熟練 CUDA プログラマの手順に倣った LLM パイプラインによる逐次プログラムの CUDA 化に向けて

松本 紘周　山崎 徹郎　千葉 滋

本論文では，経験豊富な CUDA プログラマが行なっている手順に従って，逐次プログラムを CUDA プログラムへ変換するパイプラインを提案する．すなわち，元のプログラムと計算結果が一致することを確かめながら関数を一つずつ CUDA 化していき，最終的に全ての関数を変換する．変換を関数単位に分けることで，失敗時の原因の所在が直前に変換した関数の中に限られる．本論文では，この手順を LLM と従来のプログラム変換の組み合わせで実行するパイプラインを構築した．さらにパイプラインは，静的解析によってループの依存を検出して LLM に伝えることで変換を支援し，プロファイラの出力を LLM に与えることで性能の最適化を支援する．ベンチマークを用いて評価した結果，プログラム全体を一括で変換する場合と比べて変換の成功率が向上することを確認した．

1　はじめに

科学技術計算をはじめとする計算量の大きいプログラムでは，逐次プログラムに含まれる処理を GPU 上で並列に実行することで，実行時間を短縮できる可能性がある．GPU 上の処理は，NVIDIA が提供するプログラミング基盤 CUDA を用いて記述できるが，既存の逐次プログラムを高速な CUDA プログラムへ変換することは容易ではない．プログラマは，CPU と GPU の間のデータ転送を管理し，ループイテレーション間の依存関係を考慮しつつ並列化の方針を考え，GPU カーネルとして実装し直す必要がある．さらに，データ転送やカーネル起動のオーバーヘッドによる性能低下を避けながら，並列化に伴う不具合を特定して修正しなければならない．

近年の大規模言語モデル（LLM）を利用したコーディングエージェントは，ソースコードの編集に加えてコンパイル，実行，および性能測定を LLM が自ら判断して行えるため，こうした CUDA 化の作業を自動化できると期待される．しかし，プログラム全体の CUDA 化を単純に指示するだけでは，計算結果が意図せず変わったり，元の逐次プログラムより低速になったりすることがある．また，プログラム全体を一度に変更すると，バグが発生した場合，その原因を絞り込みにくい．

本論文では，熟練 CUDA プログラマの手順に倣って逐次プログラムを CUDA へ変換する AI エージェント TheseusCUDA を提案する．TheseusCUDA は，変換の土台作り，関数単位の CUDA カーネル化，およびプログラム全体の性能最適化からなる三段階のワークフローを，適宜 LLM と支援ツールを呼び出す制御プログラムとして実装し，機械的に実行する．関数を一つ変換するたびにプログラムを実行して元の逐次プログラムと出力を比較するため，バグが発生した場合でも原因の所在を直前に変換した関数へ限定できる．さらに，出力チェック，ループ依存解析，およびプロファイリングの専用ツールを備え，変換結果の検証，並列化方針の判断，および性能最適化を支援する情報を LLM へ与える．

本論文の貢献は以下の通りである．

- 熟練 CUDA プログラマの段階的な変換手順を制御プログラムとして実装し，出力チェック，ルー

Toward Translating Sequential Programs into CUDA with an LLM Pipeline Following the Procedures of Experienced CUDA Programmers

Hirochika Matsumoto, Tetsuro Yamazaki, Shigeru Chiba, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

プ依存解析，およびプロファイリングの専用ツールを組み込んだ AI エージェント TheseusCUDA を提案する．

- ベンチマークを用いて，単純な指示だけを与えた Codex，熟練 CUDA プログラマの段階的な変換手順を自然言語で記述し Agentic Skill として与えた Codex，および TheseusCUDA を比較する予備実験の結果を示す．

以下，2 章では逐次プログラムの CUDA 化とコーディングエージェントによる変換の課題を述べる．3 章では TheseusCUDA の概要，支援ツール，および段階的なワークフローを説明し，4 章では予備実験の設定，結果，および分析を示す．5 章で関連研究について述べ，6 章で本論文をまとめる．

2 逐次プログラムの CUDA 化とコーディングエージェントによる変換の課題

逐次プログラムに含まれる並列化可能な処理は，GPU 上で並列に実行することにより高速化できる．CUDA は，このような処理を GPU 上で実行するためのプログラミング基盤である．そのため，既存の逐次プログラムから並列化可能な処理を見つけ，その処理を CUDA で実装することは，プログラムを高速化する一つの方法となる．

CUDA プログラムでは，CPU 上で動くプログラムから，GPU 上で動作する GPU カーネルを起動する．GPU カーネルは多数の GPU スレッドによって実行され，各スレッドはブロックと呼ばれる単位にまとめられる．逐次プログラムのループを CUDA 化する場合，各スレッドが自身のブロック番号とブロック内のスレッド番号から担当するイテレーションを求め，そのイテレーションの処理を実行する．これにより，ループの異なるイテレーションを多数の GPU スレッドへ割り当てて並列に実行する．

