情報処理学会論文誌　コンピューティングシステム　Vol. 3　No. 2　57–66 (June 2010)

GPU上での高速なブロック化フロイド・ワーシャル法

奥山倫弘[†1]　伊野文彦[†1]　萩原兼一[†1]

本稿では，GPU（Graphics Processing Unit）を用いて全点対最短経路問題を高速に解くために，反復型ブロック化フロイド・ワーシャル法の高速化手法を提案する．提案手法は2種類あり，いずれもオフチップメモリの参照量を削減することで高速化を図る．1つ目の手法は，共有メモリの代わりにレジスタを優先的に用い，命令数を削減する．もう1つの手法は，オフチップメモリの参照量がブロックの大きさに反比例することに着目し，より大きなブロックを用いる．既存の再帰型ブロック化手法と比較した結果，頂点数が256〜1,024個の場合，両手法ともに4%以上高速であった．頂点数が多い場合，性能は1〜10%ほど下回った．GPUのピーク演算性能に対して70%の効率を達成した．

Fast Blocked Floyd-Warshall Algorithm on the GPU

Tomohiro Okuyama,[†1] Fumihiko Ino[†1]
and Kenichi Hagihara[†1]

This paper proposes an acceleration method for the iterative blocked Floyd-Warshall (BFW) algorithm, aiming at solving the all-pairs shortest path problem rapidly on the graphics processing unit (GPU). The proposed method contains two variations, both designed to reduce data access to off-chip memory. The first one also reduces the number of instructions by using registers rather than shared memory. The remaining one increases the block size because it is inversely proportional to the amount of off-chip memory access. For graphs with 256–1,024 vertices, both variations demonstrate 4% faster performance than a previous method that employs a recursive BFW algorithm. For larger graphs, on the other hand, our iterative method is 1–10% slower than the recursive method. The proposed method achieves approximately 70% of peak computational performance.

†1 大阪大学大学院情報科学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University

1. はじめに

全点対最短経路（APSP：All-Pairs Shortest Path）問題とは，有向グラフ G 上のすべての頂点間について最短経路を求める問題である．APSP問題の解は，各最短経路の経由頂点列を表す行列 P および各々の長さを表す距離行列 D からなる．この問題は，バイオインフォマティクスなどの様々な分野で応用されている．特に，D を高速に得ることが求められている．

APSP問題を解くアルゴリズムの1つとして，フロイド・ワーシャル（FW：Floyd Warshall）法[1),2)]がある．頂点数 n のグラフ G に対し，FW法は G を n 次正方行列（隣接行列）で表す．行列の要素ごとに $2n$ 回の算術演算および $4n$ 回のメモリ参照が必要であり，メモリ参照が性能ボトルネックとなる．そこで，GPU（Graphics Processing Unit）などのアクセラレータを用いた高速化が研究されている．

Katzら[3)]は，隣接行列をタイル状に分割し，タイルごとの処理を進める反復型ブロック化FW（BFW：Blocked FW）法をGPU上に実装している．彼らの実装は計算環境CUDA（Compute Unified Device Architecture）[4)]に基づき，行列 P および D を計算する．分割により参照の局所性が向上するため，小容量だが高速なオンチップメモリを用いてチップ外への低速な参照を削減できる．

反復型に対し，Buluçら[5)]は隣接行列を再帰的に分割する再帰型BFW法を提案している．彼らはBFW法が行列積と同様の参照パターンを持つことに着目し，Volkovら[6)]の行列積ルーチンを呼び出し高速化を図っている．距離行列 D のみを計算する彼らの手法は，Katzらの反復型よりも5倍ほど高速であり，ピーク演算性能の66%を引き出している．

このように，再帰型において効率の高い実装はある．しかし，反復型の効率を高めるための工夫は定かでない．また，GPUプログラム（カーネル）の呼び出し回数は，再帰型よりも反復型の方が少ない．したがって，GPU全体の同期回数も少なく，同期オーバヘッドの点では有利である．

そこで，本研究では反復型BFW法の高速化を目的として，CUDA互換のGPU上で動作する2種類の高速化手法を提案する．いずれの手法も距離行列 D を高速に計算する．そのために，レジスタや共有メモリなどのオンチップメモリを活用し，チップ外への参照を削減する．行列積応用手法は，Buluçら[5)]と同様に行列積ルーチン[6)]を呼び出して高速化を図る．この手法は，レジスタを優先的に用い，命令数の削減により効率の向上を図る．もう一方の2段階ブロック化手法は，2段階のブロック分割を用い，共有メモリ使用量を削減す

る．削減によりタイルの大きさ t を向上させることができ，t に反比例するチップ外への参照を削減できる．

2. 関連研究

我々の知る限り，Harish ら[7] は CUDA 互換の GPU 上に最初に FW 法を実装した．この実装は，GeForce 8800 GTX を用い，$n = 2{,}048$ の APSP 問題を約 70 秒で解く．しかし，オフチップメモリの参照量が多く，メモリ帯域幅が性能ボトルネックである．

Katz ら[3] は，Venkataraman ら[8] の CPU 向け手法を基に，GPU 上に反復型 BFW 法を実装している．彼らは GPU の共有メモリを手動キャッシュとして用い，高速化を果たしている．$n = 4{,}096$ のグラフに対し，13.7 秒で行列 P および D を計算している．

Buluç ら[5] の再帰型 BFW 法では，高速な行列積ルーチン[6] を応用して高速化を図っている．特に，再帰の浅い段階においてカーネル呼び出し 1 回あたりの計算量が多く，行列積ルーチンが高い効率を発揮する．$n = 4{,}096$ のグラフに対し，GeForce 8800 Ultra を用いて 1.01 秒で距離行列 D を計算していて，実効性能は 126.7 Gflop/s に達する．

行列積ルーチンを開発した Volkov ら[6] は，共有メモリをオペランドに指定した積和算が命令実行スループットを低下させることを指摘している．現在の命令セットでは，共有メモリの指定がたかだか 1 つのオペランドに限られているため，共有データどうしを操作するときに低下が顕著である．

GPU 以外のアクセラレータを用いる既存研究としては，Cell Broadband Engine による高速化手法[9] がある．この手法は $n = 7{,}360$ までのグラフを 50.6 Gflop/s で処理できる．FPGA（Field Programmable Gate Array）を用いる手法[10] は $n = 16{,}384$ のグラフを約 15 分（9.6 Gflop/s）で処理する．

