修士論文

GPU における適応的距離場構築の高速化：BVH メモリレイアウトと探索アルゴリズムの最適化

谷口 大樹

主指導教員 井口 寧

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

2025 年 12 月

Abstract

本研究は，GPU 上での適応的距離場（Adaptively sampled Distance Fields, ADFs）構築の高速化を目的とする．ADFs は使用メモリ量の効率に優れたデータ構造だが，その非一様な階層構造という性質から GPU の並列処理能力の活用に課題があった．特に ADFs 構築に用いられる BVH（Bounding Volume Hierarchy）のトラバーサル処理において，(1) 非効率なメモリアクセスパターンおよび (2)BVH ノード探索時に距離場構築の特殊性を考慮しない探索アルゴリズムが，性能上の主要なボトルネックとなっていることを特定した．

本論文ではこれら課題の解決に向けて 2 つの最適化手法を提案する．第一に，BVH ノードのデータ構造を AoS（Array of Structures）から SoA（Structure of Arrays）レイアウトへ変更し，GPU のメモリアクセス効率を向上する．この変更により不要なデータ転送を削減し高速化する．第二に，クエリ点から BVH ノードの距離に基づいて探索順序を動的に決定する「近傍子ノード優先探索アルゴリズム」を導入する．これにより探索空間を早期に枝刈りし，総計算量を削減し高速化する．

提案手法の有効性の検証のため，NVIDIA GeForce RTX 3080 GPU 上で性能評価を行った．実験の結果として，SoA レイアウトの適用によって距離クエリカーネルの実行時間がベースライン比で 32%削減された．さらに近傍子ノード優先探索を組み合わせることで，同様の処理に関して実行時間が最大で 83%削減され，約 6.1 倍の高速化となった．ADFs 構築プロセス全体としては最大で 3.92 倍の高速化を実現した．この結果は，提案した 2 手法が相補的に機能し，GPU 上での ADFs 構築における性能ボトルネックを効果的に解消したことを実証している．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究の背景と重要性

距離場（Distance Fields）は3D空間内の各点から最も近いオブジェクト表面までの距離を格納したスカラー場であり，コンピュータグラフィックスや物理シミュレーション，ロボティクスなど様々な分野で重要なデータ構造として活用されている．特にオブジェクトの内外判定を可能にする符号付き距離場（Signed Distance Fields, SDFs）は，リアルタイムレンダリングや衝突検出などのインタラクティブなアプリケーションにおいて重要な役割を担う．

代表的な距離場の実現形式としては一様グリッド上での構築が挙げられる．この形式は構造が単純でありGPUによる並列処理に適しているが，空間解像度に応じてメモリ使用量が3乗で増加するという課題を抱えている．この問題に対して [Frisken et al.(2000)] が提案した適応的距離場（Adaptively sampled Distance Fields, ADFs）は，対象物体の情報量が多い領域には高い解像度，情報量が少ない領域には低い解像度を割り当てることでメモリ効率を改善している．しかしADFsの非一様な階層構造はGPU上での効率的な並列処理を困難にしている．GPUは規則的なメモリアクセスパターンに最適化されているため，不規則なツリー構造のトラバース（枝の探索処理）を伴うADFs構築は性能上のボトルネックとなっている．

1.2　従来技術と解決すべき課題

GPU上でのADFs構築を高速化する試みの中でも [Chen et al.(2022)] によるBounding Volume Hierarchy Based Adaptive Distance Fields（BADF）は，Bounding Volume Hierarchy (BVH) を中心的な加速構造として用いることで当時における最先端の性能を達成した．その一方でBADFにはGPUアーキテクチャの観点から見ると性能を律速するボトルネックが依然として存在する．本研究では，BADFによるADFs構築処理，特にその処理時間の大半を占めるBVHトラバーサル処理（処理時間割合は表2.1参照）に，以下の2つの主要な課題が存在していることを特定した．

1. **BVHメモリレイアウトの非効率性**: BADFではBVHノードを基本的なAoS（Array of Structures）形式のメモリレイアウトで保管しており，GPUのメ

モリアクセスパターンに最適化されていない．これにより，不要なデータの読み込みによるメモリ帯域の浪費や不規則なメモリアクセスが発生し，GPUの性能を十分に引き出せていない．

2. **探索アルゴリズムの非効率性**: BADF では BVH トラバースの際，子ノードを常に固定的な順序で探索する．この画一的なアプローチは不要なノードを多数探索する非効率な処理となっている．

これらの課題は，特にポリゴン数の多い大規模モデルや形状が変化する動的シーンにおいてリアルタイム性能の達成を妨げる障壁となっている．

1.3 研究の目的

上記の課題認識に基づき，本研究は GPU 上での ADFs 構築における BVH のメモリレイアウトおよびトラバーサル処理を最適化し，従来手法を上回る性能の達成を目的とする．具体的には以下の 2 点に取り組む．

1. GPU アーキテクチャに最適化された BVH メモリレイアウトを導入し，メモリアクセス効率を向上する．

2. 距離場構築の特性を利用した探索アルゴリズムを導入し，距離値計算のための BVH トラバーサルにおける計算量を削減する．

1.4 本研究のアプローチと貢献

本研究では上記の目的を達成するため，2 つの具体的な最適化手法を提案・実装した．第一のアプローチとして，GPU アーキテクチャにおけるメモリアクセス効率化の一般的な手法である SoA（Structure of Arrays）レイアウトを，本研究の BVH 構造に適用する．これによりメモリアクセスの無駄を排除し，GPU 性能を最大限に引き出す．第二のアプローチは，Best-First Search（最良優先探索）の原理を応用し，クエリ点からの距離に基づいて探索順序を動的に決定する「近傍子ノード優先探索（Closer Child Traversal, CCT）」アルゴリズムの導入である．これにより探索空間を早期に枝刈りし計算量の削減を狙う．

これら 2 つの手法を NVIDIA GeForce RTX 3080 GPU 上で実装し性能評価を行った．本研究の主要な貢献は以下の通りである．

- 提案手法により ADFs 構築のボトルネックである距離クエリカーネルをベースライン比で最大 6.1 倍，ADFs 構築プロセス全体として最大 3.9 倍の高速化を達成した．

- メモリアクセスの理論モデルと実測値を比較することで提案手法の有効性を理論的にも裏付け，性能向上のメカニズムを明らかにした．
- 最適化の過程で発生する性能指標間のトレードオフ（例：キャッシュヒット率の低下とリクエスト総数の削減）を分析し，GPU コンピューティングにおけるパフォーマンス最適化の複雑性とその中で本質的な処理量を削減することの重要性を示した．

1.5 本論文の構成

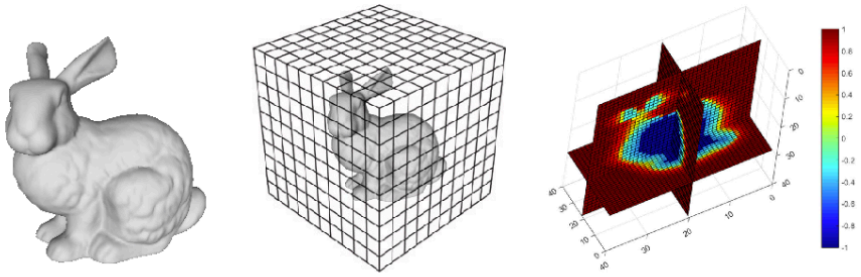
本論文は以下の構成で記述する．第 2 章では，関連研究として一様な距離場と適用的距離場，BVH と GPU に関する技術的背景を述べた後，本研究のベースラインである GPU を用いた適応的距離場構築手法とその課題について説明する．第 3 章では本研究で提案する SoA メモリレイアウトと近傍子ノード優先探索アルゴリズムについて，その実装と理論的背景を述べる．第 4 章では提案手法の有効性を定量的に評価するための実験条件，複数の 3D モデルを用いた性能測定結果，およびその結果に基づいた考察を示す．最後に第 5 章で本研究全体を総括し，得られた成果と今後の展望について述べる．

第2章　関連研究と課題

2.1　はじめに

本章では，提案手法の背景となる関連研究と本研究が解決対象とする課題について述べる．

まず本節では，距離場（Distance Fields）の基本的な概念とその形状表現における有用性について説明を行う．距離場とは，3D 空間内の各点に対して最も近くに存在する物体表面までの距離を格納したスカラー場である [Sud et al.(2004)]．このデータ構造はコンピュータグラフィックスをはじめ，物理シミュレーションやロボティクス，医療画像処理など多岐にわたる分野で応用されている．図 2.1 は距離場の基本的な概念を示している．(a) は距離場の元となる 3D ポリゴンモデル，(b) はそのモデルを内包する一様な 3 次元グリッドを表す．実際の距離場の例を示すのが (c) であり，グリッドの各点からモデル表面への最短距離を計算し，モデルの内側を負，外側を正として色分けした断面図，すなわち符号付き距離場（Signed Distance Fields, SDFs）の可視化を行っている．



(a) Surface view.　(b) Bounding volume.　(c) Generated SDF.

図 2.1: 距離場の概念図 [Slavcheva(2018)]

本研究ではこの距離場を GPU で高速に計算することを目的としている．GPU の並列計算能力を前提とすることから，次節ではその性能を理解する上で不可欠となる GPU アーキテクチャと並列プログラミングモデルの基礎について説明を行う．

その次の節では距離場構築に関する主要な先行研究について概観する．初期のアプローチである一様グリッドベースの手法から始め，そのメモリ使用量の課題を解決するデータ構造として適応的距離場（ADFs）の説明を行う．さらに ADFs の

構築を GPU 上で高速化する先行研究と，本研究の直接的な比較対象となる BVH 中心のアプローチ（BADF）を説明し，先行研究に残された課題を確認する．最後にこれらの分析に基づき，本研究が取り組むべき課題を定義する．

2.2 GPU アーキテクチャと並列プログラミングモデル

本研究の提案手法は，GPU (Graphics Processing Unit) の持つ並列計算能力とアーキテクチャ特性を十分に活用することを目指している．本節ではその背景となる GPU の並列実行モデルとメモリ階層構造について説明する．

2.2.1 SIMT: GPU の並列実行モデル

現代の GPU アーキテクチャは，実行モデルとして SIMT (Single Instruction, Multiple Threads) を採用している．SIMT では単一命令を複数のスレッドで同時に実行する．ここでのスレッド (Thread) とは，GPU 上で実行される計算の最小単位を指す．

より詳細には，GPU はこれらの複数スレッドをワープ (Warp) と言う単位で管理している．NVIDIA 製 GPU の場合，1 ワープは 32 スレッドで構成される．同じワープに属する 32 個のスレッドは単一のプログラムカウンタを共有し同じ命令を受け取る．例えばある加算命令が発行された際，ワープ内の 32 スレッドはそれぞれが持つデータに対して同時に加算処理を実行する（ただしワープ内で分岐処理が発生した場合，GPU はそれぞれのパスを直列で実行するため実行効率が低下する可能性があることに注意が必要である）．このように GPU は制御回路や命令デコーダをワープ単位で共有しており，スレッド単位での命令デコーダや制御を持つ CPU とは異なり多くのチップ面積を演算器に当てることができる．また，あるワープがメモリからのデータを読み込み待ちしている場合は，ハードウェアスケジューラがこのワープを待機させて別のワープを実行する．つまり，常に実行可能なワープを投入し続けることができれば高い演算スループットを維持することが出来る．このようにレイテンシを減らすのではなく隠蔽し，全体としてのスループットの最大化を目指すのが GPU の特徴である．

2.2.2 メモリ階層構造とアクセス効率

ここまで見たように，GPU では常に多くの実行可能ワープを送り続けることが重要である．ただし，メモリ帯域幅は有限であり，またレイテンシの隠蔽にも限界があることは重要な考慮点である．帯域幅の飽和を招く非効率なメモリアクセスや隠蔽しきれないほどのレイテンシが発生すると最終的には GPU スレッド稼働

率の大幅な低下を招く．したがって，効率的なメモリアクセスと帯域幅の活用はGPU 性能を引き出すうえで重要な課題となる．

GPU は性能と容量が異なる複数のメモリ階層を持つことでメモリアクセスの効率化を図っている．メモリの種類は，GPU コアが実装されるシリコンダイ上に直接配置されている高速なオンチップメモリと，その外部に配置されるオフチップメモリに大別できる．