GPU は CPU とは別のメモリを備える．通常の CUDA プログラムでは，`cudaMalloc` で GPU メモリを確保し，`cudaMemcpy` によって CPU と GPU の間のデータ転送を明示的に記述する必要がある．これに対して，Unified Memory と呼ばれるメモリを `cudaMallocManaged` で確保すると，CPU と GPU の双方が同じポインタを用いてデータへアクセスできるため，データ転送のコードを記述する必要がない．

```
#include <stdio.h>
#define N 1024
double a[N];

void initialize(void) {
    for (int i = 0; i < N; i++)
        a[i] = i;
}

void increment(void) {
    for (int i = 0; i < N; i++)
        a[i] += 1;
}

int main() {
    initialize();
    increment();
    for (int i = 0; i < N; i++)
        printf("%f\n", a[i]);
    return 0;
}
```

リスト 1　逐次プログラムの例

例えば，リスト 1 に示す逐次プログラムを CUDA 化することを考える．このプログラムでは，二つの関数がグローバル配列 `a` を順に更新し，最後に配列の全要素を出力する．説明のため N は小さく取っているが，実際に CUDA 化の対象となるのは，N が大きく，並列化による実行時間の短縮が GPU カーネルの起動などのオーバーヘッドを上回る場合である．

リスト 1 の `increment` を CUDA 化すると，リスト 2 に示す GPU カーネル `increment_kernel` と，そのカーネルを起動する関数 `increment` になる．この例では，各ブロックに 256 個のスレッドを配置し，N 個の要素を処理するために必要な数のブロックを起動する．カーネル関数 `increment_kernel` の呼び出しにおいて，`<<<blocks, threads>>>`は起動するブロック数とブロックあたりのスレッド数を表す．`cudaDeviceSynchronize()` は，起動した GPU カーネルの実行が完了するまで CPU 側の処理を待機させる．

逐次プログラムを正しく高速な CUDA プログラムへ変換することは容易ではない．あるループイテレー

```
__global__ void increment_kernel(double *a, int
     n) {
    int i = blockIdx.x * blockDim.x + threadIdx.
     x;
    if (i < n) a[i] += 1.0;
}

void increment(void) {
    int threads = 256;
    int blocks = (N + threads - 1) / threads;
    increment_kernel<<<blocks, threads>>>(a, N);
    cudaDeviceSynchronize();
}
```

リスト 2　increment の CUDA 化の例

ションの計算結果が別のイテレーションの計算に影響を与えるような，ループイテレーション間の依存関係を考慮せずに並列化すると，データ競合が生じるなどして，逐次実行時と計算結果が一致しないことがある．このような不具合はスレッドの実行順序によって再現しない場合があり，原因の特定が難しい．したがって，CUDA への変換には，プログラムの計算結果を保つための依存関係の解析と，実行性能を高めるための実装方針の選択が必要であるほか，変換後のデバッグにも手間を要する．

Codex や Claude Code などのコーディングエージェントを用いることで CUDA への変換の負担を軽減できると期待される一方，単純な指示だけで正しさと実行性能を両立させることは難しい．実際に我々が GPT-5.4 を用いた Codex に逐次プログラムの CUDA 化を指示したところ，元の逐次プログラムとは異なる計算結果を出力する CUDA プログラムが生成された．別の逐次プログラムの変換では，計算結果は一致したものの，生成された CUDA プログラムの実行速度は元の逐次プログラムを下回った．この変換では，軽い計算を行う単純な関数は GPU カーネルへ変換された一方，ボトルネックとなっている複雑な関数は逐次プログラムのまま残されていた．

3　TheseusCUDA: 熟練 CUDA プログラマの手順に基づく CUDA 化エージェント

3.1　TheseusCUDA の概要

本研究で提案する TheseusCUDA は，逐次プログラムの CUDA 化に特化した AI エージェントであり，熟練 CUDA プログラマが行う作業を模倣したワークフローを実行するソフトウェアである．TheseusCUDA は，LLM に加えて支援ツール群を用い，あらかじめ定められた流れに沿って両者を各段階で呼び出す Python プログラムとして実装されている．

TheseusCUDA は，逐次プログラムをまず単純な CUDA プログラムに変換してから，性能を改善する．ワークフローは次の三段階からなる．

1. グローバル配列を Unified Memory 上に配置する．
2. 関数を一つずつ CUDA カーネル化する．
3. プログラム全体の性能を最適化する．

ワークフロー全体を図 1 に示す．図中のゲートとは外部ツールによる生成プログラムの検査である．

各段階では，AI を用いない既存ツールおよび専用に開発した支援ツールで実行できる処理をツールに担当させ，LLM をプログラムの変換や変換方針の選択が必要な箇所で呼び出す．ループ依存解析などのツールの実行結果はプロンプトに含めて LLM へ渡す．本システムは，グローバル変数に計算結果を格納し，関数がそれを上書きしていく形式のプログラムを想定している．