最後に，Han ら[11] は反復型 BFW 法を CPU 向けに自動最適化している．彼らの手法は 2 段階のブロック化を採用し，キャッシュのレベルごとに適切なタイルサイズを選択する．本研究の 2 段階ブロック化手法は，この手法を基に共有メモリ使用量を削減する．

3. ブロック化フロイド・ワーシャル法

反復型 BFW 法[8] は隣接行列 M を $t \times t$ のタイルに分割し，各タイルの更新を反復する．行列要素 $M[u, v]$ はグラフ G 上の頂点 u および v 間の辺の重みを表す．u から v への辺が存在しない場合，$M[u, v] = \infty$ とする．FW 法を実行後，M は距離行列 D と等しく，要素 $M[u, v]$ は u から v への最短経路長を表す．

```
 1: IterativeBFW(M, n, t)
 2:   for K := 1 to n/t
 3:     FWI(T_{K,K}, T_{K,K}, T_{K,K}, t)     // 軸タイル
 4:     for I := 1 to n/t, I != K
 5:       FWI(T_{I,K}, T_{I,K}, T_{K,K}, t)   // 軸列タイル
 6:     for J := 1 to n/t, J != K
 7:       FWI(T_{K,J}, T_{K,K}, T_{K,J}, t)   // 軸行タイル
 8:     for I := 1 to n/t, I != K
 9:       for J := 1 to n/t, J != K
10:         FWI(T_{I,J}, T_{I,K}, T_{K,J}, t) // 非軸タイル
11: FWI(A, B, C, t)
12:   for k := 1 to t
13:     for i := 1 to t
14:       for j := 1 to t
15:         A[i,j] := min(A[i,j], B[i,k] + C[k,j])
```

図 1　反復型 BFW 法のアルゴリズム
Fig. 1　Blocked iterative FW algorithm.

図 1 に反復型 BFW 法のアルゴリズムを，**図 2** にタイルの更新順序を示す．ここで，M 中の各タイルを $T_{I,J}$ で表し，左上のタイルを $T_{1,1}$ とする．図 2 の黒色のタイル $T_{K,K}$ を軸タイルと呼ぶ．灰色で示した第 K タイル行および列を，それぞれ軸タイル行および軸タイル列と呼び，これらのタイルはそれぞれ軸行タイルおよび軸列タイルと呼ぶ．上記以外の白色のタイルは非軸タイルと呼ぶ．

まず，左上のタイル $T_{1,1}$ を軸タイルとし，このタイルを更新する．次に，軸タイル行および列を更新し，最後にすべての非軸タイルを更新する．これを軸タイル $T_{K,K}$ を左上から右下まで変えながら反復する．

各タイルは図 1 の手続き FWI を用いて更新される．引数 A が更新対象のタイルであり，B および C は更新中に参照するタイルである．軸タイル，軸行および列タイルの更新では，これらの一部は重複し，実際に参照するタイル数はそれぞれ 1 つおよび 2 つである．軸タイルの更新時はすべての引数が軸タイルを指し，軸タイルに通常の FW 法を適用することに等しい．軸行および列タイルでは，B または C の一方が A と同一であり，もう一方は軸タ

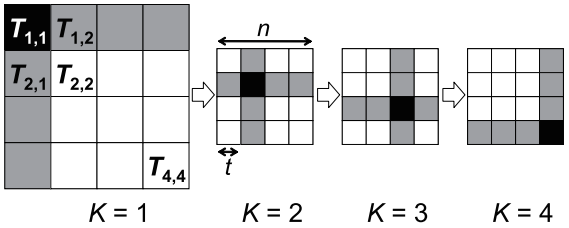

図 2　反復型 BFW 法におけるタイルの更新順序（$n/t = 4$）
黒，灰および白色のタイルはそれぞれ軸タイル，軸行・軸列タイルおよび非軸タイルを表す

Fig. 2　Tile updating process of the blocked iterative FW algorithm ($n/t = 4$).
A black tile represents a pivot tile. Gray tiles are pivot row and pivot column tiles while white tiles are non-pivot tiles.

イルである．また，非軸タイルの更新時は，B および C はそれぞれ軸列および行タイルである．第 I タイル行中のすべての非軸タイルは，同じ行中の軸列タイル $T_{I,K}$ を共通に参照する．タイル列方向についても同様である．これらのタイル間の参照の依存関係により，軸タイル，軸行および列タイル，非軸タイルの順に更新する．一方，軸行および列タイル間，非軸タイル間には依存関係がなく，これらに含まれる複数のタイルを同時に更新できる．

FWI の 1 回の呼び出しあたり，B および C の要素はそれぞれ t 回参照され，A の各要素は t 回更新される．そのため，FWI あたり $O(t^3)$，BFW 法全体では $O(n^3)$ 個の要素に対するメモリアクセスが生じる．一方，高速なオンチップメモリにタイル A，B および C を格納する場合，オフチップメモリへのアクセス要素数は約 $1/t$ になり，BFW 法のアクセスデータ量は $N_b = d(4n^3/t - 2n^2)$ である．ここで d は行列要素のデータ量である．また，文献 9) によると，反復型 BFW 法は延べ $n(n^2 - 2t + 1)$ 要素を更新する．要素の 1 回の更新あたり加算と min() の 2 回の算術演算を用い，反復型 BFW 法は $N_c = 2n(n^2 - 2t + 1)$ 回の算術演算を行う．ここで，min() は 2 つの引数をとり，それらのうち最小値を返す関数とする．

4. 提案手法

2 段階ブロック化手法および行列積応用手法は，非軸タイルの更新方法を除けば，共通の処理を持つ．そこで，まず両者に共通する設計を示したのち，手法ごとの差分について述べる．

本稿では，CUDA [4] におけるストリームプロセッサおよびマルチプロセッサをそれぞれ SP および MP と表記する．また，GPU 上でカーネルを実行する CUDA スレッドをスレッドと呼び，スレッドブロックを TB，グローバルメモリを GM と表記する．

4.1 共通設計

隣接行列 M は n^2 の大きなメモリ領域を占め，計算中に上書きされる．そこで，M は GM に格納する．メインメモリ上のグラフデータを GM 上の M へ転送したのち，GPU 上で BFW 法を実行する．最後に，計算結果をメインメモリへ転送し終了する．

