スケールフリーグラフにおける効率的な幅優先探索クエリ処理のGPU実装と評価

今西　遼人† 尾形　嵐士† 及川　一樹† 新井　淳也† 山口　滉平†
森下　慎次† 藤枝　崇史† 小田　哲†† 藤澤　克樹†††

† NTT コンピュータ＆データサイエンス研究所　〒 180–8585 東京都武蔵野市緑町 3-9-11
†† NTT 情報ネットワーク総合研究所　〒 180–8585 東京都武蔵野市緑町 3-9-11
††† 九州大学 マス・フォア・インダストリ研究所　〒 819–0395 福岡市西区元岡 744 番地
E-mail:
†{ryoto.imanishi,ogata.arashi,kazuki.oikawa,junya.arai,kohei.yamaguchi,morishita.shinji,takafumi.fujieda}@ntt.com,
††satoshi.oda.ar@hco.ntt.co.jp, †††fujisawa@imi.kyusyu-u.ac.jp

あらまし　幅優先探索 (breadth-first search, BFS) は媒介中心性の計算や最短経路クエリなど様々なグラフ分析の基礎となる重要な処理である．これらの分析では同じグラフに対して異なる始点からの BFS を繰り返し行うが，その都度不規則なメモリアクセスを伴いながら全頂点を訪問するため計算コストが高い．さらに実世界グラフのスケールフリー性は GPU などを用いた並列処理において負荷の偏りによるスケーラビリティ低下を招く．本稿では事前計算とメモリアクセスの削減によって効率的に BFS を行う手法を提案するとともに，その手法の実装をアーキテクチャが異なる 2 種類の GPU に向けて最適化する．スケールフリーな合成グラフの一種である Kronecker グラフを用いた評価において，提案手法は性能を約 2.5 倍向上させることが確認された．

キーワード　グラフデータ処理，先進ハードウェア活用・GPU，並列・分散処理

1 はじめに

大規模グラフデータ処理は，巨大集積回路の最適化や道路網の最適化，全脳シミュレーション，検索エンジン，創薬，遺伝子解析など様々な分野で活用されている [1–4]．グラフ処理において幅優先探索 (breadth-first search, BFS) は特に広範に用いられる操作である．媒介中心性の計算や最短経路クエリ処理などは異なる始点からの BFS を繰り返し行うことから，大規模グラフの活用には効率的な BFS が必要とされる．

BFS の性能向上にあたっては次のような課題がある．まず，一般的にグラフ処理は不規則なメモリアクセスを伴う [5]．そのためキャッシュヒット率が低く，メモリアクセスがボトルネックとなりやすい．さらに実世界のグラフはしばしばスケールフリー性を持ち，GPU などによる並列処理において計算負荷が偏りやすい．スケールフリーとは，少数の頂点のみが大きい次数を持つ一方で大多数の頂点は小さい次数を持つというグラフの性質である. 例えば SNS の友達関係などを表すソーシャルネットワークグラフはスケールフリーであることが知られている．頂点ごとにスレッドを割り当てる単純な並列化方式では高次数頂点の処理を他のスレッドが待つことになり，計算の効率が悪い．これまでもメモリアクセス [6–12] や並列処理 [7, 13–20] について様々な工夫がなされてきた．しかし GPU の理論性能と比べ低い性能に留まっており [21]，依然として性能向上の余地がある．

本稿ではスケールフリーグラフにおける幅優先探索の高速化手法として (i) CoreBFS と (ii) m_f 近似を提案する．CoreBFS は始点によらず共通する探索結果を事前に計算しておくことで，始点が与えられた際の BFS の探索範囲を縮小する．これにより始点を変えながら繰り返し BFS を行う場合に冗長な計算が省略され，処理時間を短縮できる．一方，m_f 近似はグラフのスケールフリー性を前提とした近似の導入により不規則なメモリアクセスを削減する．具体的には，効率的な BFS 手法として知られる Hybrid BFS [6] における探索方法の切り替えパラメータの計算を近似する．Hybrid BFS は広く用いられている [7, 18–20] にもかかわらず，驚くべきことにこのパラメータの計算コストにはこれまで注意が払われてこなかった．m_f 近似はそのコストを大幅に低減する．

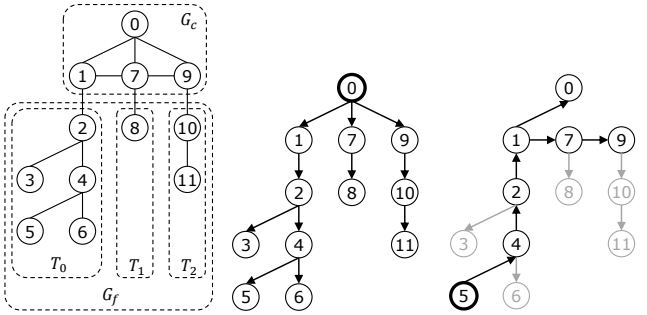
本稿では提案手法について NVIDIA RTX A6000（以下, A6000）と Apple M1 Max（以下，M1 Max）の 2 種のアーキテクチャに対し実装と性能評価を実施した．提案手法によってベースライン手法と比較して A6000 で 2.5 倍程度，M1 Max で 1.4 倍程度の高速化を達成した．特に大規模グラフ処理の性能評価プロジェクトである Graph500 [22, 23] の BFS ベンチマークにおいて，M1 Max 実装はディスクリート GPU や FPGA を用いない市販品の計算機として最も高い電力効率を達成した．

2 背　　景

本節では本稿で取り組む問題の定義，BFS に関する基本的なテクニック，および GPU について概説する．

表 1: 本稿で用いる主な記号

記号	意味
G	単純無向グラフ
$V(G), E(G)$	G の頂点の集合, 辺の集合
$N(v)$	ノード v の隣接頂点集合
$d(v)$	ノード v の次数
s	BFS の開始頂点



