のようなものです。ソフトウェアが使用される環境で適切に動作することを確認することは、潜在的な混乱を最小化し、ソフトウェアの長寿命と目的達成を最大化します。

ソフトウェアアーキテクチャを全員が理解できるように、問題をどう解決するかを説明する記述を作成します。この記述はガイドブックのように、ソフトウェアがニーズを満たす姿を示します。

ソフトウェアに関してステークホルダーはそれぞれ関心事が異なります。全員がアーキテクチャを理解できるようにするには、ソフトウェアのあるべき姿を良好に文書化するだけでなく、実装が計画から逸脱しないようにする必要があります。

Axivionによるアーキテクチャ検証のしくみ

Axivionでアーキテクチャ検証を設定するのは4ステップで済みます。

ステップ1
アーキテクチャモデルを作成

既存の機械可読なアーキテクチャモデルをインポートするか、内蔵モデラーを使います。

ステップ2
コードモデルを作成

ソースコードからコードモデルを抽出します。

コードモデルはエンティティ（ソースファイルなど）とクラスや関数、そしてそれらの依存関係を表します。Axivionはソースコードプロジェクトを解析し、自動的にコードモデルを構築できます。

ステップ3
コードをアーキテクチャへマッピング

コード要素がアーキテクチャのコンポーネントにどう対応するかを定義し、コード要素をアーキテクチャ要素に割り当てます。プロダクト構造やアーキテクチャモデルに応じて、次の方法で実行できます。

手動（例：Gravisモデラーを利用）

命名規則や階層情報を使ってAxivionで自動化

モデルに含まれる情報（例：タグ値）を利用

ステップ4
依存関係を解釈

このステップでは、アーキテクチャモデル内の依存関係を解釈し、その意味を定義します。つまり、アーキテクチャの依存関係がコードの依存関係とどう一致するかを指定します。単純な例としては「コンポーネントAはコンポーネントBの関数を呼び出せる」があります。もちろんUMLモデルでは、提供/要求インターフェースなど、より高度な依存関係の解釈も可能です。

初期設定が完了すると、Axivion for CUDAはアーキテクチャを検証し、次を特定します。

収束

不一致（逸脱）

欠落

この自動検証は網羅的で信頼性の高い結果を提供し、どう対応するかを判断できます。

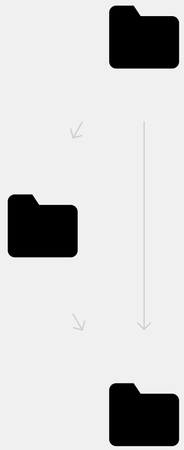
これは一度限りの検証ではない点に注意してください。Axivion for CUDAの特長は、ソフトウェア開発プロセスの中で継続的にアーキテクチャ検証を行えることです。コードをプッシュまたはコミットする前に、開発者は自分のコードが意図したアーキテクチャと一致しているか確認でき、クリーンな状態を保てます。

ソースコードの逸脱を修正する

この不一致がコードのミスによるものなら、アーキテクチャに合うようコードを修正すべきです。

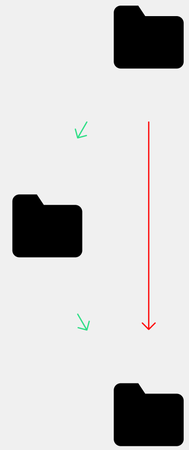
アーキテクチャ仕様を更新する

この不一致においてコードに妥当性があるなら、アーキテクチャを更新すべきです。



一時的に逸脱を受け入れる

一時的に許容できる不一致なら、そのままにして後で対処します。



アーキテクチャチェック

Axivionは継続的なソフトウェア解析・開発プロセスの一部になるよう設計されています。セットアップ後は日常のレビューに容易に統合でき、人手によるミスを招きやすいレビューを、網羅的で信頼性の高い自動チェックに置き換えます。結果はダッシュボードに集約されソースコードにリンクされ、開発者のIDEにも直接表示できます。開発者は逸脱を確認し、必要に応じて直接修正できます。

Axivionの差分解析はコードベースの変更に焦点を当てます。毎回すべての問題を確認するのではなく、最新のコードで生じたものだけに絞って素早く集中してレビューできます。そのためにベースライン（通常は前回のチェック）を設定し、新しいコードがコミットまたはプッシュされると、Axivionが現在版と前回版の差分を特定します。

開発者はコミット・ビルド・ブランチ間などの変化を即座に把握でき、全解析の完了を待つ必要がありません。新しい問題だけに集中して修正でき、コードレビューやデバッグに費やす時間を大幅に節約しつつ、アーキテクチャの一貫性を維持できます。

結果：より長いライフサイクルに耐える信頼性の高いソフトウェアアーキテクチャ。

アーキテクチャリカバリ

ソフトウェアのアーキテクチャ全体像が見えなくなる理由はさまざまです。長年にわたり異なる人がドキュメントの異なる部分を更新し、全体像がわからなくなったのかもしれません。あるいはプロジェクトが小さく始まり、アーキテクチャの文書化が優先されなかったのかもしれません。しかし機能をいくつも追加すると、特に新メンバーにとってソフトウェアを理解するのが難しくなります。