- **オフチップメモリ**
  - **グローバルメモリ**: GPU 上に搭載されている大容量の DRAM．全てのGPU スレッドからアクセス可能だが，アクセス速度は低速．本研究における BVH データはこのグローバルメモリ上に格納される．
- **オンチップメモリ** (高速な順)
  - **レジスタ**: 各スレッド専用の最も高速なメモリ．スレッド（GPU カーネル）内で宣言されたローカル変数などが格納される．
  - **共有メモリ / L1 キャッシュ**: 各プロセッサ（Streaming Multiprocessor）に搭載された高速な SRAM．単一の物理メモリが共有メモリと L1 キャッシュの二つに区別して使用される．共有メモリは，複数のワープから構成されるグループであるスレッドブロック内のスレッド間でデータを共有可能であり，プログラマが明示的に使用できる．
  - **L2 キャッシュ**: 全てのスレッド間で共有されるキャッシュ．L1 キャッシュと比較してアクセス速度は遅いが容量は大きい．

GPU プログラミングにおいて高い性能を達成するためには，低速なグローバルメモリへのアクセス回数を削減しオンチップメモリを効率的に活用することが重要となる．

2.2.3 メモリコアレッシング

GPU 性能を律速する要因であるグローバルメモリアクセスを効率化するための重要な仕組みとして，メモリコアレッシング (Memory Coalescing) が挙げられる．

現代の NVIDIA GPU（Compute Capability 5 以降）のメモリシステムは，グローバルメモリアクセスを 32 byte（セクタ）を最小の物理単位として処理する．例えば 1 ワープ 32 スレッドがそれぞれ 4 byte float のデータを必要とする時，$32 \times 4 = 128$ byte となる．もしこれらのデータがメモリ上の連続した 128 byte 領域（4 つの連続したセクタ）に存在する場合，GPU は DRAM の連続データ転送機能を利用した 1 回の効率的なメモリトランザクションにまとめることができる．この仕組みがメモリコアレッシングであり，コアレス性が高い時はメモリアクセスのオーバー

ヘッドが抑えられメモリ帯域幅の実行性能が向上する．これは図 2.2 上部の図が示す状態である．一方もしこれらのスレッドがメモリ上の互いに離れたアドレス（異なる複数のセクタ）にアクセスしようとする場合，図 2.2 下部の図が示す通りハードウェアは複数の独立したトランザクションを発行する必要があり実行性能は低下する．

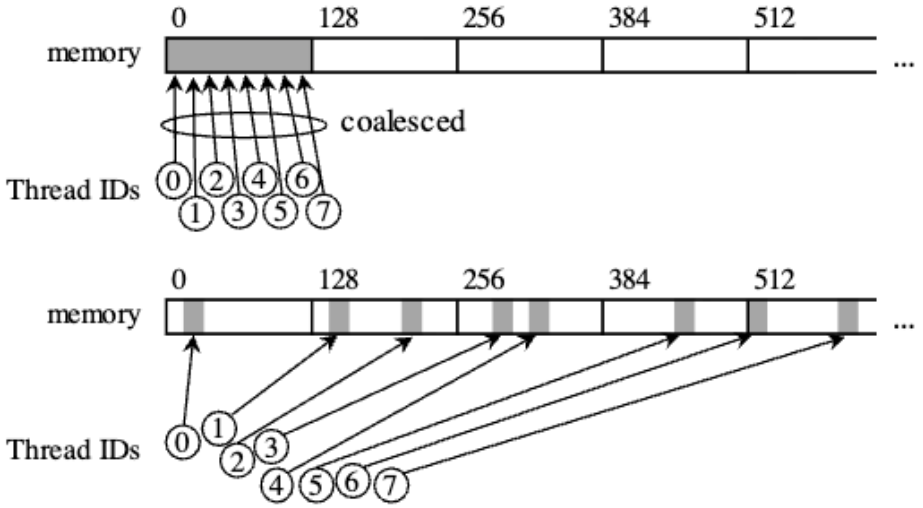


図 2.2: (上) コアレスアクセスと (下) 非コアレスアクセス [Cetin et al.(2016)]

メモリコアレッシングの恩恵を最大限に受けるためには，データレイアウト（データがメモリ上でどのように配置されているか）が重要となる．次章で詳述する AoS から SoA へのレイアウト変更は，このメモリコアレッシングを促進しメモリアクセス効率を改善することを目的としている．

2.3 関連研究

2.3.1 一様グリッドベースの距離場構築

[Rong and Tan(2006)] は GPU 上で高速に距離場を構築する研究として Jump Flooding Algorithm (JFA) を提案した．JFA は GPU の大規模な並列計算能力を活用し，ボロノイ図と距離変換（本稿における距離場）を高速に計算する．

JFA では，母点（シード）情報を各ピクセルへ伝播させる際のステップ長を対数的に変化させていく方式を採用する．すなわち $n \times n$ のグリッドに対してアルゴリズムは $\log_2 n$ 回の反復で完了する．このシード情報伝播は，図 2.3 が示す通りステップ長を倍加させていく (a) の方法と，半減させていく (b) の方法の 2 つが考えられる．

ボロノイ図および距離変換では，その処理過程ではあるピクセルにとって最近傍となり得る可能性を持つシードが複数存在することから，伝播方式の選択によっ

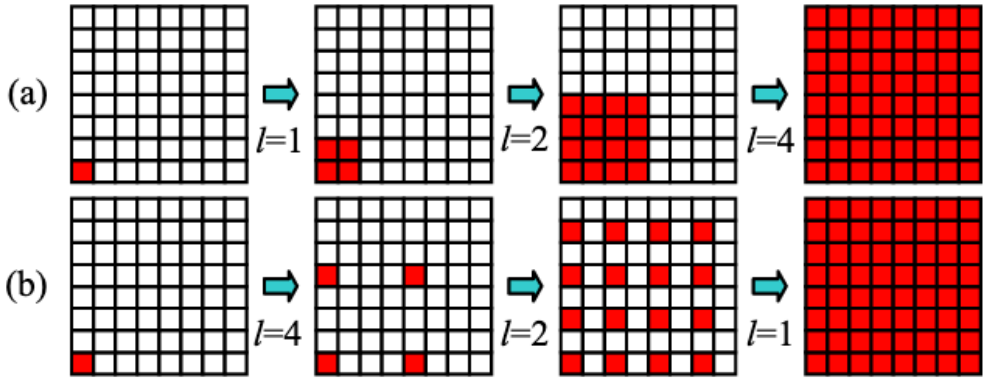


図 2.3: Jump Flooding Algorithm における情報伝播の方式比較 [Rong and Tan(2006)]

てアルゴリズムの精度が左右される．(a) の倍化方式（$l = 1, 2, 4, \ldots, n$）は直感的だが，JFA は (b) の半減方式（$l = n/2, n/4, \ldots, 1$）を採用している．ステップ長を倍加させる (a) では，初期の小さなステップでシード近傍のピクセルが早々に最近傍シードを確定させてしまう．その結果，後の大きなステップで伝播してくる他のシード（そのピクセルにとってはより遠いが，他のピクセルにとっては最近傍である可能性がある）の情報伝播を妨げてしまい，最終的に多くのエラーを引き起こす．対照的に (b) の半減方式は，まず大きなステップ長で情報を広範囲に大まかに拡散させてその後ステップを細かくして精度を上げていく．このアプローチにより本来伝播すべき情報が途中で遮断されることを防ぎ，より少ないエラーで正確な結果を得ることができる．

なお JFA で伝播される対象は距離値そのものではなく，シード情報（ID と座標）である．具体的な処理の流れは以下の通りである．

1. **初期化**: $n \times n$ のテクスチャを用意し，シードが存在するピクセルには自身の ID と座標を書き込み，それ以外のピクセルは空の状態とする．

2. **ジャンプフラッディング**: ステップ長 l を $l = n/2, n/4, \cdots, 1$ と半減させながら反復処理を行う．各反復において，全てのピクセルは並列に，自身の近傍 $(x \pm i, y \pm j)$（ただし $i, j \in \{-l, 0, l\}$）にある最大 8 つのピクセルから最近傍シードの情報を取得し，自身が保持するシード情報と比較・更新する．この際読み込みと書き込み処理の競合を防ぐため，2 つのバッファを交互に利用する Ping-Pong バッファ方式が用いられる．

3. **伝播の完了とボロノイ図・距離場の算出**: $log_2 n$ 回の反復が終了すると各ピクセルには最近傍シードの ID とその座標が格納された状態になる．この最近傍シード ID の分布がボロノイ図を形成する．図 2.4 は，64×64 のグリッド上で JFA が進行する過程を示している．初期状態ではシード点（色のついた点）のみが情報を持つが，ステップが進むにつれて各ピクセルが近傍シードの情

報を獲得していく．最終的に全てのピクセルが最近傍シードを特定し，その可視化結果としてボロノイ図が形成されている．また距離変換では最終的な距離場を得るために，各ピクセルが自身の座標と保持している最近傍シードの座標との間でユークリッド距離を計算する．

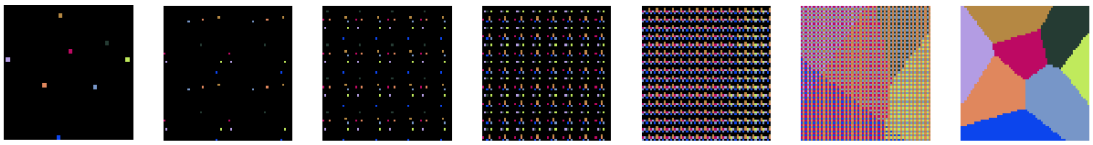


図 2.4: JFA によるボロノイ図の生成過程 [Rong and Tan(2006)]

JFA の最大の特徴は，その計算時間がグリッド解像度 n に対して $O(\log_2 n)$ であり，シードの数に依存しない点である．シード数に対して線形時間 $O(N)$ を要する従来手法とは対照的な性質と言える．

ただし JFA は一様グリッドを前提としており，高解像度化に伴うメモリ使用量が 3 乗で増加するという根本的な課題は避けられない．また JFA における近似エラーを軽減するため，反復の最後にステップ長 1 の伝播を追加する「JFA+1」などの派生手法も提案されている．

2.3.2 適応的距離場（ADFs）に関する研究

前項で述べた一様グリッドベースの手法が抱えるメモリ効率の課題に対して，[Frisken et al.(2000)] が提案した適応的距離場（Adaptively Sampled Distance Fields, ADFs）を考えることが出来る．

ADFs の核となる概念は，形状のディテールに応じて距離値を適応的にサンプリングし，それらの離散的なサンプル値から補間によって滑らかで連続的な距離場を再構成する点にある．その実装としては，サンプル値を格納する受け皿として Octtree/Quadtree に代表される階層的な空間データ構造が利用される．[Frisken et al.(2000)] において ADFs の生成は，ルートセルから開始し，セルを再帰的に分割していくトップダウンアプローチで行われる．このとき ADFs では，セル内の距離場がその 8 頂点の距離値からのトライリニア補間によって十分に近似できるかどうかを分割の判断基準とする．複数のサンプリング点において，真の距離と補間による距離の誤差が許容値を超えた場合にそのセルは 8 つの子セルに分割される．

この適応的な分割ルールにより，物体表面の曲率が低い平滑な領域は大きなセルで，コーナーやエッジなどの複雑な領域は小さなセルで密に表現される．この適応的サンプリングの効率性を図 2.5 に示す．文字「R」の形状（図左）を表現するにあたり，詳細な情報が必要な輪郭部分や角の部分ではセルが細かく分割され，形状から遠い平坦な領域では大きなセルのままであることが分かる（図右）．こ

れにより一様な高解像度グリッドを用いる場合と比較して，メモリ使用量を大幅に削減しつつも形状のディテールを高い精度で保持することが可能となっている（図中央）.