(a) グラフ (b) 頂点 0 からの BFS (c) 頂点 5 からの BFS

図 1: CoreBFS の例

2.1 問題定義

本稿では単純無向グラフ G を考える．$V(G)$ は頂点集合であり，$E(G)$ は辺の集合である．本稿で用いる主な記号を表 1 に定義する．BFS による探索経路は *BFS* 木として表現できる．頂点 s を始点とする BFS 木とは，s から到達可能な全ての頂点を含み，各頂点が自身より s までの距離が 1 つ大きい隣接頂点を子として持つような根付き木 (rooted tree) である．図 1a にグラフの例を示す．各頂点は番号で区別する．このグラフ上で頂点 0 を始点として構築された BFS 木を図 1b に示す．矢印は親から子へ向かっている．本稿では次のように定義される ***BFS* クエリ問題**を効率的に解くための手法を説明する．

定義 **1** (BFS クエリ問題)**.** 頂点 s が与えられたとき，s を始点とする BFS 木を求める問題を ***BFS* クエリ問題**と呼ぶ．

なお，BFS 木は各頂点の親を格納した**親配列**によって表現することもできる．頂点 s を始点とする BFS 木に対応する親配列 π_s とは次のような配列である : (i) $\pi_s[s] = s$, (ii) s から到達可能かつ s と異なる頂点 u について $\pi_s[u]$ は u の親，かつ (iii) s から到達不能な頂点 u について $\pi_s[u] = -1$. 例えば図 1b の BFS 木に対応する親配列を π_0 と置くと，$\pi_0[0] = 0$, $\pi_0[1] = 0$, $\pi_0[2] = 1$, ... となる．以降では BFS 木と親配列を同一視する．つまり BFS 木を求めることは親配列を求めることと同義である．

2.2 Hybrid BFS

効率的な BFS 手法として Hybrid BFS [6] が広く用いられている．Hybrid BFS は Topdown BFS と Bottomup BFS の 2 つを切り替えながら探索する．Topdown BFS はある始点 s から順に隣接頂点を調べ全ての頂点を探索する．頂点は level 単位で探索される．頂点の level が i であるとは，s からの距離が i であることと同義である．s からの距離が $i+1$ である頂点の集合を level i の frontier 集合と呼ぶ．例えば，図 1b の BFS 木において，level 0 の frontier 集合は $\{1, 7, 9\}$，level 1 の frontier 集合は $\{2, 8, 10\}$，level 2 の frontier 集合は $\{3, 4, 11\}$ である．また，frontier 集合に属する頂点のことを単に frontier 頂点と呼ぶ．Topdown BFS は frontier 集合に属する頂点から隣接する全ての頂点をチェックし未訪問頂点が存在するかどうかを確認する．そのため，frontier 集合に属する頂点 $v_{frontier}$ の隣接頂点に未訪問頂点が存在するとき，$v_{frontier}$ は BFS 木において親となる．逆に，Bottomup BFS は未訪問の頂点に隣接する頂点から探索元頂点を見つける．つまり，ある未訪問頂点が frontier 頂点に隣接するとき，その頂点は frontier 頂点の子となる．Topdown BFS は frontier 頂点に接続された全ての辺を通過して隣接頂点を訪問するため，frontier 頂点に接続された辺の数が大きいとき非効率的である．一方，Bottomup BFS では未訪問頂点に接続された全ての辺を調べる．未訪問頂点の中には frontier 頂点に隣接していない頂点も含まれ，それらの頂点の接続辺を調べることは無駄である．そのため，frontier 頂点が少ないときには非効率的である．

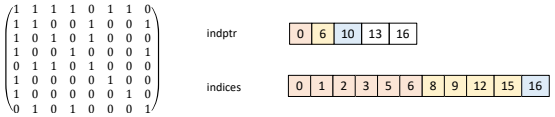


図 2: CSR 形式

Hybrid BFS は探索の進行状況に応じて Topdown BFS と Bottomup BFS を適切に切り替えることで効率的に探索する．切り替えタイミングの決定には frontier 頂点に接続される辺の総数 m_f，frontier 集合の頂点の数 n_f，及び未訪問頂点に接続される辺の数 m_u を用いる．m_f は各 level の探索時に各頂点の次数をカウントして総和をとることで求める．m_u は直接数えることが難しいため．実際には 1 頂点に接続される辺の数の平均である $\bar{d}$ と未探索頂点数 n_u を用いて，$m_u = \bar{d}n_u + |G(V)|$ で近似値を得る．Topdown BFS では各 level で常に m_f 個の辺をチェックし, Bottomup BFS では最大で m_u 個の辺をチェックする．よって $m_f > m_u$ ならばその時点で Topdown BFS から Bottomup BFS に切り替えたほうがより少ないエッジを探索することが保証される．実際には Bottomup BFS でチェックされる辺は m_u 個より小さいので $\alpha \cdot m_f > m_u$ のとき Topdown BFS から Bottomup BFS への切り替えを行う．ただし，α は任意に定めるチューニングパラメータである．一方で Bottomup BFS のオーバーヘッドは全頂点をスキャンすることと，frontier 頂点と直接接続していない頂点に訪問することである．従って，十分に探索が進み $m_u > m_f$ となるとき Bottomup BFS のメリットが消失するため Topdown BFS への切り替えが必要になる．実際には Bottomup BFS に接続された辺は m_u 個より小さいのでチューニングパラメータ β を用いて $\beta \cdot m_u > m_f$ のとき Topdown BFS から Bottomup BFS への切り替えを行う．

アルゴリズム 1 は Hybrid BFS の疑似コードである．G の表現は隣接行列を Compressed Sparse Row (CSR) 形式で圧縮したものを用いる．CSR 形式のグラフの表現は出力頂点番号を格納