最も多いケースは、サードパーティからのコードを組み込むものの、必要なドキュメントがない、あるいは不完全・古い場合です。それでもコードを有効に使い、最高の品質基準を満たすには、ソフトウェアの設計と各部分の関係を理解することが重要です。ここでAxivionのアーキテクチャリカバリが役立ちます。

チーム内にある断片や仮説をもとに、アーキテクチャの仮説を立てます。Axivionは実装との比較によりこの初期仮説を検証し、逸脱を反復的に解消します。このプロセスは、アーキテクチャチェックと同じ技術・機能を使います。

全くドキュメントがない場合はどうするのでしょうか？
その場合もAxivionはアーキテクチャ考古学で支援できます。

アーキテクチャ考古学

アーキテクチャ考古学はアーキテクチャリカバリの極端な形です。Axivionを使えば、一見不可能に思える、リリース後ずいぶん経ったソフトウェアのドキュメント化を実現できます。

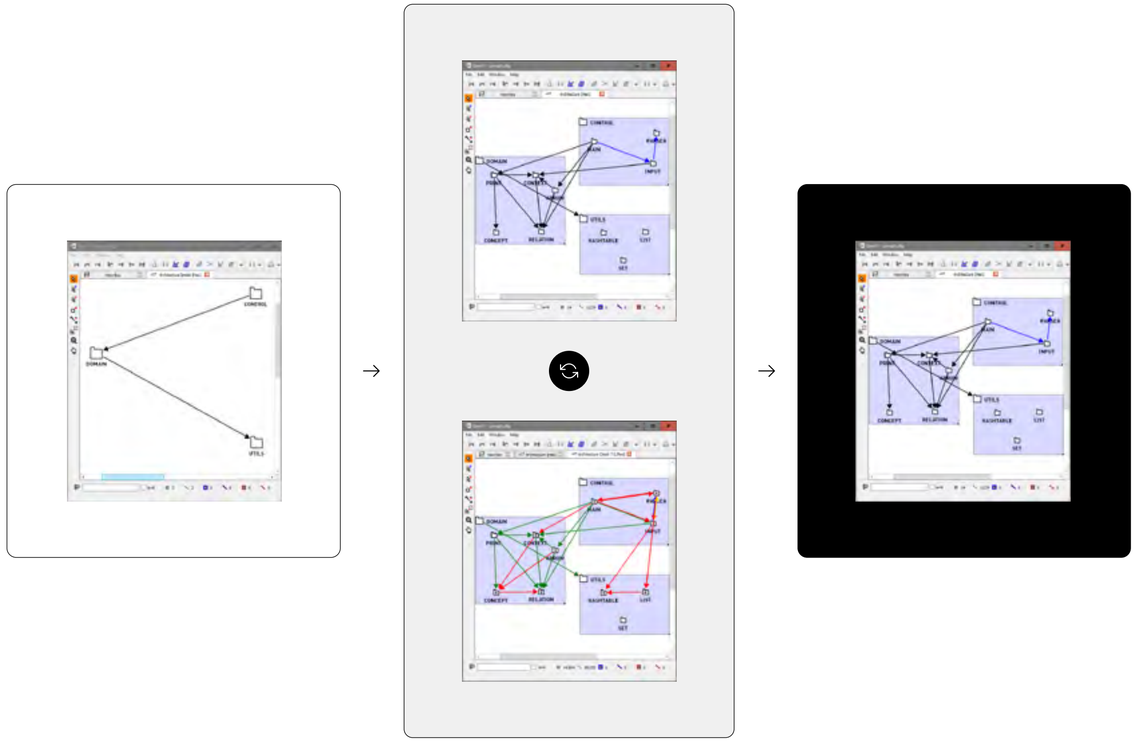
頼れるドキュメントがない場合、最初のステップは仮説を記述し、マッピングを定義することです。インポートすべきドキュメントがないので、付属のAxivionエディタで最初の仮説を描きます。

次に、自動チェックを実行し、コード内の仮説からの逸脱を見つけます。

3番目のステップでは、アーキテクチャ違反に優先順位を付け、必要な調整を行い、必要であれば仮説を洗練することで逸脱を評価します。

2番目と3番目のステップである自動チェックと逸脱の評価を、アーキテクチャが正確になるまで必要な回数繰り返します。

アーキテクチャの正しさが検証されれば、日々のアーキテクチャチェックでドリフトを防ぎ、ソフトウェアエロージョンを抑えられます。



CUDA開発の一般的なアーキテクチャの落とし穴

CUDAのソースコードでは、ロジックはホストコード（CPUで実行）とデバイスコード（GPUで実行）に分かれます。CUDAの重要な概念にカーネルがあります。これはGPU上で実行され、ホストから起動される関数です。