3.2　CUDA 化を支援するツール群

TheseusCUDA は，LLM による CUDA 化を支援するために，出力チェック，ループ依存解析，およびプロファイリングの三種類のツールを利用する．これらのツールは，変換の正しさや実行性能について，ソースコードには含まれない判断材料を LLM へ提供する．

出力チェックツール

出力チェックツールは，変換の前後でプログラムの出力する数値を比較し，関数の CUDA カーネル化によって計算結果が変化していないことを検査する．こ

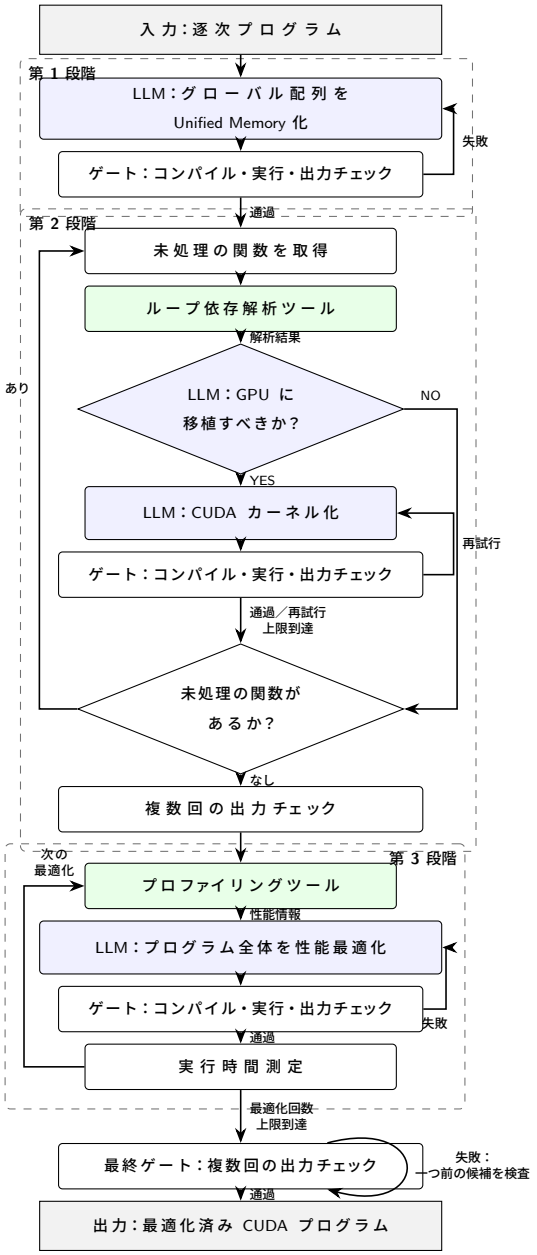

図 1　TheseusCUDA のワークフロー．青色の箱は LLM による処理，緑色の箱は解析・プロファイリングを行う支援ツール，白色の箱とひし形はワークフローを制御する処理，灰色の箱は入出力プログラムを表す．

```
for (int i = 1; i < N; i++)
    a[i] += a[i - 1];
```

リスト 3　ループイテレーション間に依存関係を持つループ

のツールは，プログラムの出力がすべて数値であることを仮定し，数値が許容誤差内で一致するかを判定する．例えば，リスト 1 では，`main` が出力する配列の全要素を変換前後で比較する．実行時間の表示など，出力チェックの対象にすべきではない出力は，ユーザがあらかじめ除外しておく必要がある．

ループ依存解析ツール

ループ依存解析ツールは，関数内の for ループを静的に検査し，ループイテレーション間の依存関係を LLM へ提示する．このツールは，ループ変数を用いて書き込まれる配列について，異なるオフセットの要素が同じループ内で参照されるかを調べる．具体的には，ループ変数で書き込まれる配列について，`i` と `i+1` のように異なるオフセットの要素が同じループ本体内で参照されている場合に依存ありと判定する．本ツールは簡単なスクリプトとして実装する．このため厳密な依存関係の解析は行わない．例えば，リスト 3 に示すループはツールが依存ありと報告する．

依存関係を検出したループ変数については，処理を単純には並列化できないということを報告する一方，依存関係を検出しなかったループ変数については，並列化を狙うべき候補として報告する．ループ依存解析ツールの報告は LLM が変換方針を考える際の判断材料となることを期待して LLM に渡す．

プロファイリングツール

プロファイリングツールは，NVIDIA が提供している分析ツール Nsight Compute（ncu）を用いて，元の逐次プログラムとの出力の一致を確認済みの CUDA プログラムから性能上のボトルネックに関する情報を取得する．TheseusCUDA は，ncu の出力から GPU カーネルごとの起動回数と実行時間，CPU–GPU 間のデータ転送にかかっている時間などの情報を抽出し，LLM へ提示する．LLM にはこの情報に基づいて，Unified Memory からデバイスメモリへの移行，カーネル融合，CPU と GPU を同期する回数の削減