アルゴリズム **1** Hybrid BFS

```
入力: 始点 s，グラフ G
出力: 親配列 π
 1: f = [0,0,...,0], f' = [0,0,...,0]          ▷ frontier 配列の初期化
 2: π = [−1,−1,...,−1]                          ▷ 親配列の初期化
 3: f[s] = 1                                    ▷ 始点 s にビットを立てる
 4: Level = 0, n_next = 1, m_f = 0              ▷ Level，次の frontier 頂点数，m_f の初期化
 5: while n_next > 0 do                         ▷ 探索終了か判定
 6:   if next-direction = top-down then         ▷ Topdown と Bottomup の切り替え判定
 7:     TopdownBFS(f, f', π)                    ▷ Topdown BFS
 8:   else
 9:     BottomupBFS(f, f', π)                   ▷ Bottomup BFS
10:   swap(f, f')
11:   Level += 1
12:
13: function TopdownBFS(f, f', π)
14:   for each v ∈ f s.t. v ≠ 0 do
15:     for each u ∈ N(v) s.t. π[u] = −1 do
16:       if π[u] = −1 then
17:         π[u] = v
18:         f'[v] = 1
19:         n_next += 1
20:         m_f += d(v)
21:
22: function BottomupBFS(f, f', π)
23:   for each v ∈ V(G) s.t. π[v] = −1 do
24:     for each u ∈ N(v) s.t. u ∈ f do
25:       π[v] = u
26:       f'[v] = 1
27:       n_next += 1
28:       m_f += d(v)
29:       break
```

する配列 indices と, 辺の頂点番号のオフセット配列 indptr からなる（図 2）．1 行目の f，f' はそれぞれ現在の level の frontier 頂点と次の level の frontier 頂点を表す配列である．頂点 v が frontier に属すとき $f[v] = 1$，属していないとき $f[v] = 0$ とする．f' も同様である．要素の値が 0 または 1 なので，実装上は各頂点の値を 1 ビットで表現したビットマップ配列として f と f' を定義する．2 行目の π は親配列であり，全ての頂点について $\pi[v] = -1$ で初期化される. 4 行目の n_{next} は次の level の frontier 頂点の数である．$n_{\text{next}} = 0$ となったとき探索を終了する．17 行目，25 行目において v の parents が u であるとき $\pi[v] = u$ を代入する．6 行目の next-direction は Topdown BFS と Bottomup BFS の切り替えを判定する．15 行目，24 行目の $N(v)$ は頂点 v の隣接頂点集合，20 行目，28 行目の $d(v)$ は頂点 v の次数を表す．CSR において $M(v)$ は indices 配列の indptr[v] 番目から indptr[$v + 1$] − 1 番目の要素に対応する．同様に $d(v)$ は indptr[$v + 1$] − indptr[v] で求められる．

2.3 次数オーダリング

Hybrid BFS としばしば併用されるテクニックとして次数オーダリング [7] がある．アルゴリズム 1 の BottomupBFS 関数に示したように，Bottomup BFS は隣接頂点の中に frontier 頂点が見つかった時点でループを抜ける．従って，frontier 頂点となっている確率が高い頂点から順番に確認することで早期にループを脱出できる可能性が高まる．ここで，多くの頂点に隣接する高次数頂点は BFS の初期に訪問されやすいことから，それぞれの level において frontier となっている確率が高い．そこで各頂点の頂点番号を次数が大きい順に振りなおし，さらに隣接頂点集合を頂点番号の昇順にソートすることで次数の昇順にループする．このようなテクニックは次数オーダリングと呼ばれ，Hybrid BFS の効率化のために広く用いられている [18–20].

本稿では Hybrid BFS と次数オーダリングの組み合わせを**ベースライン**と呼称する．提案手法はベースラインをさらに効率化するためのテクニックから成る．

表 2: NVIDIA RTX A6000 と Apple M1 Max の仕様の比較

	A6000	M1 Max
SM・コア数	84	32
CUDA コア・演算ユニット数	10,752	4,096
ブーストクロック	1,800 MHz	1,296 MHz
メモリサイズ	48 GB	64 GB
メモリ帯域	768 GB/s	400 GB/s
理論ピーク性能	38.7 TFLOPS	10.4 TFLOPS
消費電力	300 W	不明

2.4 GPU

GPU は高い並列演算性能を持つことからグラフ処理においても広く利用されている [15, 16]．GPU の性能を最大限に発揮するためには，Single Instruction Multiple Thread (SIMT) 方式を意識した実装をすることが鍵となる．SIMT は多くの GPU で採用されている実行モデルであり，warp と呼ばれるスレッドのグループが同じ命令を同時実行する．もし warp 内の特定スレッドのみ負荷が高い処理となっている場合，他スレッドは待機しなければならないため，warp 内スレッド間の負荷をいかに均一にできるかが高速化において重要である．また warp 内スレッドから不規則なアクセスパターンでメモリにアクセスすると，レイテンシが大きくなってしまう．そのため warp 内の隣接スレッドが連続したメモリアドレスにアクセスするコアレスド (coalesced) アクセス可能なメモリレイアウトを設計することも高速化のための有用な手段と言える．このような SIMT の特性を意識した具体的な実装方法については第 4 章で詳しく説明する．また，本研究で使用する A6000 と M1 Max の仕様 [24] を表 2 に示す.