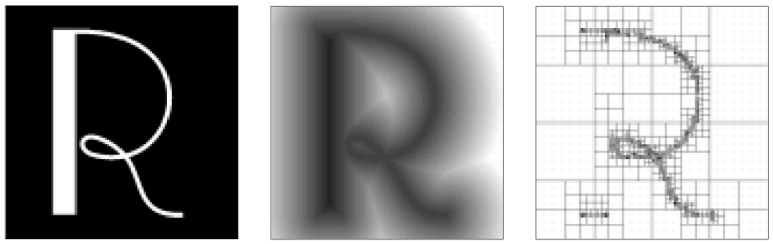


図 2.5: ADF による適応的サンプリングの例 [Frisken et al.(2000)]

このように ADFs はメモリ効率と表現精度の面で優れた特性を持つ．しかしながら，その階層的なデータ構造の構築と探索は計算コストが高く，特に動的なシーンでインタラクティブな性能を達成することが困難であった．本研究では，ADFs 構築処理の高速化によってより幅広いアプリケーションで ADFs を活用可能にすることを目的としている．

2.3.3 GPU 上での ADFs 構築

前節で示したように ADFs は優れたデータ構造である一方，階層構造の構築コストが高く，特に動的なシーンでインタラクティブな性能を達成することは困難だった．この課題に対し，GPU の並列計算能力を最大限に活用して ADF をリアルタイム CG 分野で実用化するための研究が進められてきた．

ADFs 高速化の研究において，ADFs の構築パイプラインは 2 つの主要な空間データ構造，すなわち Octree と BVH (Bounding Volume Hierarchy) を中心に構成される．空間を再帰的に分割する Octree を採用することで，形状の複雑さに応じてサンプリング密度を適応的に変化させる．一方 BVH はポリゴンメッシュを階層的な境界ボリュームで内包する構造であり，Octree によって決定された空間上の点（クエリ点）から最近傍ポリゴンまでの距離計算を高速化するための加速構造として機能する．

[Liu and Kim(2014)] では Octree の分割基準として，GPU での並列処理と親和性の高い「セル内のジオメトリ密度（三角形数）」を採用している．これによりモデルの三角形群の重心から Morton コードを算出しソートすることで，Octree の階層構造を効率的に並列構築することが可能となった．図 2.6 はこの並列構築プロセスの模式図である．

また [Liu and Kim(2014)] は距離計算時の加速構造である BVH の構築の高速化にも取り組んでいる，具体的には，BVH 構築において上の階層から順にトップダウンで構築する際に生じる並列性のボトルネック（ルートに近い階層では並列性

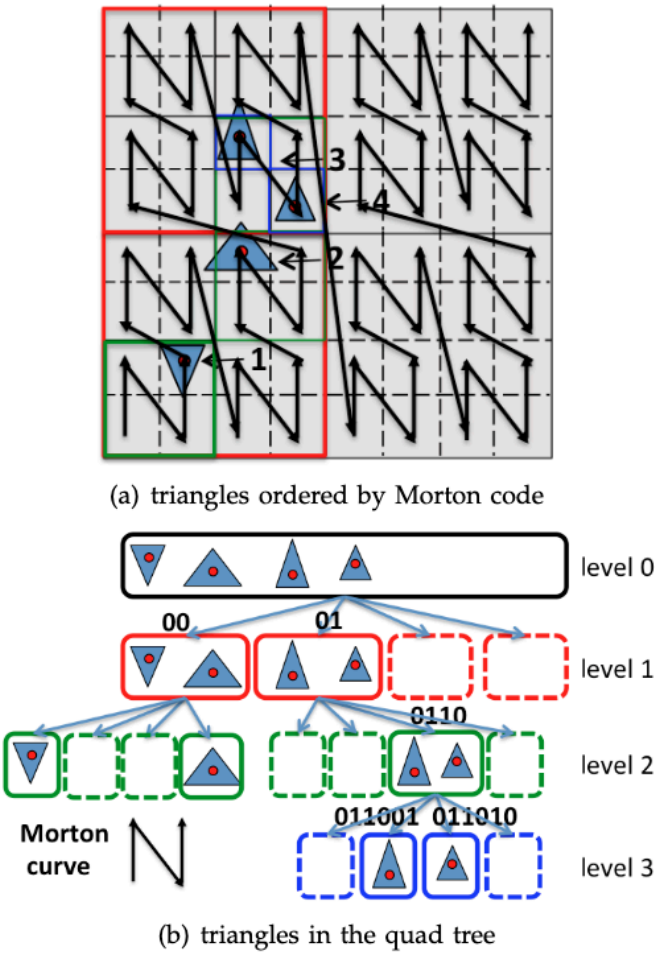


図 2.6: Morton コードに基づく Quadtree 構築の模式図 [Liu and Kim(2014)]

が上がらず GPU スレッドが遊休する問題）を指摘した．この問題を緩和するため，モデルを複数のグループに分割してそれぞれに独立した BVH を並列構築する Multi-BVH というアプローチを導入し GPU スレッドの利用効率を改善した．しかしながら個々の BVH 構築自体はトップダウンアプローチであるため，依然として並列性に関する根本課題は残っている．

2.3.4 GPU 上かつ BVH 中心の ADFs 構築

[Liu and Kim(2014)] の課題であるトップダウン構築における並列性の欠如の解決に取り組んだのが，[Chen et al.(2022)] による BVH 中心のアプローチ（BADF）である．BADF は GPU アーキテクチャの特性を活用した並列構築戦略を提案した．[Liu and Kim(2014)] が採用するトップダウン方式では，Multi-BVH の採用にも関わらずツリーの上位階層（ルートに近い部分）で処理対象のノード数が少なく，多数の GPU スレッドが遊休するというボトルネックが依然として発生してい

た．このトップダウン方式では各 BVH 構築の時間計算量は一般的に $O(n \log n)$ となる．一方で BADF はトップダウンとは異なる構築方式を採用した．三角形群から Morton コードを計算しソートした後，ソート済みの線形リストからツリー構造を構築するフェーズについて，全てのノードを一度に並列処理するアルゴリズムを導入している．このアプローチにより，トップダウン方式が抱える並列性のボトルネックを根本的に回避して時間計算量を $O(n)$ に削減した．

BADF のアルゴリズムパイプラインは以下の複数のステージで構成される．

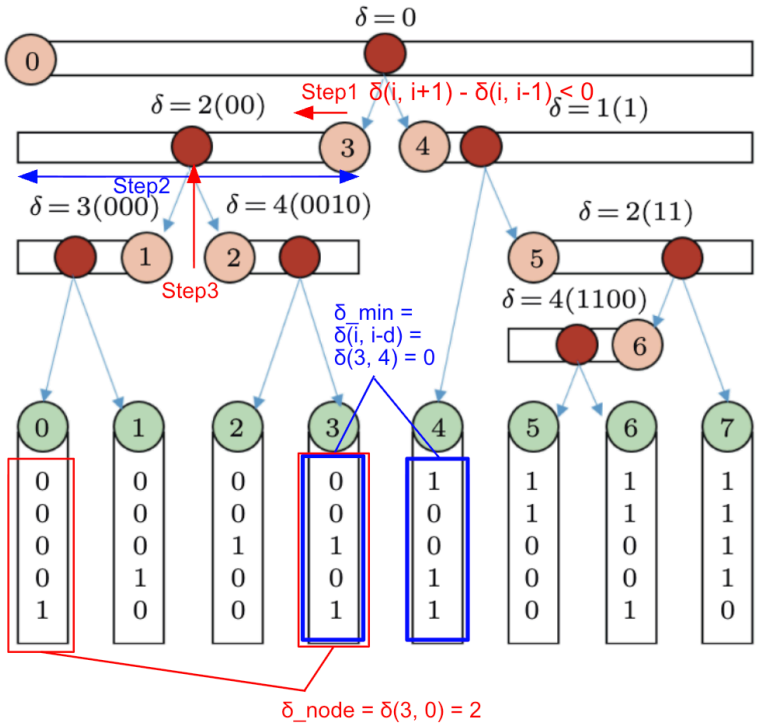


図 2.7: Morton コードに基づく並列 BVH 構築の模式図 [Chen et al.(2022)]

1. **BVH 生成**: まず，モデルを構成する各三角形の重心から Morton コードを計算し，このコードに基づいて三角形群（葉ノード）をソートする．次に [Karras(2012)] が提案した，階層ごとの逐次処理を必要としないデータ並列な構築アルゴリズムを用いる．このアルゴリズムはソート済みの Morton コード配列において，隣接するコード間の最長共通プレフィックス長 δ を比較することで階層構造を決定する．$n-1$ 個の全内部ノードが他のノードの処理結果に依存することなく，次の処理を一度に並列で実行する．次の各 Step は，図 2.7 内のステップ情報に対応する．

   (a) Step1: 内部ノード端点の伸びる方向（正の向きか負の向きか）を特定

   (b) Step2: 内部ノードの伸びる方向および，端点とその隣接ノード端点の δ 値から，内部ノードがカバーする反対の端点をバイナリサーチで特定

(c) Step3: 内部ノードの両端点から，分割点 γ をバイナリサーチで特定

以上のプロセスにより内部ノードは自身のカバー範囲および分割点（すなわち子ノード群）を決定することが出来る．以上の処理は，概念的には「内部ノード端点と同一グループにあるべき葉ノード群」を決定するためのグルーピング操作と言い換えることが出来る．

この後は各ノードの AABB を計算するためのボトムアップ処理が並列実行される（AABB は Axis-Aligned Bounding Box，軸並行境界ボックスを示す．3D モデルのような複雑な形状を内包する，各辺が座標軸に平行な最も単純な直方体であり，高速な交差判定のための単純な代理形状として用いられる構造）．以上のアプローチにより GPU の全コアを効率的に活用し，BVH の構築を高速に行うことが可能となる．

2. **Octree 生成**: 次に，構築した BVH の階層情報を利用して Octree を生成する．具体的には BVH の親子ノード間で Morton コードの共通プレフィックス長を示す δ 値を比較し，その差分から Octree ノードが存在すべき階層を特定する．これにより，BVH と Octree という 2 つの異なるデータ構造の構築が BVH を中心に統一化されたパイプラインで処理される．

3. **BVH リフィッティング**: Morton コードに基づいて生成された BVH のバウンディングボックスは必ずしもタイトではない．距離クエリの効率を向上させるため，BVH の全ノードのバウンディングボックスを，それが内包する三角形群に厳密にフィットするように並列で再計算（リフィット）する．

4. **距離クエリと時空間コヒーレンス**: 最後に ADFs の各セル頂点からの距離を算出する．このクエリ処理では，リフィットされた BVH をトップダウンに探索する．探索が葉ノード（三角形）に到達した後は，クエリ点と三角形の最近傍点との距離を厳密に計算する．この計算は，クエリ点が三角形のボロノイ領域（頂点領域，辺領域，面領域）のいずれに属するかを判定する手法に基づいている．また変形（アニメーション）するオブジェクトに対しては，距離計算の高速化のため時空間コヒーレンスを活用する．前フレームで得られた最近傍三角形を次フレームの探索における初期上限値として利用することで，BVH トラバーサル時の枝刈りを促進し計算量を削減する．

以上のパイプラインにより BADF は [Liu and Kim(2014)] 手法を上回る性能を達成した．しかし本研究の分析により，ADFs 構築全体の処理時間の大半を占める最終ステージの距離クエリ（BVH トラバーサルが主な処理を占める）において依然として性能を律速するボトルネックが存在することが判明した．次章以降では，このボトルネックを解消するための具体的な最適化手法を提案する．

2.4 解決すべき課題

2.4.1 先行研究 BADF における性能上のボトルネック

BADF は従来手法に対して大幅な高速化を達成したが，その処理の内訳を分析すると依然として性能を律速するボトルネックが存在することがわかる．

NVIDIA GeForce RTX 3080 GPU 上で行った BADF の再現実装における性能計測結果を表 2.1 に示す．列「距離クエリの割合」が示す通り，いずれのデータセット（3D ポリゴンモデル）においても距離クエリ処理が全体の 68%以上を占めており，特にポリゴン数が多い Dragon モデルや Buddha モデルでは 85%を超える．この結果は ADFs 構築全体の性能向上において距離クエリ処理の高速化が重要であることを示している．

表 2.1: BADF 再現実装における ADFs 構築処理の内訳

Dataset	三角形数 (k)	全体 (ms)	距離クエリ (ms)	距離クエリの割合
Bunny	70	6.27	4.26	68.0%
Armadillo	213	8.81	6.01	68.2%
Dragon	871	30.82	26.79	86.9%
Buddha	1100	34.69	30.62	88.3%