GPU での計算には，それぞれ軸タイル，軸タイル行・列および全非軸タイルの更新を担当する 3 種のカーネルを用いる．これらのカーネルは，反復型 BFW 法が持つ 2 段階の並列性を用いる．まず，図 1 の FWI 中の 13 および 14 行目の i および j のループを 1 つの TB 内のスレッドを用いて並列化し，複数の要素を同時に更新する．より粗い並列化として，軸タイル行・列の更新では，タイル数に応じた TB を用いて 4 および 6 行目のループを並列化し，すべての軸行および列タイルを並列に更新する．同様に，すべての非軸タイルも並列に更新する．ただし，データの一貫性を保つため，軸タイル，すべての軸行・列タイルおよびすべての非軸タイルを更新するたびに GPU でのカーネル実行を終了し GPU 全体を同期する．

前述のとおり，t が大きいほどオフチップメモリである GM へのアクセス量 N_b を削減できる．しかし，t を大きくするとオンチップメモリの使用量が増加し，並列処理の効率が低下しうる．GPU では，スレッドおよび TB がオンチップメモリであるレジスタおよび共有メモリを分け合って動作するためである．そこで，GPU の高い演算性能を使いきるために，t を大きくしてメモリ帯域幅が理論的に性能ボトルネックとなることを避ける．そのうえで，処理効率の低下を避けるためにできるだけ小さい t を用いる．

図 3 に計算に対するメモリ参照の要求比率 N_b/N_c および 3 種の GPU の演算性能 c に対する GM 帯域幅 b の比率 b/c を示す．ここで，$c = pz$ であり，p および z はそれぞれ SP 数および SP の動作周波数である．N_b および N_c は 3 章に示した式に基づき，$n = 8{,}192$ および行列要素のデータ量 $d = 4$ バイトとして計算した．図 3 より，$t \geq 32$ では 3 種の GPU すべてにおいて $N_b/N_c < b/c$ であり，演算性能を使いきったとしても，計算時間が GM アクセス時間よりも長く，アルゴリズム上は GM 帯域幅が性能ボトルネックとならない．そこで，$t = 32$ とし，各カーネルはスレッドが複数の要素を更新するように設計する．1 スレッドが 1 要素のみ更新する場合，TB 内のスレッド数は t^2 であるが，これは最大 512 に制限されており [4]，この場合は $t \leq 22$ に制限されるためである．

軸タイルの更新カーネルでは，更新対象のタイルが 1 つであるため，1 つの TB のみを

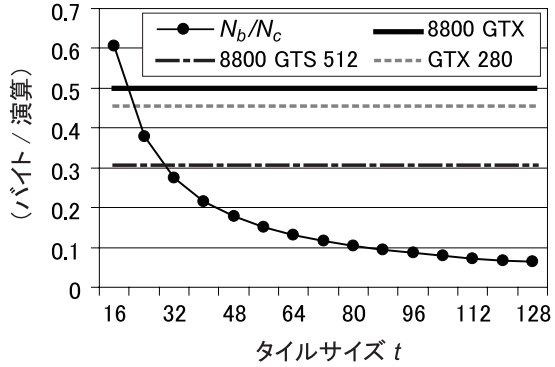

図 3　反復型 BFW 法の計算に対するメモリ参照の要求比率 N_b/N_c および GPU の演算性能に対する GM 帯域幅の比率

Fig. 3　Ratio N_b/N_c of memory access to computation demanded by the blocked iterative FW algorithm and that of global memory bandwidth to computational performance of the GPU.

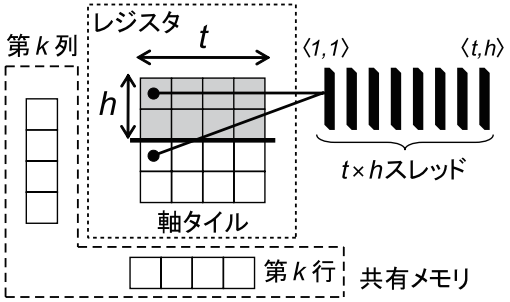

図 4　軸タイル更新時におけるオンチップメモリの使用

Fig. 4　Using on-chip memory for updating the pivot tile.

GPU 上で実行する．TB 内のスレッド数は $t \times h$ とし，軸タイルを h 行ずつ t/h 個の領域に分割し処理する（**図 4**）．スレッドは，2 次元のスレッド ID $\langle i, j\rangle$ を持ち，FWI におけるタイル A 内の t/h 個の要素 $A[i, j + hj']$，$0 \le j' < t/h$ を更新する．FWI 中の k についてのループにはデータ依存があるため，タイル中の全要素を更新するたびに全スレッドを同期させる．

タイル中の各要素は，更新を担当するスレッドのレジスタに格納する．実際のプログラムにおいてはこれら t/h 個の要素を配列に格納し，for ループにより処理対象の要素を逐次切り替えるように記述した．そのうえで，unroll プラグマ[4] によりこのループをコンパイラに展開させている．後述の実験環境においては $t = 64$，$h = 1$ および $t = 64$，$h = 2$ のときを除き，ローカルメモリ[4] を用いない効率的なコードが生成されている．

k 回目の更新時，第 k 行および第 k 列の要素はタイル内の全要素から参照される．そこで，これらの要素は k 回目の更新前に共有メモリへコピーする．このために，共有メモリアクセスが余分に生じるが，共有メモリ使用量は $2t$ であり，より大きな t にも使用できる．

軸タイル行および列の更新では，これらに含まれる $2(n/t - 1)$ 個のタイルを並列に更新する．タイルごとに 1 つの TB を割り当て，軸タイルと同様にそれぞれ th 個のスレッドを用い，h 行ずつ並列に更新する．オンチップメモリには更新対象のタイルと軸タイルを配置する．軸タイルの更新時と同じく，更新対象のタイルはレジスタへ格納し，別スレッドから参照される第 k 列もしくは第 k 行の要素は共有メモリにコピーする．軸タイルは第 k 列または第 k 行のみを共有メモリへ配置し，計算の進行に応じて GM から値を読み出す．したがって，軸タイルの更新と同じく，各 TB は共有メモリを $2t$ 使用する．

非軸タイルの更新では，前述の 2 種のタイルと異なり，更新要素間にデータ依存がなく，行列積計算などと同様にループ順序を自由に入れ替えられる．この特徴により，行列積応用手法では非軸タイルとそれ以外のタイルのカーネルが用いる分割形状を切り替え，BFW 法のタイル形状と異なる分割を用いた行列積ルーチンを使用する．2 段階ブロック化手法では，タイルをさらにブロック化し，FWI 中の k についてのループを分割することにより同時に参照される要素を削減する．