2.5 関 連 研 究

本節では BFS を高速化するためのアプローチを 3 つに分類しそれぞれ俯瞰する．

(1) 通過辺の削減　訪問済み頂点の辺を辿りながら未訪問頂点の発見を繰り返すナイーブな Topdown BFS ではグラフに存在する全ての辺を通過することになる．これに対し，Beamer らは Bottomup BFS を組み合わせた Hybrid BFS で通過辺を削減できることを示した [6]．さらに Yasui らは次数オーダリングによって Bottomup BFS における通過辺をさらに削減した [7]. これらは近年の BFS 手法において標準的なテクニックとなっ

ているが [18–20]，我々が知る限りこれらの他に効果的な通過辺の削減手法は提案されていない．

(2) データレイアウトの最適化　BFS で生じる不規則なメモリアクセスの局所性を改善するために頂点順序の並べ替え（リオーダリング）がしばしば用いられる [8,9]．前述の次数オーダリングは局所性を高める効果もある [7]．また，グラフ構造の保持に用いるデータ形式の工夫でデータサイズと間接参照を削減することも効果的である [10,11]．GPU においては複数スレッドからのメモリアクセスの集約 (memory coalescing) が BFS の性能向上に大きく寄与することが確認されている [12]．

(3) 並列化　CPU を用いた並列化では NUMA 環境向けの最適化が行われている [7] ほか，グラフ処理フレームワークの Ligra [13] および Galois [14] もスケーラブルな BFS 実装を提供している．GPU 向けには Enterprise [15], Gunrock [16] などのシステムが提案されている．また，AI アクセラレータである Graphcore を BFS に適用した研究もある [17]．分散メモリ環境のスーパーコンピュータにおいても多くの研究が行われており，近年では富岳 [18]，Sunway TaihuLight [19]，Tianhe [20] での成果が発表されている．

本論文で導入するテクニックはそれぞれ上記の 3 つのアプローチに対応する．CoreBFS は探索範囲を 2-core に限定することで通過辺を削減する．m_f 近似は不規則なメモリアクセスの回数を削減する．最適化 GPU 実装は 2 つの GPU についてそれぞれの機能を活用し並列処理性能を引き出すための方法を示している．

3 提案手法

本節では CoreBFS と m_f 近似の詳細について説明する．

3.1 CoreBFS

CoreBFS は BFS クエリ問題を効率的に解くために事前計算を用いる．具体的には，適当に選んだ頂点を始点とした場合の親配列を事前に求め，後続の BFS クエリでは指定された始点に対する親配列と事前計算した親配列の差分のみ計算する．これによりグラフ全体を探索することなく効率的に BFS 木を求める．例えば，頂点 5 を始点とする BFS 木（図 1c）のうち，グレーアウトされた頂点の親は頂点 0 を始点とする BFS 木（図 1b）と同じである．これらの頂点への訪問を省略することで BFS クエリの処理時間を短縮する．

CoreBFS の詳細を説明する．簡潔さのため G が連結であると仮定する．親が変化し得る，つまりクエリ処理時に探索する必要がある頂点は，グラフ G の 2-core [25] G_c に着目することで発見できる．グラフ G の 2-core とは，すべての頂点が 2 以上の次数を持つ G の最大連結部分グラフである閉路を構成する頂点は必ず次数が 2 以上であるから，G から G_c を取り除いたグラフ G_f は閉路を含まないグラフとなる．つまり G_f は 1 つ以上の木 T_i から成る森である．ここで T_i は G_c と接続する頂点を根とする根付き木と考えることにする．すると，T_i が始点を含む場合を除き，T_i 自体が常に BFS 木の一部を成し構造が変化しない．従って，BFS クエリが与えられた際は G_f への訪問を省略し G_c のみ探索することで親配列を得られる．また始点が T_i に含まれる場合は，その始点から G_c へ至るパス上の親子関係を逆転した上で，T_i と接続する頂点から G_c 内で BFS を行うことで親配列を得られる．この手順を図 1a のグラフを用いて説明する．G_c はこのグラフの 2-core を，G_f は 2-core を除いた部分（森）を，T_0, T_1, T_2 はそれぞれ G_f に含まれる木を示す．頂点 0 を始点とする BFS 木は事前計算により既知であるとし，さらに BFS クエリで与えられた始点は頂点 5 とする．このとき，CoreBFS はまず頂点 5 から G_c へ至るパス上の頂点（頂点 5, 4, 2, 1）の親子関係を逆転させる．さらに，G_c に到達して最初の頂点である頂点 1 から G_c 内で BFS を行う．これにより頂点 5 を始点とする BFS 木（図 1c）が得られる．BFS を行う範囲が 2-core に限定されるため，グラフ全体を探索するよりも効率的である．

ここまでは G が連結である場合について述べたが，非連結である場合も最も大きい連結成分に対して我々のアプローチを適用することで効果的に高速化できる．これは多くの実世界のグラフには極端に大きな連結成分 (*giant connected component, GCC*) が含まれるためである [26]．GCC 以外の連結成分は相対的に小さいことから，提案手法によることなく短時間で BFS を終えることができる．

CoreBFS の事前計算とクエリ処理をそれぞれアルゴリズム 2 と 3 に示す．これらのアルゴリズム中の BFS では Hybrid BFS に m_f 近似を組み合わせたものを用いる．親配列の定義に従えば始点 s からの BFS では $\pi_s(s)$ に s を代入するが，アルゴリズム 3 の 9 行目の BFS では $\pi_s(s)$ を上書きしない．これは 6–8 行目のループで $\pi_s(s)$ に正しい親が代入されているためである．また，アルゴリズム 2 の 2 行目で 2-core を発見するためには例えば Batagelj と Zaveršnik の手法 [25] を用いることができる．以上のような方法で CoreBFS は効率的に親配列を計算する．

アルゴリズム 2 事前計算

入力: グラフ G
出力: GCC G_g，2-core G_c，親配列 π_r
1: G の GCC を求め G_g と置く
2: G_g の 2-core を求め G_c と置く
3: $V(G_c)$ から任意の頂点 r を選ぶ
4: G 上で r から BFS を行い親配列を π_r と置く