したがって ADFs 構築プロセスを抜本的に高速化するためには，この距離クエリおよび BVH トラバーサル処理を直接的に改善することが不可欠となる．BADF のボトルネックを分析すると，以下の 2 点の課題が存在する．

1. **メモリアクセスの非効率性（AoS レイアウト）**: BADF の BVH 実装では，ノードデータが Array of Structures (AoS) 形式でメモリに格納されている．AoS 形式では AABB や子ノードインデックスといった異なるデータがメモリ上で混在して配置される．これらのデータ混在により，BVH トラバーサルの過程で特定のデータ（例えば AABB データのみ）にアクセスしたい場合でも，ノード全体のデータを読み込む必要がありメモリ帯域を余分に浪費する．またこのデータ配置はメモリコアレッシングを阻害し，メモリアクセス効率を低下させる要因となっている．

2. **BVH ノードの画一的な探索順序**: BADF における BVH トラバーサルアルゴリズムはスタックを用いた基本的な深さ優先探索であり，子ノードをスタックに積む順序が固定的である．このアプローチの非効率性は，BVH 探索の理論的計算量の観点から明確となる．BVH 探索の計算量は，ツリーが均衡していれば平均 $O(\log N)$ が期待できるが，最悪計算量は $O(N)$ となる．特に本研究の対象である [Karras(2012)] に基づく並列構築手法（LBVH）

は，構築速度を優先し，ノード分割基準として三角形の重心から計算される Morton コード のみを用いる．そのため複雑なポリゴン形状では AABB の重複（Overlap）が大きくなりやすい．探索時にクエリ点がこの重複領域に差し掛かると，画一的な順序では不要な子ノードを先に深くまで辿る可能性がある．この時探索ノード数は $O(\log N)$ から逸脱して $O(N)$ に接近し，全体として非効率な処理となっている．

これらの課題により，BADF は依然として BVH トラバーサル処理に多くの計算時間を費やしている．結果として動的なシーンで求められるような高いフレームレートでの ADFs 更新は達成されておらず改善の余地が大きい．

2.5 まとめ

本章では，距離場構築における研究背景と特に GPU 上での適応的距離場の高速構築に関する課題について述べた．一様グリッドベースの手法は GPU 並列化に適しているが使用メモリ量における効率が悪い．その一方で ADFs は使用メモリ量の点で優れるものの，既存手法では GPU 並列化における最適化余地が大きい．本研究では，現代の GPU アーキテクチャ特性を活かした最適化手法によりこのトレードオフを解決し，実用的な速度での ADFs 構築を目指す．次章ではこれらの課題を解決するための具体的な提案手法について詳述する．

第3章　BVHメモリレイアウトと探索アルゴリズム最適化の提案

3.1　はじめに

前章ではGPU上での適応的距離場（ADF）構築において，BVHのトラバーサル処理が大きなボトルネックとなっていることを指摘した．本章ではこのボトルネックを解消するための具体的な最適化手法を2つ提案する．

第一の手法は，GPUアーキテクチャの特性に着目したBVHメモリレイアウトの最適化である．従来のAoS (Array of Structures) 形式をSoA (Structure of Arrays) に基づくレイアウトへ変更することでメモリアクセスの無駄を削減し，スループットの向上を図る．

第二の手法は探索アルゴリズムそのものの最適化である．クエリ点からの距離情報を活用し，探索順序を動的に決定する「近傍子ノード優先探索」を導入することで不要なノードの枝刈りを促進し，総計算量を削減する．

次節でパイプラインの全体像を述べた後，それ以降の節で2つの提案手法についてその理論的背景と具体的な実装を詳述する．

3.2　提案手法の概要と処理フロー

本節ではADFs構築処理パイプライン全体を概観し，その中で提案手法がどの部分に適用されるのかを明確にする．

図3.1が示すように，本処理は3Dポリゴンメッシュを入力にとり，距離算出処理を高速化するための加速構造としてBVHを構築する．具体的には(a)各三角形の重心からモートンコードを計算・ソートし，空間的な近傍性に基づいてデータを整列させる．このソート済みデータを用いて，(b)GPU上でBVHを並列構築する．本研究の第一の提案手法であるSoAに基づくメモリレイアウトは，このBVH構築結果の格納形式として適用される．次に距離場をサンプリングする空間を定義するため，(c)3Dモデルを内包するOctreeを生成する．

最終段階として，(d)生成されたOctreeの各頂点（クエリポイント）から(b)で構築したBVHを探索（トラバース）し，最近傍の三角形面または辺までの距離を算出する．この最終ステージがADFs構築全体の処理時間の大半を占めるボトルネックであり，本研究での解決対象となる．第二の提案手法である近傍子ノード

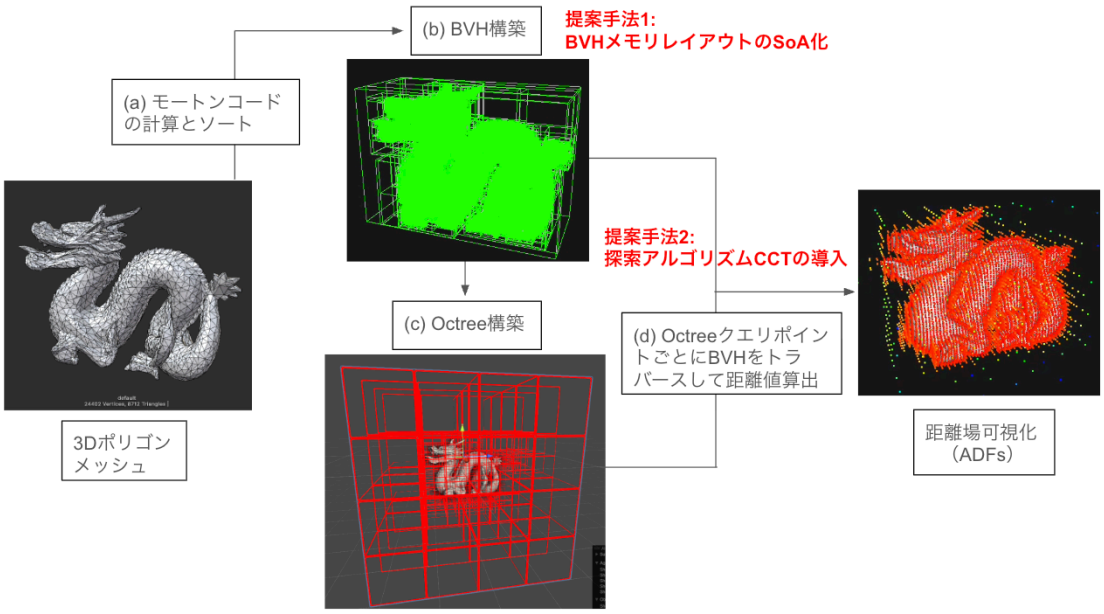


図 3.1: 提案手法を含む ADFs 構築パイプラインの全体像

優先探索（CCT）アルゴリズムはこの BVH トラバーサル処理を効率化するために導入される．

以上の処理を経て得られた距離値の集合が ADFs であり最終的な出力となる．以降の節では，(b) で適用されるメモリレイアウトの最適化と (d) で適用される探索アルゴリズムの最適化についてそれぞれ詳述する．

3.3 BVH メモリレイアウトの SoA 化による最適化

第一の提案手法は，GPU のメモリアクセスパターンに最適化された BVH のデータレイアウトの導入である．この最適化は図 3.1 に示した ADFs 構築パイプラインの (b) ステージで適用される．

データレイアウトには主に AoS (Array of Structures) と SoA (Structure of Arrays) という 2 つの形式が存在する．図 3.2 は，これら 2 つのメモリレイアウトの違いを模式的に示している．

3.3.1 背景：従来の AoS レイアウトにおける問題点

図 3.2 の上段で示す AoS は，複数のデータメンバー（図中の a, b, c, d）を一つの構造体にまとめ，その構造体の配列としてデータをメモリ上に配置する形式である．このレイアウトでは異なる種類のデータがメモリ上で交互に配置（インターリーブ）されることになる．

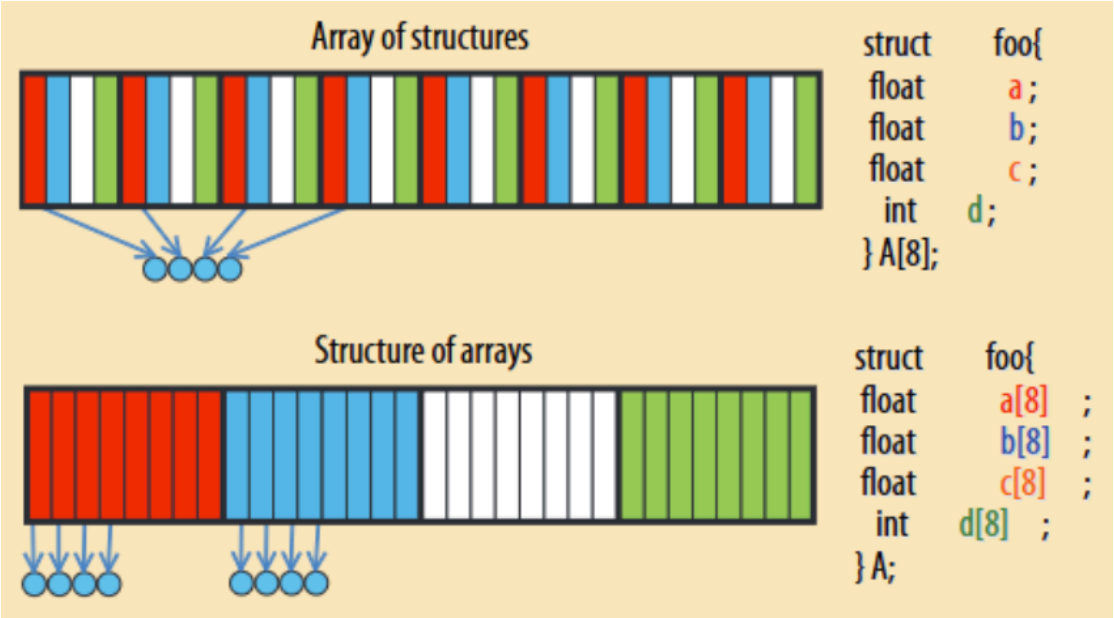


図 3.2: AoS と SoA のメモリレイアウト比較 [Stratton et al.(2012)]

[Chen et al.(2022)] の BVH 実装では，各ノードのデータがこの AoS 形式で格納されていた．具体的なデータ構造例として以下のように表現される．

```
// AoS (Array of Structures) 形式
struct BVHNode {
    AABB bounds;       // 24 バイト
    uint32_t leftChild; // 4 バイト
    uint32_t rightChild;// 4 バイト
};
```

このレイアウトは，BVH トラバーサル処理のように特定のデータメンバーに集中的にアクセスする場合に非効率性を生む．例えば探索の初期段階では，多数のノードに対して AABB (‘bounds‘) のみを用いた距離計算が繰り返し行われる．しかし AoS レイアウトでは，‘bounds‘ (24 バイト) にアクセスする際に隣接する子ノードインデックス (‘leftChild‘, ‘rightChild‘) を含むノード全体 (32 バイト) をメモリから読み出す必要がある．これにより以下の問題が発生する．

- **メモリ帯域の浪費**: アクセスしないデータ（この場合は子ノードインデックス）のためにメモリ帯域が浪費される．
- **メモリコアレッシングの阻害**: 二章で説明したように，GPU ではワープ内のスレッド群が連続したメモリアドレスにアクセスすることでメモリアクセスが一つにまとめられ（コアレッシングされ）スループットが向上する．しかし AoS レイアウトでは各スレッドがアクセスしたい AABB データがメモリ

上で飛び飛びに配置されるため（図 3.2 上段の矢印を参照），コアレッシングが阻害され性能が低下する．

3.3.2 提案：SoA レイアウトに基づくデータ分離

これらの問題を解決するため，本研究では図 3.2 の下段に示す SoA に基づくデータレイアウトへの変更を適用する．SoA は構造体の各メンバーをそれぞれ独立した配列としてメモリ上に連続配置する形式である．これにより同じ種類のデータ（図 3.2 中の a のみ，b のみ等）がメモリ上で一箇所に集まる．