4.2　行列積応用手法

図 1 中の非軸タイル更新部分をみると，軸タイル列に軸タイル行を掛ける行列積と同様のアクセスパターンを持つ．要素の更新は，行列積計算おける乗算を加算に，加算を min() に置き換えることにより計算できる．そこで，非軸タイルの更新に Volkov ら[6] の行列積ルーチンを応用する．**図 5** の太線で示すように，このルーチンは隣接行列 M を 16×64 の小行列に分割し，小行列ごとに TB を割り当てる．小行列 $T'_{I',J'}$ を担当する TB は，軸タイル列中の $16 \times t$ の領域 $B'_{I'}$ および軸タイル行中の $t \times 64$ の領域 $C'_{J'}$ を参照し，$T'_{I',J'}$ を計算する．

このルーチンは，各スレッドが小行列中の 1 列中のすべての要素を更新する．更新対象の小行列および $C'_{J'}$ をレジスタに，$B'_{I'}$ を共有メモリに配置する．$C'_{J'}$ をレジスタへ配置できるのは，各スレッドが担当する列方向の 16 要素のみが $C'_{J'}$ 上の共通した領域を参照するた

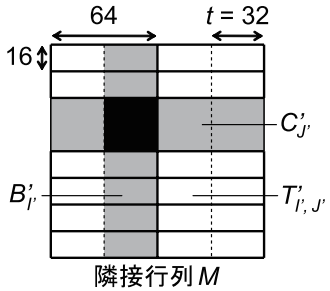

図 5　行列積応用手法における計算領域の分割
Fig. 5　Partitioning of the computational region in the matrix multiplication based method.

めである．したがって，図 1 の 15 行目の要素の更新式は，共有メモリ上の値を 1 回のみ参照する．したがって，$B'_{I'}$ および $C'_{J'}$ をともに共有メモリへ配置する場合に生じる，共有メモリからレジスタへデータを移動する余分な命令がなく，共有メモリ参照による演算効率の低下を回避できる．

また，隣接行列 M は C 言語で一般的に用いられる row-major レイアウトで GM に格納する．この格納方式は，行方向の要素を連続領域に配置する．反復型 BFW 法では，タイル単位の参照が多いため，タイルごとに要素を格納する Blocked Data Layout（BDL）を用いて参照の局所性を高められる．しかし，本手法では行列積ルーチンおよび BFW 法が異なるブロック分割を用いるため，BDL の使用は参照する要素のアドレス計算を複雑にする．一方，row-major レイアウトは特定のタイルサイズに依存せず，アドレス計算が BDL を用いる場合に比べて容易である．

反復型 BFW 法向けに行列積ルーチンに対し 3 点の変更を行う．1 点目は，非軸タイルの更新時は軸行・列タイルは更新が不要なため，これらのタイル部分は計算を行わない．2 点目は $B'_{I'}$ 全体を共有メモリへコピーし，TB 内で必須な同期を 1 回に削減する．3 点目は，小行列あたりの計算量は小さいため，演算効率を向上させるために各 TB が複数の小行列を計算する．

1 点目の計算の省略について，既存の行列積ルーチンをそのまま用いると，非軸タイルだけでなく，軸タイル行および列中の要素も更新される．一方，図 1 の 8 および 9 行目において，ループの制御変数 I および J にはそれぞれ I != K および J != K という条件がある．この 2 つの条件は，軸行および列タイルを更新しないことを表している．そこで，軸タイル行および列部分を計算しないように行列積ルーチンを変更する．軸タイル行部分はその計算を担当する TB を生成しないことにより，計算を行わない．一方，各 TB が列方向の複数のタイルを担当するため，軸タイル列部分はスレッド内で計算が必要かを判定し，計算結果を GM へ書き戻さない．更新計算自体を行わないこともできるが，結果の書き込みのみを省く方がわずかに高速であった．

2 点目については，Volkov ら[6] の実装は，$B'_{I'}$ のうち 16^2 要素のみを共有メモリに格納し，適宜共有メモリの要素を入れ替える．入れ替えるたびに，データの一貫性を保つために TB 内を同期する必要がある．本手法では，$B'_{I'}$ の大きさは $16t$ であり，全体を共有メモリに格納できる．そこで，カーネルの最初に $B'_{I'}$ のすべての要素を共有メモリへコピーし，これらの同期を削減する．

3 点目については，各 TB が行方向に連続する l 個の小行列を計算する．これらの小行列は $B'_{I'}$ を共通して参照するため，GM から共有メモリへ $B'_{I'}$ をコピーする際に消費される帯域と実行命令数を $1/l$ に削減できる．ただし，l を大きくすると TB 数が減少し，性能が低下する．現在は実験的に l を決めている．

なお，この行列積ルーチンは分割形状が固定されているため，t を大きくすることによる GM へのアクセスの削減は 2 段階ブロック化手法よりも小さい．Volkov ら[6] によると，この行列積は 1 回の実行につき軸タイル行および列を延べ $n^2t(1/64+1/16)$ 要素分参照する．したがって，更新対象である非軸タイルへのアクセスも含めた場合，すべての非軸タイルの 1 回の更新あたり，GM 上の延べ $5n^2t/64+2(n-t)^2$ 個の要素にアクセスする．これは，$n \gg t$ とすると $t \simeq 26$ の 2 段階ブロック化手法の削減率に相当する．つまり，$t > 26$ では 2 段階ブロック化手法よりも GM へのアクセスが多い．

4.3　2 段階ブロック化手法

2 段階ブロック化手法では，非軸タイルをさらに小タイルにブロック分割し，共有メモリの使用量を削減する．2 段階目のブロック分割は，行列積などで一般的に使われる単純な 2 次元タイル状の分割方法を用いる．ここで，タイル内を大きさ t' の複数の小タイルに分割するが，TB 内における並列処理の効率低下を回避するため，$t \times t'$ スレッドを用い，行方向の小タイルを並列に更新する．FWI のタイル B は t' 列分，C は t' 行分を共有メモリへ格納し，計算の進行にともなって GM から値をコピーする．共有メモリ上では B を転置し，バンクコンフリクト[4] を避ける．また，t' 列分を共有メモリへコピーするため，$t' < 16$ のとき B を GM から coalesced 参照[4] できない．そこで，軸タイル行分のメモリ領域を新たに確保し，軸タイル行をテクスチャメモリ[4] に配置する．隣接行列は BDL を用いて GM に格納し，軸タイルおよび軸タイル行・列の更新カーネルも BDL に対応させる．また，行

```
 1: RecursiveBFW(M, n, t_R)
 2:   if n ≤ t_R return FW(M) // M に FW 法を施す
 3:   M_{1,1} := RecursiveBFW(M_{1,1}, n/2, t_R)
 4:   M_{1,2} := M_{1,1} · M_{1,2}
 5:   M_{2,1} := M_{2,1} · M_{1,1}
 6:   M_{2,2} := M_{2,2} + M_{2,1} · M_{1,2}
 7:   M_{2,2} := RecursiveBFW(M_{2,2}, n/2, t_R)
 8:   M_{2,1} := M_{2,2} · M_{2,1}
 9:   M_{1,2} := M_{1,2} · M_{2,2}
10:   M_{1,1} := M_{1,1} + M_{1,2} · M_{2,1}
```