3.2 m_f 近似

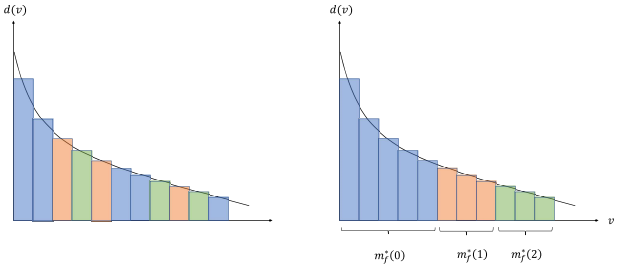
本節では m_f 近似について説明する．Hybrid BFS では Topdown BFS と Bottomup BFS の切り替え判定のために frontier 頂点に接続された辺の数の和 m_f を用いる．m_f の計算は，ナイーブには frontier 頂点 v を発見するたびに $d(v)$ (= indptr[$v+1$] − indptr[v]) を m_f へ加算することで行われる．従って BFS 全体では $2|V(G)|$ 回 indptr 配列へのメモリアクセスが発生する．頂点 v の出現順序は予測できないことから，このアクセスはキャッシュヒット率が低くコストが高い．

提案手法はこのメモリアクセスを省略するために m_f の代わり

アルゴリズム **3** クエリ処理

入力: 始点 s，グラフ G，GCC G_g，2-core G_c，親配列 π_r
出力: s を始点とした場合の親配列

if $s \notin V(G_g)$ **then**
  G 上で s から BFS を行い，得た親配列を出力
  return
$\pi_s = \pi_r$
$\pi_s(s) = s$
while $s \notin V(G_c)$ **do**
  $\pi_s(\pi_r(s)) = s$
  $s = \pi_r(s)$
G_c 上で s から BFS を行い，s 以外の訪問頂点について π_s を上書き
π_s を出力



(a) 実際の訪問順　　(b) m_f 近似が仮定する訪問順

図 3: 訪問順と次数の比較

に近似値 m_f^* を導入する．2.3 節で説明したように，探索の初期には次数の大きい頂点が探索される確率が高く，探索が終了に近づくにつれて次数の小さい頂点が探索される確率が高いと考えられる．m_f 近似はこの性質を利用する．即ち，頂点の訪問順が完全に次数によって決定されると仮定する．次数オーダリングされているときこの仮定は頂点番号の小さい順に探索されることと同義である．頂点 v の次数は $d(v) = \mathrm{indptr}[v+1] - \mathrm{indptr}[v]$ であるため，level i での m_f の近似値 $m_f^*(i)$ は次のように表せる：

$$m_f^*(i) = \mathrm{indptr}[n_{\mathrm{end}}(i)] - \mathrm{indptr}[n_{\mathrm{start}}(i)]$$

ただし $n_{\mathrm{start}}(i)$，$n_{\mathrm{end}}(i)$ はそれぞれ level i の探索開始および終了の時点で訪問済みとなっている頂点の数である．図 3 に頂点の訪問順と次数の比較を示す．横軸は頂点（左から頂点 0，頂点 1，頂点 2，...），縦軸はその頂点の次数，棒の色はそれぞれのレベルにおける frontier を表す．即ち，青色，橙色，緑色の棒に対応する頂点はそれぞれ level 0, 1, 2 の frontier である．各 level の m_f は同じ色の頂点の次数の和である．BFS では必ずしも次数順に頂点が訪問されるわけではないものの，高次数頂点ほど先に訪問されやすい傾向にある（図 3a）．一方，m_f 近似は完全に次数順に頂点が訪問されるものと仮定する（図 3b）．例えば level 0 の frontier 頂点は 5 つあるので，頂点 0–4 が frontier であるものとして m_f^* を計算する．このように計算した m_f^* に基づいて提案手法は Topdown BFS と Bottomup BFS の切り替えを行う．ここで，与えられたグラフにおける BFS の level の最大値を k とおくと，m_f^* の計算で生じる indptr 配列へのメモリアクセスは BFS 全体で $2k$ 回である．スケールフリーグラフの直径は一般に頂点数よりも十分小さい [27] ため，$k \ll |V(G)|$ が成り立つ．ナイーブな m_f の計算では $2|V(G)|$ 回のアクセスが生じるため，m_f を m_f^* に置き換えることでメモリアクセス回数を大幅に削減することができる．

4 実　　装

本節ではそれぞれの GPU に向けた実装について詳述する．

4.1 NVIDIA RTX A6000 への実装

A6000 への実装には CUDA を採用した．Frontier 頂点を示す配列 f にビットマップを用いた Hybrid BFS（アルゴリズム 1）を CUDA で実装する際，スタンダードな方法としてビットマップ配列を構成する整数値ごとに並列実行する方法が挙げられる．ビットマップ配列の各要素を 64 ビット整数とした場合，1 スレッドあたり 1 要素，すなわち 64 頂点を処理するような実装を行うことで GPU 特有のマルチコアを活かした並列処理が可能となり高速化が期待できる．しかしこのような実装では高次数頂点を処理する特定のスレッドに負荷がかかる一方で早期終了したスレッドが長時間待機させられるといった処理時間の不均一が問題となり，性能が低下する傾向にある．そこでさらなる負荷分散を可能とするため，NVIDIA GPU 特有の機能である Dynamic Parallelism (DP) [28,29] を使用する．通常 CUDA で GPU 上の処理を行う場合 CPU から CUDA カーネルを起動するが，DP を用いることにより GPU 上で動いているカーネル (Parent Kernel) 内から動的に新しい子カーネル (Child kernel) を起動できる．この機能を用いて負荷集中の原因となる次数が大きい頂点の処理をさらに並列化することでスレッド間での負荷均一化が可能となり，全体の性能向上が見込まれる．図 4(b) で具体例を示しており，高次数頂点 u_0 を担当するスレッド t_0^p 内で子カーネルを動的に起動し，u_0 の各隣接頂点を各スレッド t_i^c で並列に処理している．また，DP の導入に伴い 1 スレッドあたり 1 頂点を処理するよう修正し並列性を高めた．