この考え方を BVH ノードに適用し，アクセスパターンが異なるデータを物理的に分離する．具体的には従来の ‘BVHNode‘構造体を，以下のように AABB を格納する配列と子インデックスを格納する配列に分離する．SoA 形式のデータ構造は概念的には以下のように表現できる．

```
// SoA (Structure of Arrays) 形式
struct BVH_SoA {
    // 全ノードの AABB のみを格納する連続配列
    AABB bounds[];

    // 全ノードの子インデックスのみを格納する連続配列
    uint32_t leftChildren[];
    uint32_t rightChildren[];
};
```

この SoA レイアウトにより，AABB のみにアクセスしたい場合は ‘bounds‘配列を，子インデックスにアクセスしたい場合は ‘leftChildren‘/‘rightChildren‘配列を参照すればよく不要なデータを読み込む必要がなくなる．図 3.2 下段の矢印が示すように，ワープ内の各スレッドが同じ種類のデータ（例：AABB）を読む場合，メモリ上で連続したアクセスとなる．このメモリコアレッシングによりメモリ帯域の実行効率が向上し GPU の性能を引き出すことが可能となる．

3.3.3 理論的背景：メモリアクセストランザクションの定式化

AoS から SoA へのレイアウト変更がメモリアクセス効率を改善する理論的な根拠を，GPU のメモリアクセストランザクション数から説明する．GPU がグローバルメモリにアクセスする際の総トランザクション発行回数 N_{trans} は以下のようにモデル化できる．

AoS レイアウトの場合，トランザクション数 $N_{\text{trans,AoS}}$ は次式で表される．

$$N_{\text{trans,AoS}} = p \cdot \left\lceil \frac{N_{\text{warp}} \cdot S_{\text{struct}}}{S_{\text{trans}}} \right\rceil + (1-p) \cdot N_{\text{warp}} \tag{3.1}$$

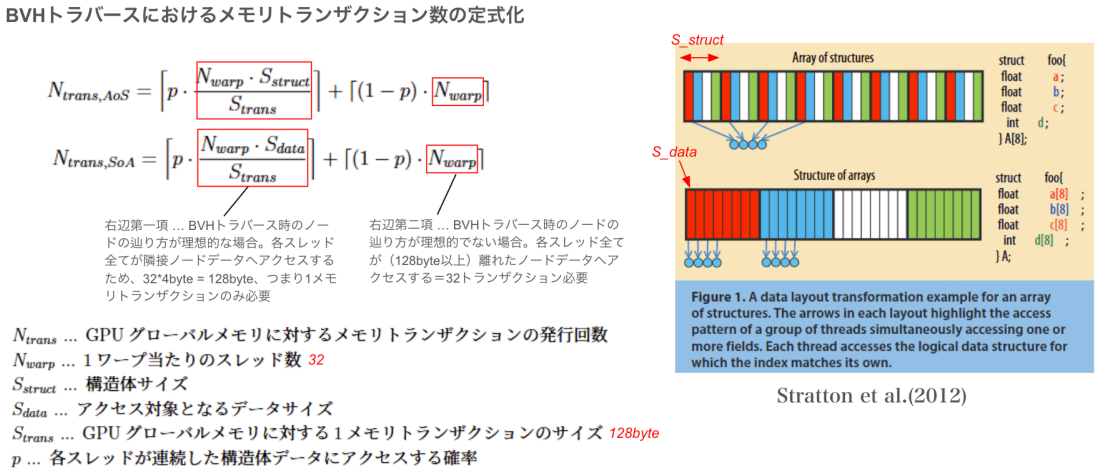


図 3.3: AoS と SoA の発行メモリトランザクション数の定式化

一方 SoA レイアウトの場合，トランザクション数 $N_{\mathrm{trans,SoA}}$ は次式となる．

$$N_{\mathrm{trans,SoA}} = p \cdot \left\lceil \frac{N_{\mathrm{warp}} \cdot S_{\mathrm{data}}}{S_{\mathrm{trans}}} \right\rceil + (1-p) \cdot N_{\mathrm{warp}} \tag{3.2}$$

ここで各変数は以下を意味する．

- N_{trans}: GPU グローバルメモリに対するメモリアクセストランザクションの発行回数
- N_{warp}: 1 ワープ当たりのスレッド数（NVIDIA GPU では通常 32）
- S_{struct}: 構造体全体のサイズ
- S_{data}: アクセス対象となるデータ要素のサイズ
- S_{trans}: GPU グローバルメモリに対する 1 メモリアクセストランザクションのサイズ
- p: 各スレッドが連続した構造体にアクセスする（コアレッシングが有効となる）確率

式 (1) と (2) の比較から分かるように，また定式と各パラメータの関係が図 3.3 から直感的に分かるように，コアレッシングが有効なアクセス（確率 p）において AoS ではトランザクション数が構造体全体のサイズ S_{struct} に依存するのに対し，SoA では実際に必要なデータのサイズ S_{data} にのみ依存する．BVH トラバーサルのように $S_{\mathrm{data}} < S_{\mathrm{struct}}$ となるアクセスが支配的な場合，SoA レイアウトは AoS レイアウトと比較してメモリアクセストランザクション数を削減し，性能向上に寄与することが理論的に示される．

3.3.4 BVH メモリレイアウト SoA 化の実際の実装方式

上記の理論的考察に基づき，BVH ノードのデータレイアウトを AoS から SoA へ変換することを提案する．具体的にはアクセスパターンが異なるデータを物理的に分離し，それぞれを連続した配列として格納する．

従来の BVHNode 構造体を以下の 2 つの配列に分離する．

- **AABB 配列**: 全てのノードの AABB (‘bounds‘) のみを格納する連続配列．
- **子インデックス配列**: 全てのノードの子インデックスペア (‘leftChild‘, ‘rightChild‘) をパックして格納する連続配列．

このアプローチは BVH ノードの属性レベルでは SoA に分離する一方で，AABB 内部のベクトル成分 (‘bMin‘, ‘bMax‘) や子インデックスペアなど常に同時にアクセスされる要素群は AoS のまま維持する．このため本実装は SoA と AoS の利点を組み合わせたハイブリッド SoA と呼べるアプローチである．これにより AABB の再構成に複数回のメモリアクセスが必要となる完全な SoA 化のオーバーヘッドを避けつつ，アクセスパターンの違いに起因する非効率性を解消できる．

3.3.5 期待される効果

SoA に基づくメモリレイアウトの導入により，以下の効果が期待される．

- **メモリ帯域効率の向上**: 距離計算時は AABB 配列から 24 バイト，子ノード探索時は子インデックス配列から 8 バイトを読み込むだけで済み，メモリアクセスの無駄を削減可能となる．
- **高いコアレッシング性**: AABB データがメモリ上で完全に連続するため，GPU ワープ内のスレッド群によるアクセスがコアレッシングされメモリスループットが向上する．
- **キャッシュ効率の向上**: AABB を中心とした処理ではキャッシュラインが AABB データのみで満たされるため，空間的・時間的局所性が高まりキャッシュヒット率の向上が見込まれる．

3.4 近傍子ノード優先による探索アルゴリズムの最適化

第二の提案手法は，BVH トラバーサル処理そのもののアルゴリズムを改善し計算量を削減することである．この最適化は図 3.1 に示した ADFs 構築パイプラインの (d) のステージに適用される．本節ではまず最適化の比較対象となるベースラインの探索アルゴリズムについて述べ，次にその問題点を解決するための提案手法を説明する．

3.4.1　ベースライン：標準的なBVHトラバーサル

ADFs構築における距離クエリ処理，すなわちBVHトラバーサル処理はスタックを用いた深さ優先探索を基本とする．ベースラインである[Chen et al.(2022)]で採用されているアルゴリズムは以下の手順に基づいている．

1. 各クエリ点は探索のためのローカルなスタックを持つ．
2. BVHのルートノードをスタックに積み探索を開始する．
3. スタックが空になるまで以下を繰り返す：
   (a) スタックからノードを一つ取り出す（pop）．
   (b) そのノードのAABBとクエリ点との距離を計算し，現在見つかっている最短距離（minDist）よりも遠ければそのノード以降の探索を打ち切る（枝刈り）．
   (c) 枝刈りされなかった場合，そのノードが内部ノードであれば子ノードをスタックに積む（push）．ノードが葉ノード（ポリゴン）であればポリゴンとの正確な距離を計算し最短距離を更新する．この正確な距離計算は，クエリ点が三角形のボロノイ領域（頂点領域，辺領域，面領域）のいずれに属するかを特定し，それぞれに対応する最近傍点（頂点，辺への射影点，面への射影点）との距離を算出するアルゴリズムに基づいている．

このアルゴリズムには，ステップ3-(c)の処理において最適化の余地が存在する．第2.4節で述べた通り，ポリゴン形状が複雑な場合，多くのBVHノードのAABBが空間的に重複し，探索計算量が$O(N)$に接近する状況が起こり得る．このようなBVHに対しベースラインの画一的な探索順序（常にまず左の子，次に右の子）を適用すると，クエリ点から遠い子ノードを先に探索してしまう場合がある．BVH探索の効率は最近傍距離（minDist）の更新速度に大きく依存するため，遠いノードから探索を始めると枝刈りが遅れ，不要な計算コストが増大する．

3.4.2　提案：距離ベースの動的探索順序決定

この課題に対して，クエリ点からの距離に基づいて探索順序を動的に決定する「近傍子ノード優先探索（Closer Child Traversal, CCT）」を提案する．このアルゴリズムではBest-First Search（最良優先探索）の原理に基づき，常に最も有望なノード（クエリ点に最も近いノード）から探索を進める．

Algorithm 1に，本提案手法を組み込んだ距離クエリ処理全体の擬似コードを示す．以下ではこのアルゴリズムの流れに沿って具体的な実装を説明する．

まず各クエリ点に対して，前フレームの計算結果を初期値として利用する時間的コヒーレンスの適用を試みる（Algorithm 1: 7-14 行目）．これにより，静的または動きの少ないシーンにおいて探索開始時点から精度の高い最短距離（`minDistSq`）を設定し，枝刈りの効率を高める．

探索は BVH のルートノードをローカルスタック `S` に積み（push），スタックが空になるまでループを繰り返すことで実行される（16-20 行目）．

ループ内部ではまずスタックからノードを取り出し（pop），そのノードの AABB とクエリ点との距離を計算する（22-24 行目）．この時，前節で提案した SoA レイアウトの利点を活かして AABB データのみ読み込む．そして計算された距離が現在見つかっている最短距離 `minDistSq` よりも大きい場合，そのノードが内包する子孫ノード群は最近傍となり得ないためこれ以降の探索を打ち切る（枝刈り）．

枝刈りを通過した場合，次に子ノードのインデックス情報を読み込み（27-28 行目），葉ノードであれば三角形との厳密な距離計算を行い最短距離を更新する（31-36 行目）．

子ノードが内部ノードであった場合，ベースライン手法とは異なり，まず左右両方の子ノードの AABB とクエリ点との距離をそれぞれ計算する（46-52 行目）．次に計算した 2 つの距離を比較し，スタックに積む順序を動的に決定する（55-63 行目）．スタックは LIFO (Last-In, First-Out) のデータ構造であるため，後から積んだ要素が次に取り出される．この特性を利用してクエリ点から遠い方の子ノードを先にスタックに積み次に近い方の子ノードを積む．この手続きによりループの次の反復でスタックから取り出されるノードは必ずよりクエリ点に近い方の子ノードとなる．

Algorithm 1: Proposed Distance Query with SoA and CCT

procedure DistanceQuery($QueryPoints, BVH, Triangles, PrevResults$)