図 6　再帰型 BFW 法のアルゴリズム
Fig. 6　Blocked recursive FW algorithm.

列積応用手法と同じく，各 TB が行方向に連続した l 個のタイルを逐次処理する．しかし，B はその一部のみ共有メモリに配置するため，GM から B をコピーする回数は削減できず，一部のアドレス計算結果のみが再利用される．

この手法では，図 1 の 15 行目の更新式は 3 命令になる．しかし，各スレッドが t/t' 個の要素を更新するため，最初の要素を更新する際に共有メモリからレジスタへ移動した C の値が再利用され，他の要素は 2 命令で更新される．したがって，演算効率は $2/(2+t'/t)$ 以下となる．t' を小さくするほど効率は高くなるが，レジスタ使用量が増加し性能が低下しうる．

4.4　カーネル実行の回数

反復型 BFW 法は，軸タイル，軸タイル行・列，および非軸タイルを更新する 3 種のカーネルを，軸タイルの位置を移しながら n/t 回実行する．ゆえに，$3n/t$ 回のカーネル実行が行われる．

一方，再帰型 BFW 法[5] では，隣接行列 M をそれぞれ大きさ $n/2$ の 4 つの部分行列に 2 次元タイル状に分割し，このうち左上および右下の部分行列を再帰的に計算する．**図 6** にアルゴリズムを示す．図 6 中で，2 行列 A および B の積 $A \cdot B$ は通常の行列積計算における要素間の乗算を加算に，加算を min() に置き換えた計算を表す．また，$A + B$ は 2 行列の要素ごとに min() を施すことを表す．

GPU は関数を再帰的に呼び出せないため，CPU が RecursiveBFW() の再帰呼び出しを担当し，隣接行列を論理的に分割する（図 6 の 3 および 7 行目）．部分行列の範囲を指定してカーネルを実行し，GPU 上で隣接行列中の各部分行列を更新する（図 6 の 4〜6 および 8〜10 行目）．このとき，データ依存を壊さないように，5，6，9 および 10 行目の更新後に GPU 全体を同期させるため，再帰のたびにカーネルを最低 4 回実行する．再帰呼び出しの深さは，再帰のたびに行列の大きさが半分になるため，$\log_2 n/t_R$ である．ここで，t_R は再帰を停止する分割サイズとする．再帰型 BFW 法では，GM 参照量は $O(n^3/t_R)$ である．大きさが t_R の部分行列は通常の FW 法を用いて更新する．1 回のカーネル実行によりこの FW 法を行うとすれば，このカーネルは再帰型の計算全体で n/t_R 回実行される．したがって，再帰型 BFW 法全体では $4\sum_{i=0}^{\log_2 n/t_R}(2^i) + n/t_R = 5n/t_R - 4$ 回カーネルが実行される．なお，文献 5) の実装は，4 および 5 行目と，8 および 9 行目で別々にカーネルを実行するため，全体で $7n/t_R - 6$ 回カーネルを実行する．

つまり，再帰型におけるカーネル実行回数は反復型に比べ，$4n/t - 6$ もしくは $2n/t - 4$ 回多い．GPU での計算時間が短く，カーネルの呼び出しにともなう同期オーバヘッドが相対的に大きくなる場合，反復型は再帰型よりも高い性能を発揮すると考えられる．

5. 評 価 実 験

実験には 2.93 GHz 駆動の Intel Core i7 940，12 GB のメインメモリ，GPU は nVIDIA GeForce GTX 280 を持つ計算機を用いた．OS は Windows XP Professional x64 Edition，ビデオドライバのバージョンは 190.38 であり，CUDA 2.3 を用いた．用いた GPU の GM 容量は 1 GB であるため，$n > 16{,}384$ を扱えない．本章では，行列積応用手法を手法 A，2 段階ブロック化手法を手法 B と呼ぶ．

表 1 に文献 12) を用いて生成したランダムグラフに対する計算時間を示す．隣接行列を GM 上に配置後から計算結果を GM に格納するまでの計算時間を計測した．各グラフは，頂点数 n，辺の数 $4n$ である．手法 A および B のパラメータは予備実験の結果，性能が高かったものを用いた．**表 2** に非軸タイル更新カーネルのパラメータおよび Occupancy[4] を示す．表 2 の共有メモリは TB あたりの使用量をバイト単位で示し，レジスタ数はスレッドあたりの数である．TB 中のスレッド数は手法 A においては 64 に固定されている．手法 B においては $t' \in \{1, 2, 4, 8, 16\}$，かつ $t' \leq 512/t$ の範囲から t' を t ごとに選択しており，TB 中のスレッド数は tt' である．手法 A および B ともに，共有メモリ使用量およびレジスタ使用量は n にかかわらず一定である（表 2）．l は TB が処理する小行列もしくはタイル数である．$l = 2^i$ とし，$0 \leq i \leq 7$，かつ手法 A においては $l \leq n/64$，手法 B においては

表 1　ランダムグラフ[12] での計算時間（単位：ミリ秒）
Table 1　Timing results for random graphs [12] with a different number n of vertices. Results are presented in milliseconds.