4.2 M1 Max への実装

M1 Max の GPU を利用するために Metal [30] を利用して BFS のカーネルを実装した．この実装では 2 つの工夫を行っている．

a） コアレスドアクセス可能なメモリレイアウト

: Metal では DP のような機能がないため，図 4(c) のように隣接頂点リストを固定ブロック長 L（図では $L = 4$）で区切り各スレッドに割り当てることでスレッド間での負荷均一化を図った．このとき各頂点の隣接頂点のうち最初の L 個を処理するスレッドを初めに起動し，次に次数が L よりも高い頂点について後続の L 個の隣接頂点の処理するスレッドを立ち上げる．これにより Bottomup BFS を効率的に処理できる．2.3 節で述べた通り，隣接頂点リストの先頭ほど frontier の存在確率は高く，さらにひとつ frontier 頂点が見つかればループから脱出してよい．ところが特定の頂点の隣接頂点を複数スレッドで処理すると，いずれかのスレッドが frontier を発見しても他のスレッドの終了を待たなければならない．これに対し M1 Max の実装のように

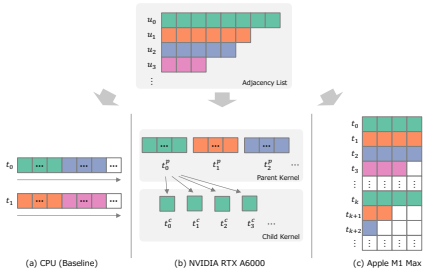


図 4: アーキテクチャごとのワークロード

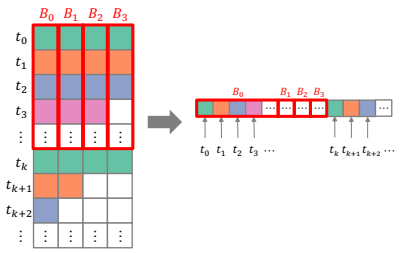


図 5: コアレスドアクセス可能なメモリレイアウト

スレッドを割り当てると，先頭から L 個ずつの隣接頂点ブロック内に frontier が見つかった頂点を以降の処理から除外できるという利点がある．さらに warp 内の各スレッドがコアレスドアクセスできるようメモリに配置することが望ましいため，先行研究 [31] で提案されたメモリレイアウト手法を採用し，メモリ上に図 5 のように配置した．図 4(c) のスレッド割り当てに対し，各ブロック B_i $(i \in \{0, 1, ..., L-1\})$ で分割し B_0, B_1, ..., B_{L-1} の順でメモリ空間に並べて格納する．このように配置することでワープ内スレッドが連続したメモリ領域にアクセスすることができるためアクセス効率が大幅に向上する．

b） CPU による初回の Topdown BFS

Level 0 の BFS は始点 s のみから辺を辿る処理であり，データ並列性が低い．そのため．ユニファイドメモリであるという特徴を活かし level 0 の BFS のみ CPU 側で処理を行う．

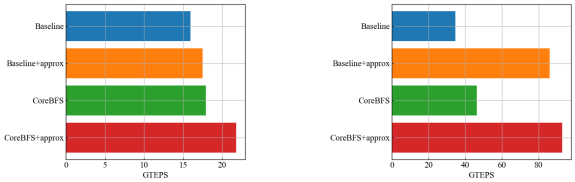
5 評価実験

本節では提案手法の有効性を確かめるための実験方法とその結果について説明する．性能評価には Kronecker グラフを用いた．Kronecker グラフはスケールフリーな合成グラフの一種であり，現実世界のグラフの性質を反映している．実験に使用した Kronecker グラフの頂点数は 2^{26}，辺の数は 2^{30} である．実験では同一の Kronecker グラフに対して異なる 64 の頂点から BFS を行った．Hybrid BFS の切り替えパラメータは先行研究 [6] を参考に $\alpha = 16$，$\beta = 16$ を用いた．

5.1 提案手法の TEPS の比較

既存手法と提案手法について性能向上の度合いを調べるために,Traversed Edges Per Second(TEPS) を計測した. TEPS は BFS の始点が属する連結部分の辺の数を実行時間で割ったものとして定義する．

図 6 は A6000 と M1 Max のそれぞれについて提案手法の



(a) M1 Max (b) A6000

図 6: 実装ごとの TEPS

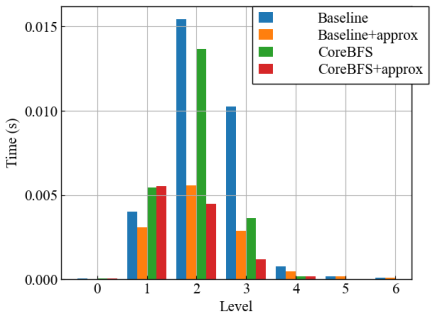


図 7: level 毎の実行時間の比較 (A6000)

TEPS を測定した結果である．図中の baseline はベースライン実装を，2-core, approx は各ベースライン実装に対して CoreBFS 及び m_f 近似を適用したものを表す．提案手法はベースライン実装と比較して M1 Max で 1.4 倍程度，A6000 で 2.5 倍程度 BFS を高速化している．このうち，A6000 においては m_f 近似の寄与が顕著に表れる．A6000 と M1 Max の性能向上の違いはアーキテクチャの構造の違いを反映していると考えられる．

5.2 Level 毎の実行時間の比較

次に, 実装の高速化の詳細を調べるために実装ごとに各 level での BFS の実行時間を計測した．実験条件は図 6 の場合と同一である. 図 7 は A6000 実装の各 level での実行時間を示す. 概ね全ての level において提案手法による高速化がなされている．また，CoreBFS では探索の最大 level が小さくなっている．これは CoreBFS による探索辺数の削減の効果である．CoreBFS は level 1 で既存手法よりも実行時間が長くなる．これは，CoreBFS では 2-core 内の頂点を始点として level を数えているため通常の BFS と level のカウントがずれる場合があることが理由である．例えば，始点を 5 とする図 1c の BFS 木において，通常の BFS では頂点 1 は level 2 で探索されるが，CoreBFS では level 0 で探索される．図 8 は M1 Max 実装の各 level での実行時間を示す. A6000 の場合と同様に CoreBFS では探索の最大 level が小さくなっている．一方で CoreBFS による高速化の度合いは A6000 と比較して小さい．これはアーキテクチャの構造の違いを反映していると考えられる．