// クエリポイントごとに並列処理
for all each query point p_i in $QueryPoints$ **do**
    $minDistSq \leftarrow \infty$
    $nearestTriId \leftarrow$ INVALID
    // 時間的コヒーレンスの適用
    if $PrevResults$ is available **then**
        $prevTriId \leftarrow PrevResults[i]$
        if $prevTriId \neq$ INVALID **then**
            $T \leftarrow Triangles[prevTriId]$
            $minDistSq \leftarrow$ PointTriangleDistSq(p_i, T)
            $nearestTriId \leftarrow prevTriId$
        end if
    end if
    Create local stack S
    if $BVH.root \neq$ INVALID **then**
        $S.push(S, BVH.root)$
    end if
    while S is not empty **do**
        $nodeId \leftarrow S.pop()$
        // AABB データのみ読み込み (SoA)
        $aabb \leftarrow BVH.Bounds[nodeId]$
        if PointAABBDistSq($p_i, aabb$) $\geq minDistSq$ **then**
            // 現在の minDist より遠い場合枝刈り
            continue
        end if
        // 枝刈りを通過した場合のみ子ノードインデックス読み込み (SoA)
        $childPair \leftarrow BVH.ChildPairs[nodeId]$
        $leftChild, rightChild \leftarrow$ UnpackChildren($childPair$)
        // 子ノードが葉である場合，三角形との距離を計算し最短距離を更新
        if IsLeaf($leftChild$, $Triangles$.count) **then**
            UpdateMinDistance($p_i, leftChild, Triangles, minDistSq, nearestTriId$)
        end if
        if IsLeaf($rightChild$, $Triangles$.count) **then**
            UpdateMinDistance($p_i, rightChild, Triangles, minDistSq, nearestTriId$)
        end if

```
36:         distSqLeft ← ∞, distSqRight ← ∞
37:         // 子ノードが内部ノードである場合，AABB までの距離を計算
38:         if not IsLeaf(leftChild, Triangles.count) then
39:             distSqLeft ← PointAABBDistSq(p_i, BVH.Bounds[leftChild])
40:         end if
41:         if not IsLeaf(rightChild, Triangles.count) then
42:             distSqRight ← PointAABBDistSq(p_i, BVH.Bounds[rightChild])
43:         end if
44:         // Closer Child Traversal (CCT) の適用
45:         if distSqLeft < distSqRight then
46:             // 遠い方の子ノードを先にプッシュし，続けて近い方をプッシュする
47:             if distSqRight < minDistSq then
48:                 S.push(S, rightChild)
49:             end if
50:             if distSqLeft < minDistSq then
51:                 S.push(S, leftChild)
52:             end if
53:         else
54:             // 遠い方の子ノードを先にプッシュし，続けて近い方をプッシュする
55:             if distSqLeft < minDistSq then
56:                 S.push(S, leftChild)
57:             end if
58:             if distSqRight < minDistSq then
59:                 S.push(S, rightChild)
60:             end if
61:         end if
62:     end while
63:     Record minDistSq and nearestTriId for query point p_i
64: end for
```

なお，本 CCT は探索の順序を最適化するものであり，枝刈りの条件（Algorithm 1: 24 行目）そのものには変更を加えないため，数学的な正当性は維持される．BVH で用いられる AABB は，内包する全てのジオメトリ（子ノードや三角形）を保守的（Conservative）に包み込む．この定義により,「AABB までの距離」は「その AABB 内部にある，いかなる三角形までの距離」よりも必ず等しいか小さい（$\leq$）ことが保証されている．したがって「上位階層の AABB は遠いが，その内部に存在する三角形が最近傍である」という状況は発生し得ない．CCT ではこの AABB の性質を前提に，クエリ点に近いノードから優先的に探索することで早期に `minDistSq` を更新し，後続ノードの枝刈り効率を高めることを目的としている．

3.4.3　期待される効果

この動的な探索順序の導入により以下の効果が期待される．

- **探索ノード数の削減**: 近いノードから探索することで最近傍距離がより早期に，より小さな値に更新される．これにより更新された距離を用いてより多くの不要なサブツリーを効率的に枝刈りでき，探索すべきノード総数が削減される．
- **キャッシュ効率の向上**: 物理的に近い領域を連続してアクセスする可能性が高まるため空間的局所性が向上し，キャッシュ効率の改善にも寄与する．

3.5　まとめ

本章では GPU 上での BVH トラバーサルを高速化するため，2 つの相補的な手法を提案した．第一にデータアクセスパターンに基づいて BVH ノードのメモリレイアウトを SoA (Structure of Arrays) に基づいて最適化する手法を提案した．これによりメモリアクセスの無駄を排除し，メモリコアレッシングとキャッシュ効率の向上を図る．第二にクエリ点からの距離に応じて探索順序を動的に決定する「近傍子ノード優先探索」を提案した．これにより探索の早い段階で最近傍距離を更新し，不要なノードの枝刈りを促進することで総計算量を削減する．
次章ではこれら 2 つの提案手法を実装し，その有効性をベースライン手法と比較する実験・評価を行う．

第4章　実験・評価

4.1　はじめに

前章ではGPU上でのBVHトラバーサルを高速化するため，(1)SoAに基づくメモリレイアウト最適化，および(2)近傍子ノードを優先する探索アルゴリズムという2つの手法を提案した．
本章では，これらの提案手法の有効性を特性の異なる複数の3Dモデルを用いて定量的に評価することを目的とする．提案手法を段階的に適用した際の性能についてGPUパフォーマンスメトリクスを用いて多角的に比較・分析し，その効果と特性，また最適化に伴うトレードオフについて考察する．

4.2　実験条件

4.2.1　比較対象と評価モデル

提案手法の効果を段階的に検証するため，以下の3つの実装バージョンを比較対象とする．

Baseline [Chen et al.(2022)]の再現実装．従来のAoSレイアウトを用いて探索順序の最適化も行わない実装．最適化適用前のベースライン．

SoA 第3.2節で提案したSoAに基づくメモリレイアウト最適化のみを適用した実装．

SoA + CCT SoAレイアウトに加え，第3.3節で提案した近傍子ノード優先探索（Closer Child Traversal）アルゴリズムの両方を適用した最終的な実装．

評価にはポリゴン数や形状特性が異なる以下の4つの標準的な3Dモデルを使用した．

- Stanford Bunny (7.0万ポリゴン)
- Stanford Armadillo (21.3万ポリゴン)
- Stanford Dragon (87.1万ポリゴン)
- Stanford Buddha (110万ポリゴン)

4.2.2 計測環境と評価指標

計測環境

- **GPU**: NVIDIA GeForce RTX 3080
- **CPU**: Intel Core i9
- **OS**: Windows 11
- **計測ツール**: NVIDIA Nsight Compute

評価指標

提案手法がメモリアクセス効率と実行時間に与える影響を評価するため，以下の主要なメトリクスを計測・分析した．

- `gpu__time_duration.avg`: カーネルの 1 起動あたりの実行にかかった平均時間（Nsight Compute がカーネル 1 回の起動を内部的に複数回リプレイして算出）．この値が小さいほど総合的な処理性能が高いことを示す．
- `dram__sectors_read.sum`: メインメモリ（DRAM）から読み出された 32 バイトセクタの総数．L1/L2 キャッシュでヒットしなかったメモリアクセスの量を示し，メモリ帯域幅への負荷を評価する指標となる．
- `l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum`: L1 キャッシュに対して発行されたグローバルメモリアクセスリクエストの総数．この値が小さいほど，メモリアクセスの発行回数そのものが少ないことを意味する．
- `l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum`: L1 キャッシュから読み出された 32 バイトセクタの総数．L1 キャッシュレベルでの総データ転送量を示す．

計測コマンド

これらのメトリクスは NVIDIA Nsight Compute を用いて収集した．計測に用いたコマンドラインは以下である．

```
ncu -f -o result --print-summary per-kernel \
-k regex:"distanceQueryKernel|distanceQueryKernelHybrid" \
-s 5 -c 180 --page details --csv \
--metrics \
gpu__time_duration.avg,\
```

```
l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum,\
l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum,\
dram__sectors_read.sum \
./App.exe [model_path] > [output.csv]
```

このコマンドは以下の設定に基づきプロファイリングを実行する.

- `-k regex:"..."`: 計測対象を正規表現で指定し，ベースラインの `distanceQueryKernel` および提案手法を適用した `distanceQueryKernelHybrid` に限定する.
- `-s 5 -c 180`: プログラム実行時の初期化処理などによる影響を排除するため，最初の 5 フレームをスキップしその後の任意のフレーム数（例では 180 フレーム）を計測対象とする.
- `--csv`: 結果を CSV 形式で出力する.
- `--metrics ...`: 評価指標で述べた特定のパフォーマンスメトリクスのみを収集対象とする.

出力結果

上記コマンドの実行結果として，以下項目を持つ CSV データを得ることが出来る.

```
"Process ID","Process Name","Host Name","Kernel Name","Block Size",
"Grid Size","Device","CC","Invocations","Section Name","Metric Name",
"Metric Unit","Minimum","Maximum","Average"
```

特に見るべき項目は以下である.

- `Metric Name`: 先述した評価指標を示す（”dram__sectors_read.sum” や”gpu__time_duration.avg”など）.
- `Metric Unit`: 各評価指標に付随する単位を示す（32 バイトセクタの数を示す”sector”，時間を示す”ns”など）.
- `Invocations`: 対象カーネルの連続呼び出し回数を示す.
- `Minimum`，`Maximum`，`Average`: 上記 Metric Name（カーネル 1 起動あたり）の集計値について，Invocations 回数分のカーネル起動における最小・最大・平均値を示す.

4.3 実験結果と評価

本節では前節で示した実験結果に基づき，提案手法の有効性について複数の観点から考察を行う．まず計測されたパフォーマンスメトリクスから，第 3 章で述べた各提案手法の「期待される効果」がどの程度達成されたかを検証する．次にベースライン手法との比較を通じて本研究が達成した性能向上の要因を分析し，その意義を明らかにする．

4.3.1 期待される効果と実測値の対応の評価

本研究では，(1)SoA に基づくメモリレイアウトと (2) 近傍子ノード優先探索（CCT）という 2 つの手法を提案した．本項では，それぞれの導入時に期待された効果が実験結果によって裏付けられていることを示す．

SoA メモリレイアウトの有効性検証

第 3.3.5 節では，SoA レイアウト導入による効果として「メモリ帯域効率の向上」「高いコアレッシング性」「キャッシュ効率の向上」を期待したが，実験結果はこの期待を裏付けている．表 4.1 から表 4.4 において，`Baseline` 列と `SoA` 列の数値を比較すると，SoA への移行により全てのモデルで L1 キャッシュへのリクエスト総数 (`L1 Load Requests`) が 21%以上削減され，実行時間も約 1.4 倍高速化したことが分かる．これは AoS レイアウトで発生していた不要なデータ（AABB アクセス時の子インデックス等）の読み込みが削減され，メモリ帯域が効率的に利用されたことを示唆している．

表 4.1: Dragon モデル (87.1 万ポリゴン) の詳細メトリクス

Metric	Baseline	SoA	SoA + CCT
実行時間 (ms)	29.92	20.40	**5.07**
DRAM Read Sectors	83.0M	61.9M	**19.7M**
L1 Load Requests	240.0M	190.2M	**16.8M**
L1 Load Sectors	3,390M	2,290M	**105.3M**
Avg. Sectors/Req	14.13	12.04	**6.25**

またメモリアクセスの質を示す `Avg. Sectors / Request` (リクエスト 1 回あたりの転送セクタ数) に着目すると，例えば Dragon モデルでは 14.13 から 12.04 へと値が減少している．これは SoA 化によってデータがメモリ上で連続的に配置され，一度のメモリアクセストランザクションでより多くの有効なデータを取得できるようになったこと，コアレッシングが改善されたことを示している．

表 4.2: Bunny モデル (7.0 万ポリゴン) の詳細メトリクス

Metric	Baseline	SoA	SoA + CCT
実行時間 (ms)	5.43	3.79	**1.22**
DRAM Read Sectors	0.44M	**0.21M**	1.05M
L1 Load Requests	50.5M	39.5M	**4.0M**
L1 Load Sectors	547.6M	350.4M	**26.7M**
Avg. Sectors/Req	10.84	8.87	**6.65**

表 4.3: Armadillo モデル (21.3 万ポリゴン) の詳細メトリクス

Metric	Baseline	SoA	SoA + CCT
実行時間 (ms)	10.57	7.47	**1.95**
DRAM Read Sectors	6.67M	**2.86M**	4.63M
L1 Load Requests	78.2M	60.9M	**6.4M**
L1 Load Sectors	1,074M	707.7M	**41.3M**
Avg. Sectors/Req	13.73	11.61	**6.45**