カーネルの呼び出しではグリッドとブロックの次元を指定し、スレッドをGPUハードウェア上にどう起動・分散するかを定義します。この呼び出しプロトコルは、ターゲットアーキテクチャやカーネルのロジックに基づいて慎重に構成する必要があります。

設定が不適切だと、

- 並列性が活かせず**性能が低下する**
- **バッファ境界外アクセスなどの実行時エラー**

といった問題につながります。

もう1つの重要なポイントはCUDAのデバイスメモリ階層です。ブロック内の全スレッドがアクセスできる共有メモリは、グローバルメモリより低レイテンシです。ただしホストからデバイスに転送されるデータは、最初はすべてグローバルメモリに置かれます。

一般的なパターンは次のとおりです。

1. **ホストからデバイス（グローバルメモリ）へデータを転送**
2. **カーネル冒頭で共有メモリにコピー**
3. **計算を実行**
4. **結果をグローバルメモリへ書き戻し**

よく設計されたアーキテクチャでは、計算負荷の高いロジックを起動する前に、カーネルのラッパー関数や前処理ステップ（例：copyToShared()）を強制することでこの流れを支援できます。

開発者は生産性と性能を高めるため、定評あるCUDAライブラリに頼ることがよくあります。

代表的なものには次があります。

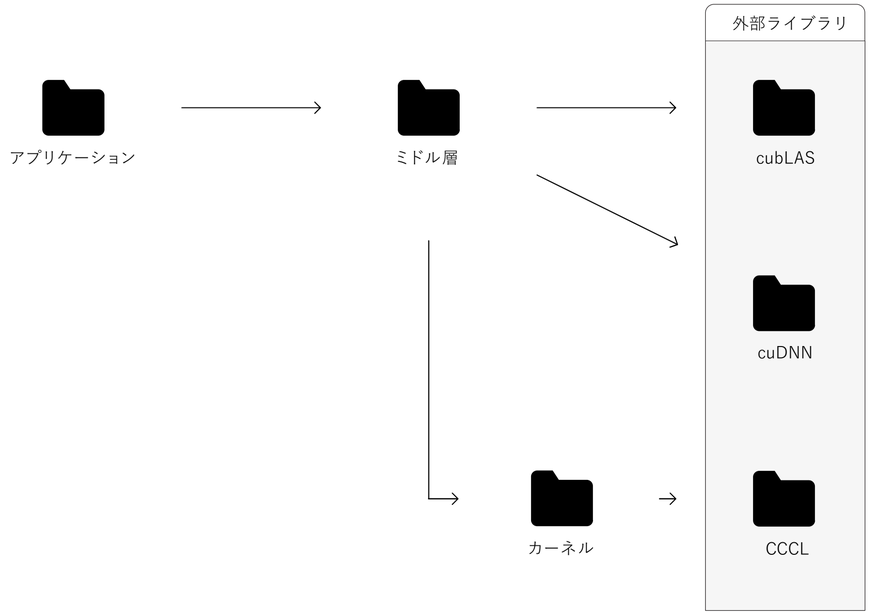
CUDA C++ Core Libraries (CCCL)：C++ STLに近い抽象化とアルゴリズムユーティリティを提供	**cuBLAS**：最適化された線形代数処理を提供	**cuDNN**：ディープニューラルネットワークと推論向けに設計

外部依存に共通して重要なのは、プロジェクトのアーキテクチャにおいてこれらのライブラリの利用を明示的にモデル化することです。一般的な手法として、明確なソフトウェア層を定義します。

カーネル層 デバイスコード（CUDAカーネル）を含み、CCCLユーティリティを使用する場合があります。	**外部ライブラリ層** CCCLやcuBLASといった外部ライブラリを含みます。
ミドル層 カーネルやcuBLASのような外部ライブラリを基に上位の計算処理を提供します。	**アプリケーション層** ビジネスまたはドメインロジックを含み、ミドル層のみを利用します。