5.3 m_f と m_f^* の比較

m_f 近似について，m_f^* がどの程度正確に近似されているか確かめるために，各 level での m_f と m_f^* の値を計算した．

図 9 は Topdown BFS と Bottomup BFS の切り替えタイミン

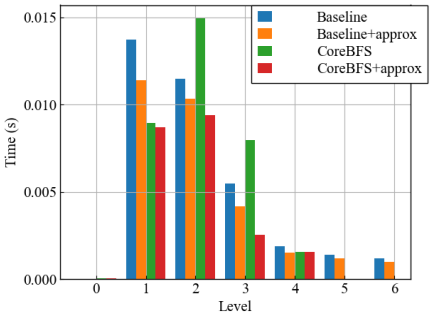


図 8: level 毎の実行時間の比較 (M1 Max)

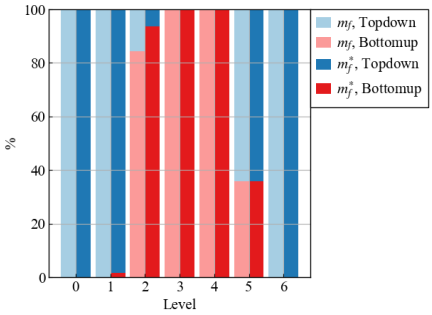


図 9: Topdown BFS と Bottomup BFS の切り替えタイミングの比較

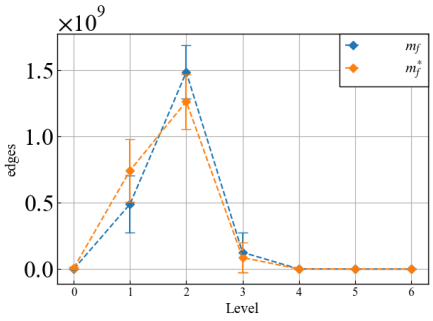


図 10: m_f と m_f^* の比較

グの比較である．縦軸は 64 回の BFS のなかで TopdownBFS と BottomupBFS が行われるパーセンテージを表す．m_f 近似は従来の BFS と同様に level 1 と level 2 の間にて Topdown BFS から Bottomup BFS への切り替えを行う．

図 10 は baseline 実装について，64 頂点から BFS を行ったときの各 level での m_f と m_f^* の平均値を表す. TopdownBFS から BottomupBFS への切り替えが起こる level 1，level 2 においては m_f と m_f^* の誤差は 50%程度であり，m_f 近似が m_f の値のオーダーを変えない良い近似となっていることを表す．

5.4 CoreBFS による頂点数と辺の数の変化

本節では CoreBFS によってどの程度探索すべき頂点数と辺の数が減少したかを確かめた．表 3 は CoreBFS を行わない場合のグラフの頂点数 (original vertex) と辺の数 (original edge) と，CoreBFS を行った場合，つまり GCC の 2-core 部分のグラフの

表 3: 2-core 分解前後の頂点数と辺の数の比較

original vertex	core vertex	original edge	core edge
67108863	24450436	2103844848	2087157496

頂点数 (core vertex) と辺の数 (original edge) を表す．Kronecker グラフにおいては探索すべき頂点数が約 3 分の 1 程度まで減少している．そのため，Bottomup BFS において頂点が未探索かどうかを調べるコストが CoreBFS によって減少し全体の高速化に寄与していると考えられる．

5.5 前処理時間

各アーキテクチャ毎に BFS 以外の前処理にかかる時間を計測した．前処理は通常の BFS では Kronecker グラフ生成，始点の選択，次数オーダリングである．CoreBFS では通常の BFS に加えて 3.1 節で説明した事前計算が含まれる．A6000 での前処理に要した時間は通常の BFS で 223 秒，CoreBFS で 592 秒である．M1 Max での前処理に要した時間は通常の BFS で 222 秒，CoreBFS で 285 秒である．どちらも現実的な時間で前処理が可能であるが，A6000 では M1 Max と比較して 2-core 分解処理に大幅に時間がかかる一方，2-core 分解処理以外の前処理時間は M1 Max と同程度である．この前処理時間の違いは前処理の実装方法の違いと各アーキテクチャーのシングルスレッド性能の違いに起因すると考えられる．A6000，M1 Max 共に前処理のうち 2-core 分解処理がシングルスレッドでの実行に，それ以外の部分はマルチスレッドでの実行となっている．シングルスレッド性能について A6000 は M1 Max に劣るためその差が 2-core 分解処理の所要時間に影響する．並列性能については A6000，M1 Max 共に十分並列化されているため 2-core 分解処理以外の前処理の所要時間の差に大きな差は生じない．

5.6 電力効率

実行時間のほかに電力効率も重要な性能指標の一つである．Graph500 にも電力あたりの TEPS (TEPS/W) を比較する Green ランキングが存在する [22] ことを踏まえ，同じ指標で提案手法の実装の電力効率を計測した．計測にはワットチェッカー RS-BTWATTCH2 を用いた．実験時の温度は摂氏 25.3 度，湿度は 31%であった．なお M1 Max はバッテリーを取り外せないノート型 PC に搭載されており，バッテリーからの給電によって実際の消費電力よりもワットチェッカーの測定値が小さくなってしまう可能性がある．今回の実験では計測開始前後でバッテリー残量に変化がなかったため，AC 電源に接続したワットチェッカーの数値をそのまま消費電力とみなすことにした．また，A6000 では他の実験と同様に頂点と辺の数がそれぞれ 2^{26} と 2^{30} の Kronecker グラフを用いたが，M1 Max ではそれぞれ 2^{27} と 2^{31} のグラフを用いた．実験結果を表 4 に示す．M1 Max の電力効率は A6000 と比べ 2.26 倍高かった．プロセッサの電力効率や 4 節で示した実装の差異が影響していると考えられるが，そのほかにもハードウェア上の違いが多く存在するため，今後より詳しく比較していく必要がある．実験で得た A6000 と M1 Max の性能はそれぞれ Graph500 の November 2022 Green