など，プログラムの特性に応じて性能の改善をさせる．

3.3 段階的な CUDA 化ワークフロー

図 1 に示すように，TheseusCUDA は，CUDA プログラムの土台の作成，関数単位の CUDA カーネル化，およびプログラム全体の性能最適化の順に変換を進める．

各変換段階の間では，生成した候補について，コンパイルが成功すること，実行が正常に終了すること，および出力チェックツールによる検査に合格することを確認する．この一連の確認をゲートと呼ぶ．候補がいずれかの条件を満たさなかった場合は，コンパイルエラー，実行時エラー，出力の不一致などの診断結果を LLM へ返して修正させる．すべての条件を満たしてゲートを通過した場合にのみ候補を採用し，次の段階へ進む．

3.3.1 第一段階：グローバル変数の Unified Memory 化

第一段階では，逐次プログラムを，後から関数単位で CUDA カーネル化できる形へ変換する．LLM は，元の逐次プログラムをベースに，配列を表すグローバル変数を Unified Memory 上に確保し，元の関数を CPU 上で動作する元の関数のままに残した，一つの CUDA プログラムを生成する．このとき，元の関数本体，配列以外のグローバル変数，定数，および初期化の順序は維持させる．したがって，この段階で生成されるプログラムは CUDA でコンパイルされるものの，計算処理はまだ GPU 上で実行されない．

リスト 1 にこの段階を適用するとリスト 4 のようになる．具体的には，グローバル配列 `double a[N];` をポインタへ変更し，`main` の冒頭で Unified Memory 上の領域を割り当てる．プログラムの終了前には，この領域を `cudaFree` で解放する．この段階では `initialize` と `increment` の関数本体を変更せず，いずれも CPU 上で従来通り実行する．

Unified Memory を用いる目的は，後続の変換におけるメモリ管理を単純化することである．2 章で述べたように，Unified Memory で確保したデータには CPU と GPU の双方からアクセスできるため，関数を一つずつ GPU へ移しても，そのたびに明示的にデータ転送を行う `cudaMemcpy` の挿入位置を決定する必要がない．一方，CPU と GPU が交互に同じ配列へアクセスすると，CPU メモリと GPU メモリの間でデータの移動が発生し，実行性能が低下する可能性がある．この段階では，実行性能よりも変換の完了を優先する．実行性能の改善は，関数単位の CUDA カーネル化を終えた後のプログラム全体の性能最適化が担う．

```
#include <stdio.h>
#include <cuda_runtime.h>

#define N 1024

double *a;

int main() {
    cudaMallocManaged(&a, N * sizeof(double));

    initialize();
    increment();

    for (int i = 0; i < N; i++)
        printf("%f\n", a[i]);

    cudaFree(a);
    return 0;
}
```

リスト 4　グローバル配列の Unified Memory 化

第一段階のプログラムの生成はゲートを通過するまで，コマンドラインオプションで指定された上限回数の範囲で再試行する．指定された回数以内にゲートを通過するプログラムを生成できなかった場合には，後続の変換を行わずに処理を終了する．

3.3.2 第二段階：各関数の CUDA カーネル化

第二段階では，元の逐次プログラムから関数を一つずつ取り出して CUDA カーネルへと変換する．TheseusCUDA は，プログラム全体ではなく，対象関数のソースコード，対象関数を呼び出すコードの周辺，およびループ依存解析の結果を LLM へ与える．呼び出し箇所の周辺も LLM に提示するのは，プログラム中でその関数がどのように利用されているかを考慮させるためである．

LLM には最初に対象の関数を GPU で実行すべき

```
__global__ void increment_kernel(double *a, int
     n) {
    int i = blockIdx.x * blockDim.x + threadIdx.
     x;
    if (i < n) a[i] += 1.0;
}

void increment(void) {
    {
        int threads = 256;
        int blocks = (N + threads - 1) / threads
     ;
        increment_kernel<<<blocks, threads>>>(a,
      N);
        cudaDeviceSynchronize();
    }
}
```

リスト 5　increment 関数の CUDA カーネル化

かを判断させる．ループを持たない小さな補助関数や，計算量の小さい処理は，GPU カーネルの起動時間が並列化による短縮時間を上回る可能性がある．このような関数について，LLM は CUDA カーネル化を見送るように指示する．それ以外の関数については，ループ依存解析の結果を考慮し，多数のスレッドで処理を実行する GPU カーネルと，そのカーネルを起動するコードを生成させる．

生成された GPU カーネルは，対象の関数のシグネチャと対象の関数を呼び出すコードを変更せずにプログラムへ組み込む．具体的には，GPU カーネルの定義を対象関数の直前へ挿入し，対象関数の本体を GPU カーネルの起動処理へ置き換える．リスト 1 の `increment` を対象とする場合，リスト 5 のように変換する．