この分離によりモジュール性が高まり、プラットフォーム間の移植性も向上します。テストや保守を簡素化する助けにもなります。

CUDAベースプロジェクトの高水準レイヤードアーキテクチャ



4

経済的優位としてのソフトウェア品質

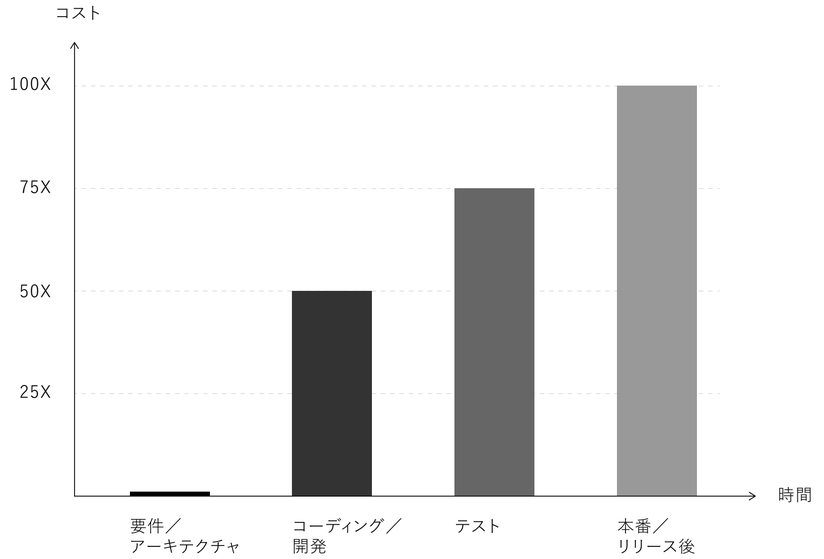
経済的優位としてのソフトウェア品質

ソフトウェアエロージョンにできるだけ早く対処し品質を高めること（→第1章）に加え、経済的なメリットも大きいです。

「時は金なり」という言葉は18世紀に遡りますが、250年以上経った今も真実です。アーキテクチャを理解する時間やバグを修正する時間は、社員に支払う時給に置き換えられます。

バグは、実装中に見つかれば設計中に見つかった場合の約6倍、製品版で見つかると最大100倍のコストがかかると研究で示されています。開発工程の早い段階で欠陥を検出する「シフトレフト」によって、バグ探しや修正ではなくイノベーションに時間を使え、その方がはるかに生産的で経済的です。

バグ修正コストの相対値（発見時期別）



これは、ソフトウェアのアーキテクチャ構造の理解やメンテナンス全般にも当てはまります。信頼できるソフトウェアドキュメントは、アーキテクチャの理解を早めるだけでなく——つまり費用を節約するだけでなく——新機能の計画にも役立ちます。

変更の影響を「想像する」のではなく「把握する」ことは、後々の不快なサプライズを防ぎます。コードの一部の変更が他の部分の不具合につながると、両者の関連付け、対策検討、実装、テストに何時間も費やすことになります。締め切り厳守や予算内完了が不可能になり、場合によってはプロジェクト計画全体を練り直す必要すら出てきます。

最初から信頼できる計画があれば、市場投入までの時間を短縮し、新製品や改善版からすぐに収益を得られます。競合より速く最新アップデートを市場に届けられれば優位性も得られます。

42%

ソフトウェア保守

~33%

作業時間の増加

ROI/TCOの改善：より少ない投資でより多くを得る

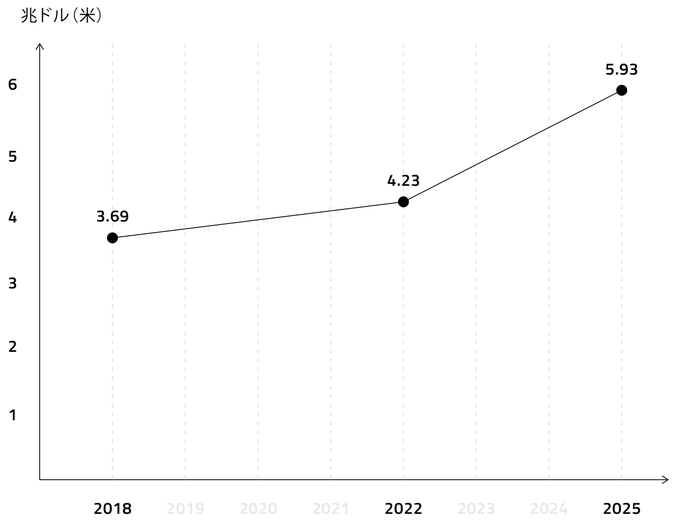
上記から、ソフトウェア品質への投資は、開発リソースを効率的に使うことでROIを高めると分かります。高品質なコードは保守・拡張・リファクタリングが容易で、長期的な機能拡張やアップデートのコストを削減します。

総所有コスト（TCO）にも大きな影響があります。品質が低いと保守コストが増え、停止も頻発し、サポート負荷も高まります。一方、品質重視の開発はシステムを安定させ、事後対応の必要性を減らし、チームがイノベーションに集中できるようにします。

2020年のCISQ（Consortium for Information & Software Quality）の数字を2025年に外挿すると、米国だけで技術的負債によるコストは約1.84兆ドルに上ります。世界全体では5.93兆ドルにも達します！

これは、ソフトウェアエロージョンを無視する余裕がないことをはっきり示しています。

ソフトウェア開発 **2018–2025**



5.93兆ドル

低品質なソフトウェアは
世界で5.93兆ドルの損失を生みました。

ベストな人材にベストパフォーマンスを発揮してもらう

高品質なコードベースは理解・変更・拡張が容易です。これはオンボーディングに直接効き、新しい開発者が早く戦力になります。開発者にとって、コード品質や保守性は仕事の満足度を左右する重要要因の上位に挙げられます。優秀な人材の確保が難しい今、定着は特に大切です。