表 4: 電力効率

	消費電力	BFS 性能	電力効率
A6000	288 W	117.24 GTEPS	407.08 MTEPS/W
M1 Max	27.4 W	25.16 GTEPS	919.25 MTEPS/W

SMALL DATA ランキングで 25 位と 18 位を記録した [22]．特に M1 Max はディスクリート GPU や FPGA を用いない市販品のシステムとしては最も高い順位となった．

6 ま と め

本稿では BFS の高速化手法として (i) CoreBFS と (ii) m_f 近似を提案し，A6000 と M1 Max の 2 種のアーキテクチャに対し実装し性能評価を行った．CoreBFS は事前計算によって探索する頂点数を約 3 分の 1 程度まで削減する．m_f 近似は HybridBFS の切り替えパラメータを高精度に近似し不規則なメモリアクセスを省略する．提案手法のベースライン手法と比較した評価では NVIDIA RTX A6000 で 2.5 倍程度，Apple M1 Max で 1.4 倍程度の高速化を達成した．今後は Kronecker グラフ以外の実世界グラフに対しても提案手法が有効かどうか検証し手法の有用性を確かめることが課題である．

文 献

[1] J. Jordan, T. Ippen, M. Helias, I. Kitayama, M. Sato, J. Igarashi, M. Diesmann, and S. Kunkel, "Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers," *Frontiers in neuroinformatics*, p. 2, 2018.

[2] P. Bühlmann, P. Drineas, M. Kane, and M. van der Laan, *Handbook of big data*. CRC Press, 2016.

[3] S. Brohee and J. Van Helden, "Evaluation of clustering algorithms for protein-protein interaction networks," *BMC bioinformatics*, vol. 7, no. 1, pp. 1–19, 2006.

[4] Y.-F. Dai and X.-M. Zhao, "A survey on the computational approaches to identify drug targets in the postgenomic era," *BioMed research international*, vol. 2015, 2015.

[5] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, pp. 5–20, 2007.

[6] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," pp. 12:1–12:10, IEEE Computer Society Press, 2012.

[7] Y. Yasui, K. Fujisawa, and Y. Sato, "Fast and energy-efficient breadth-first search on a single numa system," 2014.

[8] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," pp. 22–31, 5 2016.

[9] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on gpus with unified memory," *Proc. VLDB Endow.*, vol. 13, pp. 1119–1133, 3 2020.

[10] K. Ueno, T. Suzumura, N. Maruyama, K. Fujisawa, and S. Matsuoka, "Extreme scale breadth-first search on supercomputers," pp. 1040–1047, 2016.

[11] J. Gao, W. Ji, Z. Tan, Y. Wang, and F. Shi, "Taichi: A hybrid compression format for binary sparse matrix-vector multiplication on gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 3732–3745, 2022.

[12] R. Dong, H. Cao, X. Ye, Y. Zhang, Q. Hao, and D. Fan, "Highly efficient and gpu-friendly implementation of bfs on single-node system," pp. 544–553, 2020.

[13] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," pp. 135–146, Association for Computing Machinery, 2013.

[14] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," pp. 456–471, ACM, 2013.

[15] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on gpus," pp. 68:1–68:12, ACM, 2015.

[16] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: Gpu graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, 8 2017.

[17] Johannes, S. D. Thilo, P. Konstantin, L. J. B. Luk, and Moe, "ipug: Accelerating breadth-first graph traversals using manycore graphcore ipus," pp. 291–309, Springer International Publishing, 2021.

[18] N. Masahiro, Koji, Ueno, , F. Katsuki, , K. Yuetsu, , and S. Mitsuhisa, "Performance of the supercomputer fugaku for breadth-first search in graph500 benchmark," pp. 372–390, Springer International Publishing, 2021.

[19] H. Lin, X. Tang, B. Yu, Y. Zhuo, W. Chen, J. Zhai, W. Yin, and W. Zheng, "Scalable graph traversal on sunway taihulight with ten million cores," pp. 635–645, 2017.

[20] X. Gan, Y. Zhang, R. Wang, T. Li, T. Xiao, R. Zeng, J. Liu, and K. Lu, "Tianhegraph: Customizing graph search for graph500 on tianhe supercomputer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 941–951, 2022.

[21] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on gpus: A survey," *ACM Comput. Surv.*, vol. 50, 1 2018.

[22] "Graph500 and Green Graph500." `https://graph500.org`.

[23] J. A. Ang, B. W. Barrett, K. B. Wheeler, and R. C. Murphy, "Introducing the graph 500.," 2010.

[24] AnandTech, "Apple's m1 pro, m1 max socs investigated: New performance and efficiency heights," 2021. `https://www.anandtech.com/show/17024/apple-m1-max-performance-review`.

[25] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310, 2003.

[26] Y. Lim, U. Kang, and C. Faloutsos, "Slashburn: Graph compression and mining beyond caveman communities," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, pp. 3077–3089, 2014.

[27] L. A. N. Amaral, A. Scala, M. Barthelemy, and H. E. Stanley, "Classes of small-world networks," *Proceedings of the national academy of sciences*, vol. 97, no. 21, pp. 11149–11152, 2000.

[28] "Dynamic parallelism in cuda." `https://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf`.

[29] D. Tödling, M. Winter, and M. Steinberger, "Breadth-first search on dynamic graphs using dynamic parallelism on the gpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2019.

[30] Apple, "Metal の概要." `https://developer.apple.com/jp/metal/`.

[31] R. Dong, H. Cao, X. Ye, Y. Zhang, Q. Hao, and D. Fan, "Highly efficient and gpu-friendly implementation of bfs on single-node system," in *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pp. 544–553, IEEE, 2020.