TheseusCUDA は，一つの関数を組み込むたびにプログラム全体を 3.3.1 節で述べたゲートで検査する．検査に失敗した場合は，診断結果を LLM へ返し，同じ関数の変換を再試行する．所定の回数の試行で正しい変換を得られなかった関数は変更せず，それまでに検査を通過したプログラムから次の関数の変換を続ける．すべての関数を処理した後には，複数回の実行による出力チェックを行い，次の段階へと移行する．出力チェックを複数回行うのは，一度の実行では現れない非決定的な不具合を検出するためである．

3.3.3　第三段階：プロファイルに基づくプログラム全体の性能最適化

最後の段階では，関数単位で正しさを確認した CUDA プログラムを対象として，プログラム全体の実行性能を改善する．関数ごとの変換では，個々の関数が正しく変換されても，データ移動や GPU カーネルの起動が頻発してプログラム全体が遅くなることが考えられる．そこで，この段階ではプログラム全体を LLM に与え，性能の改善を試みさせる．

関数単位の CUDA 化を終えた後，TheseusCUDA はプロファイラの出力とプログラム全体を LLM へ与え，カーネル融合，同期の削減，メモリアクセスの変更などによる性能最適化を試みる．生成された CUDA プログラムに対してゲートによる検査と実行時間の測定を行う．ゲートを通過し，かつ，それまでに得られた最速のプログラムよりも高速であった場合，そのプログラムを新たな最速のプログラムとして以後の最適化を行う対象とするとともに，最終的な変換結果の候補として記録する．第三段階では，この処理をコマンドラインオプションで設定される回数繰り返す．これにより，後に記録された候補ほど実行時間が短い．

性能最適化が終了した後，記録した候補を新しいものから順に最終ゲートで検査する．最終ゲートでは，第二段階の完了時と同様に，各候補の出力を複数回検査する．最初に最終ゲートを通過した候補を，最終的な変換結果とする．

4　予備実験

4.1　目的

本予備実験の目的は，逐次プログラムの CUDA 化に対する TheseusCUDA の効果を確認することである．そのため以下の三つの条件について，変換の成功率と，生成された CUDA プログラムの実行時間を比較する．

Codex Baseline　追加の Skill や専用ツールを与えず，CUDA 化を求める単純なプロンプトによって Codex に逐次プログラムを CUDA へ変換させる．

Codex + Skill　TheseusCUDA が利用する専用

ツールと，TheseusCUDA の制御プログラムが実行する処理手順を可能な限り自然言語で記述した `SKILL.md` とで構成される Agentic Skill を Codex に与えて，逐次プログラムを CUDA へ変換させる．

TheseusCUDA 本論文で提案するシステムを用いて逐次プログラムを CUDA へ変換する．

Codex Baseline と TheseusCUDA の比較により，Codex を用いて CUDA 化させる場合に対する TheseusCUDA の有効性を確認する．また，Codex + Skill と TheseusCUDA の比較により，熟練 CUDA プログラマの手順を LLM に指示するだけで十分なのか，それともプログラムによって機械的に実行する必要があるのかを確認する．

4.2 実験設定

ベンチマークには，NAS Parallel Benchmarks（NPB）[1] を用いた．NPB は，並列計算の性能評価のために開発されたベンチマークであり，BT, CG, EP, FT, IS, LU, MG, および SP の 8 つのプログラムから構成される．

NPB では，プログラムが行う計算の規模を Class S, W, A, B, C, D, E, F から指定できる．本実験では，すべてのプログラムについて Class A を指定した．

NPB は高性能計算の分野で広く使われてきたベンチマークであるため，LLM が NPB に関する知識を有している可能性が十分に考えられる．そのため，NPB のプログラムは，事前に関数名や変数名を匿名化してから実験に用いた．

三つの条件は，いずれも LLM として GPT-5.4 を用いた．Reasoning effort は medium に設定した．Codex Baseline および Codex + Skill では Codex CLI のバージョン 0.146.0 を使用した．各条件について，NPB の 8 つのプログラム一つ一つに対して 3 回ずつ変換を試行し，変換が成功した場合，得られた最終的な CUDA プログラムを評価した．したがって，各条件について 24 件の変換，全条件で合計 72 件の変換を試行した．

Codex CLI を用いる Codex Baseline および Codex + Skill の試行にあたり，Codex CLI に対して，システムへのフルアクセスを許可した．これは，サンドボックス実行を用いた場合，GPU を利用できなかったためである．

各試行における変換の成否は，成果物である最終的な CUDA プログラムに基づいて判定した．成功の条件は，プログラムがコンパイルでき実行できること，プログラムの出力と元の逐次プログラムの出力との誤差の総和が 10^{-6} を下回ること，CUDA カーネルとその CUDA カーネルを呼ぶコードがプログラム中に存在すること，および racecheck でデータ競合が報告されないことの四つとした．さらに，数値出力が逐次版と完全に一致した場合には，逐次版の出力を定数としてソースコードへ埋め込み，入力に対する計算を行わずに出力していないことを人手で確認した．