高品質なコードで働けば、開発者は本番障害の火消しに費やす時間が減り、有意義な仕事に集中できます。それが高い定着率と意欲的なチームにつながります。また、エンジニアリング文化の強い企業はトップ人材を引き寄せ、競争力をさらに強化します。

結果として、頻繁な人員入れ替えによる混乱が減り、人事コストも下がります。

創り上げたものを守る

ソフトウェアエラーによる金銭的損害は甚大になり得ます。失われた収益、賠償請求、規制罰金、修正費用など直接的コストが含まれます。これらは多くの場合定量化でき、それだけでも重大ですが、影響はそれにとどまりません。失敗による評判の失墜はさらに破壊的で、測りにくいものです。

ネガティブな報道、ソーシャルメディアでの批判、口コミによる非難は、短時間で世間の信頼を損ないます。一度企業イメージが傷つくと、立て直しは長く不確実な道のりになります。多額のPRキャンペーンや顧客対応、ブランド回復に何年も投資する組織もあれば、完全に回復できない組織もあります。そのダメージは波及します。顧客は製品への信頼を失い購買をやめ、ビジネスパートナーやサプライヤーは関係を見直し、契約解除や条件厳格化に動くかもしれません。

極端な場合、評判被害が悪循環を引き起こします。売上減が収益縮小を招き、それが事業、雇用、イノベーションに影響します。投資家が資金を引き揚げ、ステークホルダーが経営陣の交代を求めることもあります。技術的なエラーが全面的な危機へと発展し、企業が縮小・再編、あるいは完全な閉鎖に追い込まれることさえあります。