近傍子ノード優先探索（CCT）の有効性検証

第 3.4.3 節では CCT アルゴリズム導入による効果として，特に「探索ノード数の削減」を期待したが，この効果も実験結果において確認することができる．表 4.1 から表 4.4 において `SoA` 列と `SoA + CCT` 列の数値を比較すると，CCT を適用したことで `L1 Load Requests`（メモリアクセスリクエスト数）が削減されたことが分かる．例えば Dragon モデルでは 1 億 9020 万回から 1680 万回へと約 93%削減している．これは CCT アルゴリズムが期待通りに不要な枝刈りを促進し探索すべきノード総数を削減したことを示している．
一方で Bunny モデル（表 4.2）や Armadillo モデル（表 4.3）では，CCT の適用により `DRAM Read Sectors` が SoA 単体時よりも増加するトレードオフが観測された．これはアクセスパターンが変化したことでキャッシュヒット率が局所的に低下した可能性を示唆する．しかしそれを補うほどアクセス総数が削減されたため最終的な実行時間は短縮された．最適化において本質的な処理量（ここでは探索ノード数）を削減することの重要性がこの結果から示されている．

4.3.2 ベースライン手法との比較評価

本研究のベースラインは [Chen et al.(2022)] によって提案された BVH 中心の ADFs 構築手法（BADF）である．なお [Chen et al.(2022)] はその論文内で「2.3.3 GPU

表 4.4: Buddha モデル (110 万ポリゴン) の詳細メオリクス

Metric	Baseline	SoA	SoA + CCT
実行時間 (ms)	32.16	23.55	**5.29**
DRAM Read Sectors	126.3M	97.6M	**21.5M**
L1 Load Requests	245.3M	202.6M	**17.5M**
L1 Load Sectors	3,158M	2,242M	**113.7M**
Avg. Sectors/Req	12.87	11.07	**6.48**

上での ADFs 構築」で紹介した [Liu and Kim(2014)] の手法に対して実行速度面での優位性を示しているため，本稿での比較対象は BADF に絞る．
表 4.5 は各 3D モデルにおける距離クエリカーネルの高速化倍率をまとめたものである．なおこの数値は表 4.1 から表 4.4 に示された各実装の実行時間とベースラインの実行時間の比率である．

表 4.5: 距離クエリカーネルの高速化倍率（対ベースライン）

3D モデル	SoA 化による高速化	SoA + CCT による高速化
Dragon	1.47x	5.90x
Bunny	1.43x	4.46x
Armadillo	1.41x	5.42x
Buddha	1.37x	6.08x

この性能向上の要因は，前項で分析した 2 つのアプローチが相補的に機能した結果であると結論付けられる．まず SoA レイアウトがメモリアクセスの質を改善し GPU のメモリシステムが効率的に動作するための土台を構築したことで，約 1.4 倍の安定した性能向上が得られた．その上で CCT アルゴリズムがアクセスの量そのものを削減したことで探索すべきデータ量が削減され，4.4 倍から 6.1 倍という高速化に繋がった．以上から，データレイアウトの最適化とアルゴリズムによる処理量削減を行う二段階のアプローチが，対象データの特性によらず高い実行性能を達成する戦略となったことが示されている．

4.4 考察

4.4.1 手法全体でのメモリアクセス量削減の達成について

前節で示した最大 6.1 倍という実行時間の大幅な短縮は，性能を律速するグローバルメモリへのアクセス効率が劇的に改善された結果である．実験データは提案手

法が単に計算を高速化するだけでなく，メモリアクセスの総量そのものを削減したことを示している．
代表例である Dragon モデルにおいて，最終的な実装である SoA + CCT はベースラインと比較して L1 キャッシュへのグローバルメモリアクセスリクエスト数 (`l1tex__t_requests`) を約 93%，キャッシュミスにより実際に DRAM から読み出されたデータ総量 (`dram__sectors_read`) を約 76%削減した．なおこれらの削減率は，ベースラインの値を基準として以下の式に基づいて算出されている．
L1 リクエスト数の削減率の算出：

$$\begin{aligned} \text{削減率 (\%)} &= \left(1 - \frac{\text{最適化後の値}}{\text{ベースラインの値}}\right) \times 100 \\ &= \left(1 - \frac{16,838,180}{239,999,420}\right) \times 100 \approx 93.0\% \end{aligned} \tag{4.1}$$

DRAM リード量の削減率の算出：

$$\text{削減率 (\%)} = \left(1 - \frac{19,680,971}{83,008,845}\right) \times 100 \approx 76.3\% \tag{4.2}$$

この結果は，性能向上の主要因がアルゴリズムの改善によって探索対象を削減し，それに伴いメモリアクセス数とデータ転送量を削減できたことにある旨を示している．以降の項ではこのメモリアクセス効率の改善について，提案の二手法それぞれについて詳細な分析を行う．

4.4.2 理論モデルと実測値の比較による妥当性検証

本節では第 3 章で導入した理論モデルが本章で観測された実測値の傾向をどの程度説明できるかを検証し，提案手法の有効性の理論的根拠を補強する．分析の目的は性能予測モデルを構築することではなく，観測された性能向上の要因を (1)SoA レイアウトによるメモリアクセス効率の改善と，(2)CCT アルゴリズムによるアクセス総量の削減という 2 つの異なるメカニズムに切り分け，それぞれを定量的に評価することである．

分析 1: SoA レイアウトの理論的有効性の検証

まず CCT を適用する前の純粋なデータレイアウト変更の効果（Baseline から SoA への移行）に着目する．第 3 章で提示したメモリアクセストランザクション数の理論式を再掲する．

$$N_{\text{trans,AoS}} = p \cdot \left\lceil \frac{N_{\text{warp}} \cdot S_{\text{struct}}}{S_{\text{trans}}} \right\rceil + (1 - p) \cdot N_{\text{warp}} \tag{4.3}$$

$$N_{\text{trans,SoA}} = p \cdot \left\lceil \frac{N_{\text{warp}} \cdot S_{\text{data}}}{S_{\text{trans}}} \right\rceil + (1-p) \cdot N_{\text{warp}} \tag{4.4}$$

ここで，各パラメータを $N_{\text{warp}} = 32$, $S_{\text{trans}} = 128$ Byte, $S_{\text{struct}} = 32$ Byte と定義する．BVH トラバーサルで支配的な AABB アクセスを考慮すると，$S_{\text{data}} = 24$ Byte となる．このときコアレッシングアクセス時（確率 p で発生）のトランザクション量を示す天井関数内の項は，AoS と SoA で以下のように計算される．

- AoS の場合: $\lceil (32 \cdot 32 \text{ Byte})/128 \text{ Byte} \rceil = \lceil 8 \rceil = 8$
- SoA の場合 (AABB アクセス): $\lceil (32 \cdot 24 \text{ Byte})/128 \text{ Byte} \rceil = \lceil 6 \rceil = 6$

この計算より SoA レイアウトは AoS レイアウトと比較して，1 回あたりのコアレッシングアクセスにおけるトランザクション量を理論上 25%削減する効果があることがわかる．
この理論的な予測は，Dragon モデルにおける実測値と高い整合性を示している．表 4.2 によると，Baseline から SoA への移行で `l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum`（L1 キャッシュへのアクセス要求回数）は約 21%削減（2.40 億回→ 1.90 億回）された．これは理論予測の 25%削減と近い値であり，SoA 化による性能向上は理論モデルによって予測される「1 回あたりのメモリアクセス効率の改善」によって定量的にもたらされたものであると示唆される．

分析 2: CCT アルゴリズムによるアクセス総量削減の定量的評価

次に最終的な実装である SoA + CCT が達成した性能向上を分析する．Dragon モデルにおいて，L1 キャッシュへのアクセス要求回数はベースラインの 2.40 億回から 1680 万回へ約 93%の削減を達成した．
この 93%という削減率のうち，分析 1 で示した SoA 化による効率改善の貢献は約 21%である．したがって残りの削減効果は SoA 化だけでは説明できず，性能向上への最大の貢献は CCT アルゴリズムが不要な枝刈りを促進し，GPU がアクセスすべき BVH ノードの総数を削減したものによると推測できる．CCT は理論モデルが扱う「1 回あたりのアクセス効率」を改善するのではなく，モデルが前提とする「アクセス回数の総和」そのものを減らしたものと考えられる．

総括

以上の分析から，本研究で観測された性能向上は，提案した 2 つの手法がそれぞれ異なるメカニズムで寄与した結果であることが定量的に実証された．すなわち (1)SoA レイアウトはアクセスの質的な改善（1 回あたりのトランザクション量を削減）を行い，(2)CCT アルゴリズムはアクセスの量的な改善（アクセス総数の削

減）を行った．この 2 つのアプローチが相補的に機能し最終的な高速化に繋がったことが，理論モデルと実測値の比較分析から裏付けられた．

4.4.3 メモリアクセス効率の詳細分析

L1 キャッシュにおけるメモリアクセスの質をより深く分析するため，リクエスト 1 回あたりに転送されたセクタ数（‘Avg. Sectors/Req‘）を考察する．この値は，一度のリクエストでどれだけのデータがまとめて転送されたかを示す指標である．GPU アーキテクチャにおいて，32 スレッド（1 ワープ）がそれぞれ 4 バイトのデータに完全にコアレッシングされたアクセスを行う場合，1 リクエストあたり 128 バイト（= 4 セクタ）の転送が発生するため，この値が 4 に近いほど効率的なメモリアクセスが行われていることを示唆する．
Baseline ではこの値は 10.84 から 14.13 の範囲にあり，理想値である 4 を大きく上回っている．これは AoS レイアウトにおいて，AABB（24 バイト）のような構造体を読み込む際に 1 スレッドあたりが 4 バイトを超えるデータを要求するため，ワープ全体で 128 バイト（4 セクタ）を超える非効率なトランザクションが頻繁に発行されていることを示している．
次に SoA 導入時では，この値は 8.87 から 12.04 の範囲へと低下することが確認できる．これは SoA 化によってデータアクセスが分離され，AABB のみにアクセスする際のトランザクションがより効率化された結果であると考えられる．このことから，SoA レイアウトの導入は L1 リクエスト数や転送セクタ総数の削減（量的な改善）だけでなく，メモリアクセス一回あたりの効率（質的な改善）にも寄与していることがわかる．
そして SoA + CCT 導入時では値は 6.25 から 6.65 の範囲へと改善している．これは，CCT アルゴリズムによって探索において近いノードと遠いノードの混在が減り，より局所性の高いアクセスパターンが増加した結果としてメモリアクセスの全体的な効率が向上したことを示している．
ただし SoA + CCT においても値は依然として 4 を上回っており，完全なコアレッシングには至っておらず，データ構造のさらなる最適化の可能性が残されていることを示唆している．

4.4.4 最適化におけるトレードオフの分析

本実験では，最適化の過程における副作用も確認された．表 4.2 (Bunny) および表 4.3 (Armadillo) に示す通り，SoA + CCT 導入時は SoA 単体導入時よりも DRAM 読み込み量が増加している．
この現象は，CCT アルゴリズムによるアクセスパターンの変化が GPU のキャッシュ戦略と必ずしも一致せず，結果としてキャッシュヒット率が低下した可能性を示唆している．つまり，ある最適化が全ての性能指標を一様に向上させるわけで

はなく，メモリアクセスの局所性とメモリ階層の間には複雑なトレードオフが存在することがわかる．
しかし，この DRAM アクセス増という負の影響があったにもかかわらず最終的な実行時間は短縮されている．このことは，アルゴリズム改善による L1 リクエスト数の大幅な削減効果がキャッシュヒット率低下による性能への影響を上回り，全体として実行時間短縮に寄与した結果であると分析できる．

4.4.5 ADFs 構築全体への影響

ここまで議論してきた性能改善は，表 2.1 で示したように ADFs 構築プロセスの中で最も時間を要する距離クエリカーネルに焦点を当てたものである．本節ではこのカーネルの高速化が ADFs 構築プロセス全体に与える影響を評価する．表 4.6 に，提案手法（SoA + CCT）を適用した際の ADFs 構築全体の性能測定結果を示す．