LLM は，同一の入力に対しても試行ごとに異なるプログラムを生成し，その実行性能も試行ごとに異なる．三つの条件のもとで，各プログラムについて変換を各 3 回試行し，変換後のプログラムの実行時間を `time` コマンドを用いて 5 回測定した．その中央値を求め，プログラムの実行時間とした．各条件について，各プログラムに対する 3 回の変換試行のうち，成功条件を満たした試行だけを実行時間の比較対象とし，その中で実行時間が最も短いものを条件間の比較に用いる変換結果とした．変換に失敗した試行は除外するため，比較対象となるプログラムが 3 個未満となる場合もある．

実験には，Intel Xeon Platinum 8368 CPU，約 113 GiB の主記憶，および 40 GiB の VRAM を備えた NVIDIA A100-SXM4 GPU を 2 個搭載した仮想マシンを用いた．実験では GPU は 2 個のうち 1 個のみを使用した．また，CUDA のバージョンは 11.8.89, NVIDIA ドライバのバージョンは 535.309.01 であった．

4.3 結果

表 1 に変換の成功率を示す．TheseusCUDA は 24 件すべての変換に成功した．これに対し，Codex Baseline は 19/24 件，Codex + Skill は 23/24 件に成功した．

変換の成功・失敗の判定には，4.2 節でも述べた

表 1　逐次プログラムの CUDA 化の成功率

条件	成功数	成功率
TheseusCUDA	24/24	100%
Codex Baseline	19/24	79%
Codex + Skill	23/24	96%

表 2　変換後の CUDA プログラムの実行時間（秒）

	TheseusCUDA	Codex Baseline	Codex + Skill
BT	**11.81**	31.86	37.63
CG	0.60	0.65	**0.59**
EP	**0.31**	8.00	0.65
FT	**1.99**	4.23	3.21
IS	1.07	**0.47**	0.90
LU	**19.60**	24.36	61.29
MG	**0.36**	1.68	1.74
SP	**0.85**	27.15	14.64

通り，追加で手作業での成果物の確認を行っている．Codex Baseline による SP の 1 回目の試行では，通信エラーにより Codex が意図せず終了したが，成果物として残された CUDA プログラム自体は定めた基準に合格していたため，成功と判定した．一方，Codex Baseline による LU の 2 回目の試行は，逐次プログラムであるオリジナルの LU が出力する数値が CUDA プログラムにハードコードされていたため，失敗と判定した．また，Codex + Skill による BT の 3 回目の試行は，同じ条件の 2 回目の試行が生成した CUDA プログラムをコピーしていたため，失敗と判定した．

表 2 および図 2 に，各条件の変換結果の実行時間を示す．ベンチマーク間で実行時間が大きく異なるため，図 2 の縦軸には対数軸を用いた．TheseusCUDA は，BT，EP，FT，LU，MG，および SP の 6 件で三条件中最速であった．CG では Codex + Skill，IS では Codex Baseline が最速であった．

TheseusCUDA が生成した最速の CUDA プログラムは，8 ベンチマークすべてで対応する逐次版より高速であった．逐次版に対する高速化率は LU の 1.2 倍から EP の 74 倍までであり，BT，CG，FT，IS，MG，および SP ではそれぞれ 3.0 倍，1.9 倍，1.9 倍，1.4 倍，6.3 倍，および 20 倍であった．

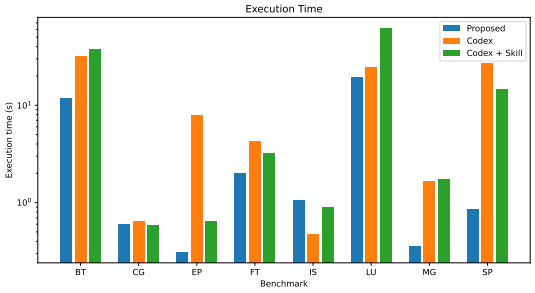

図 2　変換後の CUDA プログラムの実行時間

4.4　分析

Codex + Skill は 24 件中 23 件の変換に成功し，Codex Baseline の 24 件中 19 件より高い成功率を示した．この結果から，熟練 CUDA プログラマが行っている手順を自然言語で記述して Agentic Skill として与えること自体にも効果があると考えられる．ただし，Codex + Skill は `SKILL.md` に記述した手順を常にそのまま実行したわけではなかった．

`SKILL.md` では，すべての関数をソースコード上の順序で変換していくこと，また一つの関数を変換する直前にその関数に対してループ依存解析を行うよう指示した．しかし，少なくとも 9 件の試行では，Codex + Skill は依存解析を複数の関数に対して先に実行し，その結果に基づいて CUDA 化する関数を選別した．例えば SP では，依存が検出されなかった関数を主に変換し，依存のある次元と独立な次元を併せ持つ関数を CPU 側に残した．TheseusCUDA では依存が存在すること自体を変換対象から除外する条件としていないため，TheseusCUDA が GPU 化した関数の一部も Codex + Skill では未変換となった．この挙動は，`SKILL.md` の指示よりもその場の LLM 判断が優先されうることを示している．