つまり、失敗のコストは単なる金銭的損失ではなく、企業の存続に関わります。

5

最重要領域：安全性が要求される環境でのCUDAアプリケーション

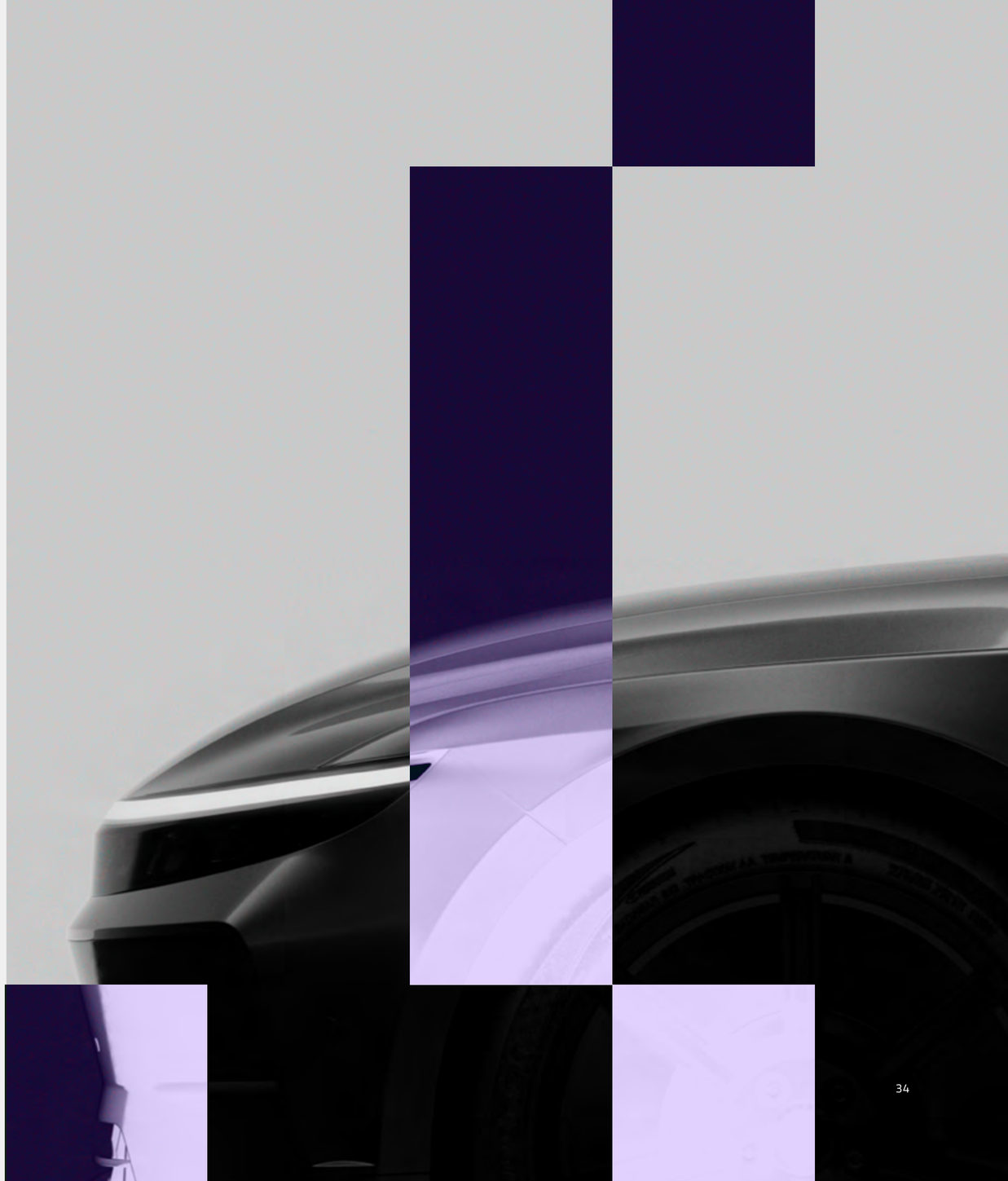
安全性が要求される環境でのCUDAアプリケーション

既に述べたように、CUDAの人気は高まっています。これは複数の産業で見られる傾向です。CPUベースのHPC（高性能コンピューティング）コードのGPU移行に始まり、今日GPUはグラフィックスやHPC、娯楽や利便性の枠を超えています。ADAS（先進運転支援システム）や産業・医療ロボティクスなど、安全性が要求されるシステムを実現するアクセラレータとして活躍しています。これらは長年にわたる多くの規則や標準が存在する厳しく規制された業界で、遵守が必要です。

こうした環境でCUDAの役割がこれまで以上に大きくなるなかで、ユーザーコミュニティにも、性能と効率だけでなく信頼性と安全性を確保する明確で実行可能なコーディングルールが求められます。

コーディングガイドラインは、保守性が高く堅牢で予測可能なコードを作るための標準化されたルールを定めます。単なるスタイル推奨にとどまりません。ガイドラインが促す一貫性は、開発者間の誤解を減らし、新メンバーのオンボーディングを容易にし、大規模・分散プロジェクトでの品質維持に貢献します。

特に性能や並列性が複雑さを増すCUDA開発では、ガイドラインはスケール時や特定の実行条件でのみ顕在化するような微妙なバグの回避にも役立ちます。



ソフトウェアの安全性とセキュリティ

ソフトウェア開発では、安全性とセキュリティは異なるリスクを扱う重要な技術領域です。CUDAのようにGPU上で並列実行する高性能環境では、両者の重要性は特に高まります。

一般に、**安全性**とは故障があっても意図しない危害を与えずに運用できる能力を指します。技術的には、故障検出、封じ込め、復旧が焦点です。例えばロボットアームを制御するアプリで、競合状態やメモリアクセス違反が起きれば物理的損傷や怪我につながる可能性があります。こうした文脈での安全工学には次が含まれます。

- 不正なメモリアクセスを検知する静的解析
- 境界外エラーを検知するランタイムチェック
- 重要な出力を検証するための計算冗長化
- ウォッチドッグタイマーやフェイルセーフのような安全機構

一方の**セキュリティ**は、悪意ある脅威からシステムを守ることに焦点を当てます。GPUメモリの保護、無許可のカーネル起動の防止、計算中のデータ機密性の確保などが含まれます。対策としては次があります。

- ホストとデバイス間のメモリ分離
- バッファオーバーフローやインジェクションを防ぐ安全なAPI
- PCIe経由やGPUメモリに保存するデータの暗号化
- 実行権限を制限するアクセス制御

CUDAはハードウェアへの低レベルアクセスを可能にするため、特にクラウドGPUクラスターのようなマルチテナント環境ではサイドチャネル攻撃の標的になり得ます。攻撃者はタイミング変動や共有メモリを利用して機密情報を推測するかもしれません。したがって安全なCUDAプログラミングには、正しいロジックだけでなくハードウェアを意識した脅威モデリングが必要です。

安全性とセキュリティはしばしば交差します。たとえば医療機器を制御するCUDAカーネルへの不正アクセスのようなセキュリティ侵害は、安全性を直接損ないます。逆に、未チェックのポインタ参照のような安全上の欠陥が、任意コード実行に悪用される可能性もあります。

つまり安全性は故障による危害を防ぐことであり、セキュリティは意図的な危害を防ぐことです。CUDAベースのシステムでは、GPUコンピューティングの複雑さと性能重視の性質ゆえ、両方に専門的な注意が必要です。

コーディングガイドラインと業界標準は、安全性とセキュリティの双方を確保します。したがって、ソフトウェア内のC/C++コードと同様に、CUDAコードがこれらに準拠していることを確認することが重要です。

コーディングガイドラインは必須

ルールには意味があります。状況は変わりました。QA部門がバグを見つけるだけでは不十分で、今では開発者が欠陥の不存在を保証する責任を負います。さらに、故障や失敗がなくともソフトウェアが意図通り動作する際の安全性も考慮する必要があります。コーディングガイドラインは業界の専門家によって作られ、検証プロセスを支援し、求められる品質と透明性を得るための重要な手段となります。