表 4.6: 提案手法適用時の ADFs 構築全体の性能

3D モデル	全体 (ms)	距離クエリ (ms)	対 Baseline 高速化率
Bunny	3.10	1.12	2.02x
Armadillo	3.62	1.90	2.43x
Dragon	8.30	4.87	3.71x
Buddha	8.85	5.04	3.92x

表 4.6 に示す通り，提案手法の適用により ADFs 構築プロセス全体の処理時間はベースラインと比較して 2.02 倍から 3.92 倍高速化された．これはボトルネックであった距離クエリ（表 2.1 では全体の 68-88%を占有）が最大 6.1 倍高速化された結果，その効果が全体プロセスにも波及したことを示している．
一方で全体の高速化率は距離クエリ単体の高速化率よりも緩やかであり，これは距離クエリ以外の処理（例：Octree 構築，BVH 構築，動的なクエリポイントの決定など）が新たな律速段階となっていることを示唆している．ADFs 構築全体のさらなる高速化にはこれらの処理の最適化が今後の課題となる．

4.5 まとめ

本章では提案した 2 つの最適化手法の有効性について，特性の異なる 4 つの 3D モデルを用いて実験的に評価した．その結果，SoA に基づくメモリレイアウト最適化が安定したベースライン性能の向上に寄与し，CCT によるアルゴリズム最適化が更なる性能向上の主要因となったことが示された．また第 3 章で提示した理論モデルと実測値との比較を行い，提案手法の有効性が理論的にも裏付けられるこ

とを確認した．さらに最適化の過程で生じる性能指標間のトレードオフを分析し，L1 リクエスト数のような本質的な処理量を削減することの重要性を示した．
以上の結果から，データレイアウトの最適化を土台としつつ，アルゴリズムによって根本的な処理量を削減するという二段階のアプローチは，対象データの特性によらず距離クエリカーネルの性能向上において有効な戦略であると結論付けられる．また該当カーネル高速化により，ADFs 構築プロセス全体においても最大 3.92 倍の性能向上を実現した．

第5章　おわりに

5.1　本研究の総括

5.1.1　研究の動機とアプローチ

本研究は，GPU 上での適応的距離場（ADFs）構築，特にその性能を律速する主要因である BVH トラバーサル処理の高速化を目的として出発した．分析の結果，性能ボトルネックは (1) 該当処理のメモリアクセス特性にとって非効率な AoS メモリレイアウト，および (2) 距離場構築の特性を考慮しない画一的な探索アルゴリズムという 2 点に起因することを特定した．この課題認識に基づき，本稿ではメモリレイアウトと探索アルゴリズムの両面から最適化を行う二段構えのアプローチを提案・実装した．

5.1.2　BVH メモリレイアウト最適化の成果

第一の提案である BVH ノードのデータレイアウトを AoS から SoA に基づくハイブリッド形式へ変更する最適化は，評価した全てのモデルにおいて安定して約 1.4 倍の性能向上を達成した．この成果は，アクセスパターンが異なるデータ（AABB と子インデックス）を物理的に分離することで GPU メモリアクセスの無駄を省き，コアレッシングを促進した結果と言える．単に実行時間が短縮されただけでなく，リクエスト 1 回あたりの転送データ量を示す‘Avg. Sectors/Req‘の値が改善されたことからも，メモリアクセスの質が向上したことが確認された．この手法は今回実験対象としたデータ形状を問わずベースラインの性能を向上する策として有効であることが示された．

5.1.3　探索アルゴリズム最適化の成果

第二の提案であるクエリ点からの距離情報を活用した「近傍子ノード優先探索（CCT）」アルゴリズムにより，追加で性能が向上した．本手法の適用により，距離クエリカーネルはベースライン比で最大 6.1 倍，ADFs 構築プロセス全体としても最大 3.9 倍の高速化を達成した．この高速化の主要因は，近いノードから優先的に探索することで最近傍距離を早期に更新し不要なサブツリーを効率的に枝刈

りできた点にある．結果として探索すべきノード総数が削減され，L1 キャッシュへのメモリアクセスリクエスト数は最大 93%削減された．一部モデルではアクセスパターンの変化により DRAM 読み込み量が増加するトレードオフが観測されたが，メモリアクセス回数そのものを大きく削減する効果がそれを上回った．これは GPU コンピューティングにおいて，アルゴリズムによって本質的な処理量を削減することの重要性を示すものである．

5.1.4 結論

以上の結果から，データレイアウトの最適化を土台としつつアルゴリズムによって根本的な処理量を削減するという本研究のアプローチは，GPU 上での ADFs 構築における性能ボトルネックを解消する汎用的な戦略であることが実証された．

5.2 今後の課題

本研究によって BVH トラバーサルの高速化は達成されたが，ADFs 構築全体のプロセスや関連技術との連携にはさらなる検討の余地がある．今後の課題として以下が挙げられる．

- **BVH 構築アルゴリズムとの統合**: 本研究は構築済みの BVH に対するトラバーサル処理の最適化に焦点を当てた．ADFs 構築パイプライン全体のさらなる高速化には [Benthin et al.(2024)] のような最新の高品質な BVH 構築アルゴリズムと本手法を統合し，構築からクエリまで一貫した最適化を図る研究が今後の課題となる．

- **キャッシュ効率を考慮したアクセスパターンの最適化**: 第 4 章で考察したように，一部モデルで観測された DRAM Read の増加はキャッシュの挙動をより深く考慮したアクセスパターンの最適化が今後の検討課題であることを示唆している．

- **データ構造のさらなる最適化**: 同様に，メモリアクセス効率の指標である‘Avg. Sectors/Req‘は提案手法によって改善されたが，依然としてコアレッシングの観点から理想値との間には乖離がある．AABB のような構造体をよりハードウェアに適した単位に分割・パッキングするなどデータ構造レベルでのさらなる最適化も今後の課題である．

- **一様グリッド手法との包括的比較**: 本研究は ADFs の持つ適応性（Adaptability）によるメモリ効率と距離クエリ母数削減の利点を前提としたが，一様グリッドベースの手法（JFA など）も GPU 上での並列処理に適している．一様グリッド手法と適応的グリッド手法間で，メモリ効率と距離クエリ速度以

外にも，構築速度，近似精度，実装・運用の容易性など複数の指標において定量的な比較・分析が加えられることが望ましい．

- **他分野への応用展開**: 本研究で高速化した距離クエリは，高速な衝突検出や物理シミュレーションなど他の応用分野へ展開できる可能性がある．各分野特有の要求に合わせて本手法を適用しその有効性を検証することも重要な課題である．

参考文献

[Apetrei(2014)] Ciprian Apetrei. 2014. Fast and Simple Agglomerative LBVH Construction. https://doi.org/10.2312/CGVC.20141206 Artwork Size: 4 pages ISBN: 9783905674705 Pages: 4 pages Publication Title: Computer Graphics and Visual Computing (CGVC).

[Benthin et al.(2022)] Carsten Benthin, Radoslaw Drabinski, Lorenzo Tessari, and Addis Dittebrandt. 2022. PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited. Proceedings of the ACM on Computer Graphics and Interactive Techniques 5, 3 (July 2022), 1–13. https://doi.org/10.1145/3543867

[Benthin et al.(2024)] Carsten Benthin, Daniel Meister, Joshua Barczak, Rohan Mehalwal, John Tsakok, and Andrew Kensler. 2024. H-PLOC: Hierarchical Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. Proceedings of the ACM on Computer Graphics and Interactive Techniques 7, 3 (Aug. 2024), 1–14. https://doi.org/10.1145/3675377

[Cetin et al.(2016)] Omer Cetin, Selcuk Keskin, and Taskin Kocak. 2016. Real-Time FFT Computation Using GPGPU for OFDM-Based Systems. Circuits, Systems, and Signal Processing 35, 3 (March 2016), 1021–1044. https://doi.org/10.1007/s00034-015-0106-5

[Chen et al.(2022)] Xiao-Rui Chen, Min Tang, Cheng Li, Dinesh Manocha, and Ruo-Feng Tong. 2022. BADF: Bounding Volume Hierarchies Centric Adaptive Distance Field Computation for Deformable Objects on GPUs. Journal of Computer Science and Technology 37, 3 (June 2022), 731–740. https://doi.org/10.1007/s11390-022-0331-x

[Frisken et al.(2000)] Sarah F. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thouis R. Jones. 2000. Adaptively Sampled Distance Fields: A General Representation of Shape forComputer Graphics. In Seminal Graphics Papers: Pushing the Boundaries, Volume 2 (1 ed.), Mary C. Whitton (Ed.). ACM, New York, NY, USA, 173–178. https://doi.org/10.1145/3596711.3596732

[Karras(2012)] Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. https://doi.org/10.2312/EGGH/HPG12/033-037 Artwork Size: 5 pages ISBN: 9783905674415 ISSN: 2079-8679 Pages: 5 pages Publication Title: Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics.

[Karras and Aila(2013)] Tero Karras and Timo Aila. 2013. TRBVH: Fast parallel construction of high-quality bounding volume hierarchies. In Proceedings of the 5th High-Performance Graphics Conference. ACM, Anaheim California, 89–99. https://doi.org/10.1145/2492045.2492055

[Lauterbach et al.(2009)] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. LBVH: Fast BVH Construction on GPUs. Computer Graphics Forum 28, 2 (April 2009), 375–384. https://doi.org/10.1111/j.1467-8659.2009.01377.x

[Liu and Kim(2014)] Fuchang Liu and Young J. Kim. 2014. Exact and Adaptive Signed Distance FieldsComputation for Rigid and DeformableModels on GPUs. IEEE Transactions on Visualization and Computer Graphics 20, 5 (May 2014), 714–725. https://doi.org/10.1109/TVCG.2013.268

[Meister and Bittner(2018)] Daniel Meister and Jiri Bittner. 2018. PLOC: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction. IEEE Transactions on Visualization and Computer Graphics 24, 3 (March 2018), 1345–1353. https://doi.org/10.1109/TVCG.2017.2669983

[Rong and Tan(2006)] Guodong Rong and Tiow-Seng Tan. 2006. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In Proceedings of the 2006 symposium on Interactive 3D graphics and games - SI3D '06. ACM Press, Redwood City, California, 109. https://doi.org/10.1145/1111411.1111431

[Slavcheva(2018)] Miroslava Slavcheva. 2018. Signed Distance Fields for Rigid and Deformable 3D Reconstruction. (2018).

[Stratton et al.(2012)] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Li-Wen Chang, Nasser Anssari, Geng Liu, Wen-mei W. Hwu, and Nady Obeid. 2012. Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems. Computer 45, 8 (Aug. 2012), 26–32. https://doi.org/10.1109/MC.2012.194

[Sud et al.(2004)] Avneesh Sud, Miguel A. Otaduy, and Dinesh Manocha. 2004. DiFi: Fast 3D Distance Field Computation Using Graphics

Hardware. Computer Graphics Forum 23, 3 (Sept. 2004), 557–566. https://doi.org/10.1111/j.1467-8659.2004.00787.x

[Vinkler et al.(2017)] Marek Vinkler, Jiri Bittner, and Vlastimil Havran. 2017. EMC: Extended Morton codes for high performance bounding volume hierarchy construction. In Proceedings of High Performance Graphics. ACM, Los Angeles California, 1–8. https://doi.org/10.1145/3105762.3105782

謝辞

本研究の遂行にあたって，主指導教員である井口 寧教授には学術研究の進め方や科学的態度の重要性，論文執筆の要点から細部にわたるまで丁寧なご指導を賜りました．研究をまとめることができたのは井口先生のご支援の賜物であり，深く感謝を申し上げます．
また，中間審査で重要なご助言をいただきました上原 隆平教授，田中 清史教授と，入学時の研究室選定および副テーマ研究にて多大なご指導をいただいた宮田 一乗教授に心より御礼申し上げます．
井口研究室の皆様には，研究の進捗共有や互いに対するアドバイスなど多くの刺激をいただき，研究活動を進める糧とすることができました．あらためて感謝を申し上げます．
最後に，学業と仕事で多忙な私を日々献身的に支えてくれた家族に心から感謝の言葉を捧げます．