頂点数 n	反復型						再帰型[5]		
	手法 A			手法 B			$t_R=16$	$t_R=32$	$t_R=64$
	$t=16$	$t=32$	$t=64$	$t=16$	$t=32$	$t=64$			
128	0.427	0.404	0.639	0.702	0.567	0.791	0.783	0.561	0.533
256	0.896	0.819	1.295	1.570	1.196	1.606	2.028	1.585	1.498
512	3.047	2.558	3.675	4.629	3.394	3.928	5.645	4.738	4.415
1,024	15.62	13.82	15.15	20.11	15.88	15.68	19.03	17.21	16.58
2,048	93.93	87.42	100.2	120.5	96.32	99.69	95.64	91.97	91.97
4,096	663.9	630.3	710.3	862.7	699.9	706.9	639.0	631.5	631.5
8,192	5,135	4,869	5,472	6,676	5,353	5,434	4,883	4,866	4,867

表 2　非軸タイルを更新するカーネルのパラメータ
Table 2　Kernel parameters for updating non-pivot tiles.

	手法 A			手法 B		
t	16	32	64	16	32	64
t'	—	—	—	4	2	4
TB 中のスレッド数	64	64	64	64	64	256
共有メモリ	1,140	2,228	4,404	572	572	2,108
レジスタ数	39	40	42	18	30	30
l（$n=8{,}192$）	16	128	8	16	16	1
TB 数（$n=8{,}192$）	4,088	510	8,128	16,352	4,080	16,256
Occupancy [4]	38%	38%	19%	50%	50%	50%

$l \le n/t$ の範囲から計算時間が最も短くなる l を n および t ごとに選択した．表 2 では l および l に依存する TB 数を $n=8{,}192$ の場合のみ示す．手法 A では n および t を固定した場合，予備実験において性能が高くなるパラメータには再現性があった．l のみを変えたとき，手法 A においては非軸タイルの更新に要する時間が最大で 1.5 倍長くなる．また，手法 B においては最大で 2.2 倍長くなる．手法 B において t' のみを変えた場合，非軸タイルの更新に要する時間が最大約 2 倍長くなる．なお，反復型の両手法ともに要素のデータ型は int 型であり，計算結果は CPU 版と一致している．また，float 型では，手法 A（$t=32$）の計算時間は int 型と比べ最大 0.14%増減し，手法 B（$t=32$）では float 型の方が 0.04〜4.5%遅い．再帰型（$t_R=16$）[5] は $t_R \le 22$ を前提に逐次の FW 法を実行するカーネルを設計しており，t_R を単純に大きくできない．そこで，$t_R=32$ および 64 の実装では，このカーネルを 4.1 節で述べた軸タイルを更新するカーネルに置き換え，計算時間を測定した．

表 1 より，手法 A（$t=32$）は $n \le 1{,}024$ のとき，再帰型（$t_R=32$）と比べ 20%以上高速である．また，再帰型（$t_R=64$）と比べると，$n=128$ のときの手法 B を除き，$n \le 1{,}024$ では $t=32$ の手法 A および B は 4%以上速い．n が大きくなると性能差はなくなるが，$n=8{,}192$ においても再帰型に対する手法 A（$t=32$）の計算時間の増加は約 3 ミリ秒であり，再帰型とほぼ同等の性能を発揮する．

再帰型では，t_R が大きいほど計算時間が短縮される．しかし，その効果は n が大きいほど小さく，$n \ge 1{,}024$ のとき $t_R=32$ および 64 の差はわずかである．再帰型のカーネル実行回数は t_R に反比例する．そのため，n が小さい場合は t_R が大きいほどカーネルの呼び出しにともなう同期オーバヘッドが少なく，高速である．再帰型（$t_R=32$）は，$n=8{,}192$ のときカーネルを 1,786 回実行する．そのうちの 90%は 0.1 ミリ秒以下の短いカーネル実行である．一方，再帰の最も浅い段階における 6 回のカーネル実行は，それぞれ約 590 ミリ秒である．このように，n が大きくなると，再帰の浅い段階におけるカーネル実行が計算時間の大半を占め，カーネルの呼び出しにともなう同期オーバヘッドは相対的に小さくなる．加えて，再帰の浅い段階におけるカーネルの計算量は t_R に依存しない．ゆえに，t_R による計算時間の変動は n が大きい場合に小さくなる．

以降は反復型手法の性能解析について述べる．まず，表 1 において $t=32$ および 16 を比べると，手法 A および B ともに，$t=32$ の計算時間が短い．特に手法 B においてその短縮率が 19〜27%と高い．図 3 より，GeForce GTX 280 を用いる場合，$t=32$ のときは GPU の演算性能を使いきることができるが，$t=16$ のときは GM 帯域幅が性能ボトルネックとなり演算性能を使いきることができない．$t=16$ のときは $N_b/N_c=0.61$ より，$t=16$ において GM 帯域幅 b を使いきるときの演算性能 $c_{t=16}$ は $b/c_{t=16}=0.61$ である．GeForce GTX 280 のピーク演算性能 c に対する b の比率は $b/c=0.46$ である（図 3）．したがって，$t=32$ のときは $t=16$ のときに比べ，GPU の演算性能が見かけ上 $c/c_{t=16}=1.33$ 倍になり，計算時間は $1/1.33=0.75$ 倍になる．つまり，$t=32$ のときは $t=16$ のときと比較して，計算時間は理論上 25%短くなる．表 1 の実測値より，$n=8{,}192$ のとき手法 B（$t=32$）の計算時間は $t=16$ に対して 19.8%短縮されており，理論上の短縮率に近く，t を大きくすることによる性能の向上が確認できる．一方，$n=512$ では手法 B の計算時間は実測値で 25%以上短縮している．n が小さい場合は GPU 上で並列処理されるスレッド数が少なく，GPU の性能を使いきっていないと考えられる．加えて，カーネルの実行時間が短く，カーネルの呼び出しにともなう同期オーバヘッドが相対的に大きい．そのため，理論上の短縮

表 3　$n = 8{,}192$ のグラフを処理する際の演算効率（単位：%）
Table 3　Computational efficiency for a graph with $n = 8{,}192$.