CUDA開発の基盤となるC/C++には、よく確立されたコーディングガイドラインが複数存在します。

MISRA C:2025 / MISRA C++:2023 : 自動車業界に起源を持ち、安全関連ソフトウェア以外でも広く採用されています。

←

CERT C / C++ : セキュリティに焦点を当て、脆弱性を防ぐルールを提供します。

←

CWE : 一般的なソフトウェア・ハードウェアの脆弱性カタログに基づくルールです。

←

これらのルールセットは範囲が異なりますが、不適切な構造を防ぎ、可読性、信頼性、保守性、適応性を高めるという目的を共有しています。ガイドライン適用の微妙なポイントとして、開発者はコードの現行版だけでなく、将来の変更や、システムの一見無関係な部分に持ち込まれるリスクをどう最小化するかも考慮しなければなりません。

CUDAアプリケーションの機能安全

機能安全とは、入力に対する正しい動作に依存するシステム全体の安全性を指し、故障の検出や適切な安全措置の実行を含みます。自動車、産業オートメーション、医療機器などの安全関連システムが、故障が発生しても正しく動作することを保証します。

機能安全は次の標準で管理されています。

ISO 26262 (自動車)

IEC 61508 (産業全般)

IEC 26304 (医療)

これらの標準は、ハザード分析、リスク評価、安全ライフサイクル管理、安全機構の検証/バリデーションのプロセスを定義します。言語非依存であり、CUDAアプリケーションにも適用されます。

機能安全の再考 ↗

Freedom from Interference（干渉の排除）は、混在クリティカリティ、つまり安全クリティカルなソフトウェアと非安全クリティカルなソフトウェアが共存するシステムにおける機能安全の重要概念です。

これにより、次のことが保証されます。

異なる安全レベルのソフトウェア間で意図しない影響が起きない

データが干渉から保護される

安全機構が無関係な部分の故障や振る舞いにより損なわれない

アーキテクチャ分析で干渉排除を実証 ↗

Axivion for CUDAは、機能安全を適用する必要があるソフトウェア開発を支援します。

Axivion Tool Qualification Kit はツールチェーンの分類と「適合性確認（適合性認証）」を支援します。実行と結果評価のプロセスを自動化可能な、あらかじめ用意されたテストスイートで構成されます。これにより、機能安全要件のある環境でAxivionの適合性を特定的にチェック・検証できます。

NVIDIAのCUDA C++ガイドライン：堅牢で安全性が要求されるプログラミングのために

NVIDIA CUDA C++で作業する際にコーディングガイドラインを適用することは、ベストプラクティスというだけではなく、堅牢でセキュア、保守性の高いソフトウェアを確保するために不可欠です。明確なルールと規律あるコーディング標準がなければ、開発者は気づきにくく修正が高コストな微妙なバグ、性能ボトルネック、脆弱性を容易に持ち込みます

安全・セキュリティ工学の指針は「利用可能なら適用しなければならない」です。最先端の手法を適用しない場合、失敗時に確立された予防策を怠ったとみなされ、メーカーの責任が問われやすくなります。これは新規プロジェクトだけでなく既存プロジェクトにも当てはまります。この原則は、利用できる防護策やベストプラクティスがあるならそれを使うべきで、無視すべきでないという能動的姿勢を反映しています。

NVIDIA CUDA C++コーディングガイドラインの適用はこの考え方に沿っており、利用可能なツールと知見を最大限活用して、より安全で信頼性の高いGPUアクセラレーテッドシステムを構築できます。

ただし他のコーディングガイドラインと同様、コードがルールに準拠しているかチェックする必要があります。ここで Axivion for CUDA が登場します。Axivion は NVIDIA の CUDA C++ Guidelines for Robust and Safety-Critical Programmingへの準拠チェックを自動化します。これにより開発者は新ルールを安心して適用できます。現行・将来のアプリケーションだけでなく、既存のソフトウェアにも適用可能です。

NVIDIAのCUDA C++ Guidelines for
Robust and Safety-Critical Programming
への準拠は簡単です。

製品ツアー：NVIDIAのCUDA C++ガイドラインをチェックする方法 ↗

その他の安全・セキュリティ対策

Axivionは脆弱性検出に関連する他の対策もサポートします。Axivion for CUDAの欠陥解析は、潜在的なランタイムエラーをソースコードからチェックし、スケーラブルなデータフロー・コントロールフロー解析を含みます。次を対象とします。

バッファオーバーフロー検知

バッファオーバーフローは安全性の問題でもあり、最も一般的な脆弱性の1つです。Axivionはスタックベース、ヒープベースのメモリアタックを可能にするエラーを検出し、修正してアプリケーションを保護できます。

競合状態

複数のプロセスやスレッドが共有リソースにどうアクセスするかを管理し、同時に1つのプロセスだけがアクセスできるようにする措置を講じることは、デッドロックを避けるのと同じくらい重要です。Axivion for CUDAはこうした競合状態を回避するため、コードの不正なパターンを特定します。