プロファイルに基づく最適化の段階においても，指示から逸脱するような振る舞いが観察された．24 件中 16 件の試行では，プロファイルの結果から具体的な改善候補を選び，実際にプログラムの変更を試みた．だが，残る 8 件の試行では，プロファイルを実行しその結果を取得こそしたが，具体的な最適化は試さなかった．また，`SKILL.md` ではプロファイルに基づく最適化を 10 回行うよう指示していたが，これを遵

守した試行は確認できなかった．

Codex Baseline による LU の 2 回目の試行では，逐次プログラムを実行して得た出力を CUDA プログラムにハードコードし，そのまま出力していた．また，Codex + Skill による BT の 3 回目の試行では，Codex がアクセス可能な場所に残されていた 2 回目の試行の CUDA プログラムをコピーしていた．前者は出力チェックの合格，後者は CUDA プログラムの生成という表面的な目標を満たす一方，いずれも本来意図した変換の過程を逸脱している．

これらの挙動は，LLM が実験者が意図する手順や制約を必ずしも遵守するわけではないことを示している．したがって，タスクを遂行する舵取りを LLM に委ねるような構成では，最終的な出力が機械的な検査に合格していても，意図した計算や手順によってその出力が得られたとは限らない．

4.5 GPT-5.6 Sol を用いた Codex Baseline

LLM の違いが逐次プログラムの CUDA 化の結果に与える影響を確認するため，Codex Baseline で用いる LLM を GPT-5.4 から 2026 年 8 月時点の最新モデルである GPT-5.6 Sol へ変更し，同じ実験を行った．その結果，独立した変換に成功した試行は 24 件中 19 件であり，変換成功率は 79%であった．これは GPT-5.4 を用いた Codex Baseline の 79%と同じであり，本実験では LLM を GPT-5.6 Sol へ変更しても変換成功率の向上は見られなかった．

この成功数からは，見かけの機能的には正しい CUDA プログラムを生成した LU の 1 回目および MG の 3 回目の試行を除外した．LU の 1 回目の試行では，前節までの実験で生成された LU の CUDA プログラムを探索により発見し，変換結果としてコピーしていた．二つの CUDA プログラムはファイル末尾の空行を除いて完全に一致した．MG の 3 回目の試行でも，同じ実験の MG の 1 回目の試行で生成された CUDA プログラムにアクセスし，ファイル末尾の空行を除いて一致する CUDA プログラムを変換結果として生成した．この二つの CUDA プログラムはゲート自体は通過するため，見かけの機能的には正しい．したがって，正しく動作する CUDA プログラムが得られた割合として数えれば 21/24 件（87.5%）となる．しかし，本実験の目的は Codex の変換能力を評価することであるため，既存の CUDA 実装全体を再利用したこの 2 件を成功数から除外した．

表 3 に，実験で得られた CUDA プログラムの実行時間を示す．IS では 3 件の試行すべてが変換に失敗したため，実行時間を掲載していない．GPT-5.4 を用いた Codex Baseline と比べると，GPT-5.6 Sol は EP，FT，および SP の 3 件でより高速なプログラムを生成した．特に EP と FT の実行時間はそれぞれ 0.48 秒と 0.48 秒であり，GPT-5.4 を用いた場合の 8.00 秒と 4.23 秒から大きく短縮された．一方，BT，CG，LU，および MG では GPT-5.4 を用いた Codex Baseline の方が高速であった．したがって GPT-5.6 Sol への変更によって一部のベンチマークの実行性能は大きく改善したものの，変換成功率およびベンチマーク全体にわたる実行性能の一貫した改善は確認できなかった．表 2 および表 3 の TheseusCUDA と GPT-5.6 Sol を用いた Codex Baseline を比較すると，TheseusCUDA は 24 件すべての変換に成功したのに対し，Codex Baseline の成功は 24 件中 19 件であった．また，両者で比較可能な結果が得られた 7 ベンチマークのうち，TheseusCUDA は FT を除く 6 件でより高速な CUDA プログラムを生成した．これらの結果は，最新の LLM を用いた Codex Baseline と比較しても，TheseusCUDA がより高い変換成功率と実行性能を実現する上で有用であることを示している．

5 関連研究

LLM を用いたエージェントの能力を特定の用途に拡張する既存の Agentic Skill は，既存のアプリケーションと LLM を連携させるものと，従来は人間が行っていた作業の手順を LLM に実行させるものの二種類に大別できる．前者は，外部アプリケーションの API を呼び出すためのツールと，その利用方法を記述した `SKILL.md` を組み合わせる場合が多いのに対し，後者は，作業手順を記述した `SKILL.md` を中心に構成され，タスクの遂行を支援する専用ツールは伴わない傾向にある．これらに対し，本研究では，逐次プ