	反復型						再帰型		
	手法 A			手法 B					
	$t = 16$	$t = 32$	$t = 64$	$t = 16$	$t = 32$	$t = 64$	$t_R = 16$	$t_R = 32$	$t_R = 64$
効率	68.8	72.7	64.6	53.0	65.9	64.9	72.3	72.7	72.7

率を上回ったと考えられる．一方，手法 A では手法 B よりも $t = 16$ のとき GM へのアクセス量が少ないが，$t = 32$ では逆に多い．そのため，手法 A では計算時間の短縮率は手法 B よりも低い．手法 A（$t = 16$）に対する手法 A（$t = 32$）の計算時間の短縮率は理論上 7.5%であり，実際に計算時間の短縮も手法 B より小さい．

また，**表 3** に $n = 8{,}192$ のときの演算効率を示す．効率を算出するにあたり，反復型および再帰型 BFW 法の算術演算回数は FW 法の $2n(n-1)^2$ 回に統一し GPU のピーク演算性能は 311 Gop/s とした．Gop/s は 1 秒間に 10^9 回演算することを示す．なお，用いた GPU は 1 クロックあたり float 型の積和算 1 回および乗算 1 回を実行できるため，Flop/s の公称値は 933 Gflop/s である．手法 A（$t = 32$）および再帰型は $n = 8{,}192$ のとき 70%以上の高い演算効率を達成する．

表 4 は手法 A および B のタイルの種類ごとの計算時間を示す．計測のための同期オーバヘッドにより，これらの和は表 1 に示す計算時間よりもやや長い．

$t = 64$ では，反復型の両手法ともに $t = 32$ の場合と比べ計算時間が長い．手法 A は 603 ミリ秒長く，表 4 より，この増分のうち約 90%を非軸タイルにおける計算時間が占めている．増加の主な原因は，$t = 64$ のときの Occupancy が $t = 32$ のときと比べて半減するためである（表 2）．$t = 64$ のとき，非軸タイルを更新するカーネルでは各 TB が共有メモリを 4 KB 以上消費する．これにより，MP が同時実行できる TB 数が，$t = 32$ の場合よりも少ない 3 つとなり，Occupancy が低下し計算時間が長くなる．また，両手法とも軸タイル行および列の計算時間が $n = 8{,}192$ のとき，$t = 32$ に比べ $t = 64$ では 40〜60 ミリ秒長い．$t = 64$ のとき，軸タイル行および列を更新するカーネルのコンパイル時に生成される命令が多く，コンパイラがカーネル中の一部のループを展開しない．そのため，このカーネルでは $t = 32$ の場合よりもループにともなうオーバヘッドが大きい．また，手法 B（$t = 64$）の非軸タイルの計算時間は $t = 32$ の場合に対して 30 ミリ秒長い．したがって，軸タイル行および列更新時のオーバヘッドを除いても，t を 32 から 64 に大きくした場合の性能向上は得られなかったといえる．

表 4　$n = 8{,}192$ のグラフを処理するときの反復型手法の計算時間内訳（単位：ミリ秒）
Table 4　Breakdown timings of the blocked iterative FW method in milliseconds. The number n of vertices is $n = 8{,}192$.

	手法 A			手法 B		
t	16	32	64	16	32	64
軸タイル	10.01	8.520	16.58	17.97	12.35	18.51
軸タイル行・列	70.64	74.21	132.5	121.7	85.8	128.1
軸タイル	5,072	4,795	5,328	6,553	5,262	5,292

表 5　非軸タイル計算における命令スループット（$n = 8{,}192$）
Table 5　Instruction throughput of updating non-pivot tiles. The number n of vertices is $n = 8{,}192$.

	手法 A			手法 B		
t	16	32	64	16	32	64
命令スループット	0.855	0.850	0.746	0.974	0.876	0.822

反復型の 2 手法を比較すると，表 1 より手法 A は手法 B よりも $t = 32$ のとき 9.0〜32%計算時間が短い．$t = 32$ のとき，両手法ともに $n = 8{,}192$ では非軸タイルの計算時間が全体の 98%を占める．そこで，非軸タイルを更新するカーネルの命令数を調べ，命令実行の効率が良いかを検討した．

表 5 に $n = 8{,}192$ のグラフを処理する場合の命令スループットを示す．ここで，命令スループットとは，各 SP が 1 クロックあたりに実行した命令数の平均値であり，値が大きいほど命令実行の効率が良い．計測には CUDA Visual Profiler [4] を用いた．表 5 より，$t = 32$ の場合は手法 A および B の命令スループットはほぼ同じである．手法 A において命令スループットが 1 に達しない原因として，メモリアクセス待ちが生じている可能性がある．手法 A の非軸タイルでは，TB は最初に GM から共有メモリへ $16t$ 個の要素をコピーする．この粒度の大きなメモリ参照が実行ワープの切替えによる計算とメモリ参照のオーバラップを妨げ，メモリアクセス待ちが生じうる．一方，手法 B では共有メモリ上の要素を入れ替えるたびに TB 内を同期するため，この同期がスループットを低下させている可能性がある．

図 7 に $n = 8{,}192$ のグラフにおいて，非軸タイルを更新するカーネルで実行される総命令数を示す．カーネル中の命令数はコンパイル後の cubin 形式のファイルを decuda [13] を用いてディスアセンブルし，その出力から算出した．算出した命令数および CUDA Visual Profiler により計測した実行命令数の差は 1%以下であり，算出した命令数はほぼ正しいと考えられる．

アルゴリズム上必要な命令（必須命令）は，手法 A において全命令の 87%，手法 B（$t' = 2$）

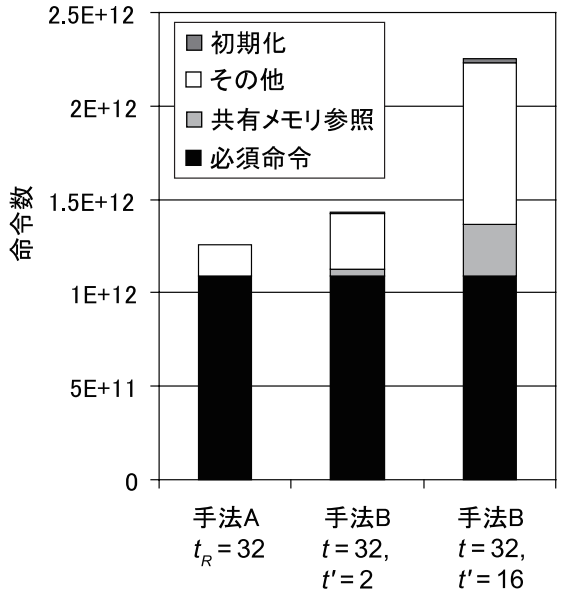

図 7　非軸タイル更新カーネルの命令内訳（$n = 8{,}192$）

Fig. 7　Breakdown of the number of instructions in the kernel for updating non-pivot tiles. The number n of vertices is $n = 8{,}192$.