汚染解析（Taint Analysis）

Axivionは、特に注意すべきデータのフローをプログラム全体で追跡し、誤用を防ぐ手助けができます。適切に設定された汚染解析を開発プロセスに組み込むことで、開発者はセキュリティリスクを事前に防止できます。

汚染解析：攻撃者に先んじて隙を塞ぐ ↗

6

CUDAで
コード解析を活用する

CUDAで
コード解析を活用する

CUDAアプリケーションは、大規模な並列計算や複雑な数学的解析を要する分野で使用されます。自動運転、人工知能（ディープラーニング、機械学習）、データサイエンス、科学計算（気候モデリング、計算化学、バイオインフォマティクス）、画像・信号処理、金融計算、医用画像、メディア&エンターテインメント、量子計算など多岐にわたります。

これらの領域はすべて、NVIDIA GPU の並列処理能力を活用して、CPUでは遅すぎる計算負荷の高いタスクを加速できるというCUDAの利点を大いに享受します。

自動運転（AD/ADAS）

AD/ADASは最高レベルの安全要件を持ちながら、CUDAの恩恵も受けます。そのため公道対応には安全クリティカルなコード保証が必須です。ISO 26262やMISRA C++と同等のコンプライアンスを得るために、コード解析はロジック欠陥、未定義動作など多くの問題を検出します。アーキテクチャ検証は個々のコンポーネント統合や、世界中に分散した開発チームのコントロール維持に役立ちます。

人工知能（AI）

CUDAはディープラーニングや機械学習に不可欠で、膨大な行列・テンソル演算を並列化して学習を高速化します。コード解析を使えば、実装時のエラーを排除し、たとえば微妙な浮動小数点の切り捨てバグを避けてモデルの正しさを高められます。さらに、境界外インデックスやバッファ誤操作のようなメモリ・セキュリティ問題を早期に対処できます。コード解析はチーム間で一貫した開発パイプラインを強制する効果もあります。コードベースは大規模化・分散化しがちなので、アーキテクチャ検証は依存関係を囲い込み、アーキテクチャ違反を防ぐ助けになります。

科学シミュレーションと高性能コンピューティング（HPC）

気候・天気モデリング：複雑な気候システムのシミュレーションはGPUの並列化で恩恵を受けます。

計算化学：タンパク質ドッキングや分子動力学シミュレーションのようなタスクを加速し、発見や解析を早めます。

バイオインフォマティクス：ゲノムデータの処理、シーケンス、遺伝子シミュレーションといった負荷の高いタスクでCUDAが性能向上をもたらします。

このドメインでは、コード解析が大規模並列コード構造を分析してスレッド安全性や競合検出に貢献します。データ型の正しい使用や変換をチェックすることで、低レベル演算での精度やデータ損失の可能性を特定できます。アーキテクチャ検証は、増大するアプリケーションコードベースをソフトウェアアーキテクトがコントロールする強力な手段となります。

メディア＆エンターテインメント

レンダリングや動画編集のようなコンテンツ制作・処理における高性能タスクを加速できます。コード解析は非効率やメモリ処理の問題を検出するヒントを提供します。アプリケーションは複数プラットフォームにまたがることが多いため、コード解析とアーキテクチャ検証の両方が良いコーディング慣行を強制し、クロスプラットフォームで欠陥のない状態を保つのに役立ちます。

画像・信号処理

医用画像（MRI、CT）や信号処理におけるリアルタイム処理・再構成は、CUDAにより大幅に高速化されます。境界チェックやオーバーフロー解析を通じて、データの危険な使い方を防ぎます。意図しない整数除算や丸め誤差など、結果を歪める原因も検出可能です。アーキテクチャ検証は一貫した処理パイプラインの維持を助けます。

データサイエンス＆アナリティクス

複雑なデータ分析、数値計算、その他のデータ集約タスクは大幅な高速化が見込め、企業はより良い意思決定を迅速に行えます。CUDA向けコード解析は、誤った型の使用や不適切な型変換、データ取込みコードでのnull処理漏れを検出します。正しく設定すれば、深いネストループの特定など性能面の指針も与えられます。同時に、アーキテクチャ検証はコードベースの高水準の見取り図を保ち、保守性を最良の状態に保てます。

金融計算

CUDAは複雑なリスク分析やその他の金融計算を加速し、トレーディングや投資判断に不可欠なリアルタイム分析を可能にします。コード解析は精度と丸め制御を維持し、浮動小数と整数型の混在を検知して警告できます。アーキテクチャ検証は、正しいAPIの使用と提供、コードが意図したアーキテクチャ構造に準拠していることを保証します。

CUDAアプリケーションを「将来に強い」状態にしませんか？

お問い合わせ ↗

ソフトウェア品質、コンプライアンス、イノベーションを同時に極める方法について、ぜひお問い合わせください。

もっと知る

リソースセンターへ移動

Axivion 評価ワークショップを依頼