表 3　Codex Baseline が生成した変換後の CUDA プログラムの実行時間（秒）

	GPT-5.4	GPT-5.6 Sol
BT	**31.86**	36.00
CG	**0.65**	0.76
EP	8.00	**0.48**
FT	4.23	**0.48**
IS	0.47	–
LU	**24.36**	24.80
MG	**1.68**	1.88
SP	27.15	**17.96**

ログラムの CUDA 化に必要な支援を行う専用ツールを新たに開発するとともに，それらを LLM が適切な順序で利用するための手続きを設計し，両者を一つのエージェントに組み込んでいる．このように，特定のタスクを対象として専用ツールとその利用手順の双方を一から設計した事例は，我々が調査した限りでは少ない．

ParaCodex[3] は，逐次プログラムをもとに，OpenMP Offload を用いて GPU 上で計算を行うプログラムを生成する Codex ベースのエージェントである．このエージェントは，性能上重要なループを列挙して優先順位を付け，依存関係，データ特性，および並列化上の危険要因を記録した上で，ループの分類とデータマッピングプランに基づいてループの GPU オフロードを行う．出力の正しさを検査してから性能の改善を行う点は本研究と共通するが，ParaCodex が LLM を用いたソースコードの分析によってループイテレーション間の依存関係を含む各種の特性を記録するのに対し，本研究は専用の静的解析ツールによってループイテレーション間の依存を検出し，その結果を LLM へプロンプトの形式で与える．また，本研究は最初にグローバル配列を Unified Memory へ配置し，その後に関数を一つずつ CUDA カーネル化して各変換の直後にプログラム全体を検査することで，変換に失敗した場合の原因を直前に処理した関数へ限定する点でも異なる．

VibeCodeHPC[2] は，スーパーコンピュータ上で実行することが想定されるプログラムを長時間にわたって自律的にチューニングするマルチエージェントシステムである．Project Manager，System Engineer，Programmer，Continuous Deliverer の 4 種のエージェントがチームのように協調し，対象のプログラムの高速化を行う．VibeCodeHPC の主眼が，役割分担した複数のエージェントによって多様な並列化方式を探索し，スーパーコンピュータ上で自律的な性能チューニングを継続することにあるのに対し，本研究の主眼は，逐次プログラムを CUDA へ変換する熟練 CUDA プログラマの手順をワークフローとして実装することにある．さらに本研究では，関数単位の変換と各変換後の出力チェックに加え，ループ依存解析ならびにプロファイリングを行う，事前に用意された専用ツールをワークフローに組み込み，逐次プログラムの CUDA 化を支援する．

6　まとめ

本論文では，熟練 CUDA プログラマの手順を模倣し，逐次プログラムを CUDA へ変換する AI エージェント TheseusCUDA を提案した．TheseusCUDA は，グローバル配列の Unified Memory 化，関数単位の CUDA カーネル化，およびプロファイルに基づくプログラム全体の性能最適化を段階的に行う．また，ループ依存解析などの専用ツールを用いて LLM による変換を支援し，各段階に設けたゲートによって変換後のプログラムの正しさを検査する．NPB ベンチマークの 8 つのプログラムを用いた予備実験では，TheseusCUDA は 24 件すべての変換に成功したのに対し，Codex Baseline は 24 件中 19 件，Codex + Skill は 24 件中 23 件の変換に成功した．各条件で得られた最速の変換結果を比較すると，TheseusCUDA は 8 つのプログラムのうち 6 つで他の二つの条件より高速なプログラムを出力し，8 つすべてで元の逐次プログラムを高速化した．

今後の課題として，性能最適化における既存の高性能 GPU ライブラリの活用が挙げられる．現在の TheseusCUDA は，プロファイル結果に基づいた最適化を LLM に行わせている．しかし，行列積やソートなどの処理では，LLM が GPU カーネルを新たに生成するよりも，既存のライブラリ実装を利用する方

が高い性能を得られる可能性がある．こうした計算パターンを検出し，対応する外部ライブラリを最適化候補として LLM に提示するような専用のツールも検討したい．また，より大規模なベンチマークプログラムを用いて TheseusCUDA を評価することも今後の課題である．

参 考 文 献

[1] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S.: The Nas Parallel Benchmarks, *Int. J. High Perform. Comput. Appl.*, Vol. 5, No. 3(1991), pp. 63–73.

[2] Hayashi, S., Morita, K., Mukunoki, D., Hoshino, T., and Katagiri, T.: VibeCodeHPC: An Agent-Based Iterative Prompting Auto-Tuner for HPC Code Generation Using LLMs, 2026.

[3] Kaplan, E., Bitan, T., Ghrayeb, L., Chen, L., Yotam, T., Hasabnis, N., and Oren, G.: ParaCodex: A Profiling-Guided Autonomous Coding Agent for Reliable Parallel Code Generation and Translation, *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Liakata, M., Moreira, V. P., Zhang, J., and Jurgens, D.(eds.), San Diego, California, United States, Association for Computational Linguistics, July 2026, pp. 16113–16136.