では 76%を占める．つまり，手法 A は実効演算性能に寄与しない命令がより少なく，より最適化されている．図 7 中のその他は，更新時に参照する要素のアドレスを再計算する命令などである．手法 B では，このうち共有メモリからレジスタへの転送命令数を別けて示す．$t' = 2$ のとき，手法 B ではこの参照命令が占める割合は 2.4%であり，共有メモリの使用による命令数の増加は小さい．$t' = 2$ の場合，レジスタに移動された値がスレッド中で 16 回再利用されるためである．t' を大きくすると，この再利用回数は減少する．$t' = 16$ の場合，共有メモリ参照が全命令中の 12%を占める．加えて，アドレスの再計算命令も増加するため，必須命令の比率が 48%に低下し，$t' = 2$ よりも非軸タイルの計算時間は 33%長い．共有メモリ参照を削除した場合においても，必須命令の割合は手法 A より低い．したがって，手法 B は共有メモリ参照だけでなく，アドレスの再計算などの命令も多いため，手法 A よりも性能が低いといえる．

最後に GPU 上での計算時間に加え，GM の確保および入出力データ転送時間を含めた実行時間について述べる．実行時間は，$n = 8{,}192$ のとき手法 A において 4,940 ミリ秒であり，表 1 に示す計算時間が実行時間の 98.6%を占める．一方，$n \leq 512$ の場合，GM を確保する処理の時間が相対的に大きく，実行時間の約 30〜60%を占める．したがって，n が小さい場合，メモリの確保および入出力データ転送は性能ボトルネックとなりうる．

6. まとめと今後の課題

全点対最短経路問題における距離行列を高速に求めるために，CUDA を用いた反復型 BFW 法の高速化を図った．GPU のメモリ帯域幅と演算性能比を基にブロック分割の大きさを $t = 32$ とし，それを実現するために 1 スレッドが複数個の行列要素を更新する．また，既存の高速な行列積ルーチンを応用し，レジスタを優先的に用いて余分な共有メモリ参照を削減し，演算効率の向上を図った．また，反復型 BFW 法において 2 段階のブロック化を用いて主に共有メモリへタイルを配置する手法も作成し，レジスタを優先的に用いる手法と計算時間および命令の構成を比較した．

結果，頂点数が 256〜1,024 個の場合，CUDA を用いた既存の再帰型手法よりも 4%以上高速であった．頂点数が多い場合は，性能は 1〜10%ほど下回ったが，演算効率は既存の再帰型手法と同じく用いた GPU のピーク演算性能の約 70%を達成した．

今後の課題は，GPU のオフチップメモリに格納できない大規模なグラフを複数の GPU を用いて処理することである．

謝辞　本研究の一部は，科学研究費補助金基盤研究（A）（2）（2024002）および大阪大学グローバル COE プログラム「予測医学基盤」の補助による．また，有益なご意見をいただいた査読者の方々に感謝いたします．

参 考 文 献

1) Floyd, R.W.: Algorithm 97: Shortest path, *Comm. ACM*, Vol.5, No.6, p.345 (1962).
2) Warshall, S.: A Theorem on Boolean Matrices, *J. ACM*, Vol.9, No.1, pp.11–12 (1962).
3) Katz, G.J. and Kider, J.T.: All-Pairs Shortest-Paths for Large Graphs on the GPU, *Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware* (*GH'08*), pp.47–55 (2008).
4) nVIDIA Corporation: CUDA Programming Guide Version 2.3 (2009). http://developer.nvidia.com/cuda/
5) Buluç, A., Gilbert, J.R. and Budak, C.: Gaussian Elimination Based Algorithms on the GPU, Technical Report UCSB/CS-2008-15, University of California (2008).

6) Volkov, V. and Demmel, J.W.: Benchmarking GPUs to Tune Dense Linear Algebra, *Proc. Int'l Conf. High Performance Computing, Networking and Storage* (*SC'08*) (2008). p.11 (CD-ROM).
7) Harish, P. and Narayanan, P.J.: Accelerating Large Graph Algorithms on the GPU using CUDA, *Proc. 14th Int'l Conf. High Performance Computing* (*HiPC'07*), pp.197–208 (2007).
8) Venkataraman, G., Sahni, S. and Mukhopadhyaya, S.: A Blocked All-Pairs Shortest-Path Algorithm, *Proc. 7th Scandinavian Workshop Algorithm Theory* (*SWAT'07*), pp.419–432 (2000).
9) Matsumoto, K. and Sedukhin, S.G.: A Solution of the All-Pairs Shortest Paths Problem on the Cell Broadband Engine Processor, *IEICE Trans. Inf. Syst.*, Vol.E92-D, No.6, pp.1225–1231 (2009).
10) Bondhugula, U., Devulapalli, A., Dinan, J., Fernando, J., Wyckoff, P., Stahlberg, E. and Sadayappan, P.: Hardware/Software Integration for FPGA-based All-Pairs Shortest-Paths, *Proc. 14th IEEE Symp. Field-Programmable Custom Computing Machines* (*FCCM'06*), pp.152–164 (2006).
11) Han, S.-C., Franchetti, F. and Püshel, M.: Program Generation for the All-Pairs Shortest Path Problem, *Proc. 15th Int'l Conf. Parallel Architectures and Compilation Techniques* (*PACT'07*), pp.222–232 (2006).
12) 9th DIMACS implementation challenge — Shortest paths. http://www.dis.uniroma1.it/~challenge9/download.shtml
13) vander Laan, W.J.: decuda. http://wiki.github.com/laanwj/decuda

(平成 21 年 10 月 2 日受付)
(平成 22 年 4 月 1 日採録)

奥山　倫弘（学生会員）
平成 20 年大阪大学基礎工学部情報科学科卒業．平成 22 年同大学院情報科学研究科修士課程修了．現在，同大学院情報科学研究科博士課程 1 年．GPU を用いた汎用計算の高速化に関する研究に従事．

伊野　文彦（正会員）
平成 10 年大阪大学基礎工学部情報工学科卒業．平成 12 年同大学院基礎工学研究科修士課程修了．平成 14 年同大学院同研究科博士課程中退．同年，同大学助手．現在，同大学准教授．博士（情報科学）．高性能計算に関する研究に従事．

萩原　兼一（正会員）
昭和 49 年大阪大学基礎工学部情報工学科卒業．昭和 54 年同大学院基礎工学研究科博士課程修了．工学博士．同大学助手，講師，助教授を経て，平成 5 年奈良先端科学技術大学院大学教授．平成 6 年より大阪大学教授．平成 4〜5 年文部省在外研究員（米国メリーランド大）．現在，並列処理の基礎および応用に興味を持っている．