Gather 機能を有するメモリアクセラレータの疎行列計算への応用

田邊 昇[†]　小郷 絢子[‡]　小川 裕佳[‡]　高田 雅美[‡]　城 和貴[‡]

本論文では，疎行列ベクトル積のベクトルがデバイスメモリに入りきらないほど大きな問題向けの並列処理方式を提案する．提案手法は GPU が Gather 機能を有する大容量機能メモリ(メモリアクセラレータ)をアクセスするシステムを用いる．長い行を適切な折り目で折り畳む提案アルゴリズム（Fold 法）が負荷分散を改善し並列性を高める．これが生成した行列を転置して用いる方式は GPU 向けのアクセス順序にしている．フロリダ大の疎行列コレクションを用いて提案方式の性能評価を行った．その結果，間接アクセスの直接アクセス化により，単体性能は既存研究の最大 4.1 倍に向上した．GPU 内キャッシュが溢れる心配も無い．GPU 間の 1 対 1 通信を完全に排除可能にした構成によりスケーラビリティは保証されており，機能メモリとのインタフェースのバースト転送バンド幅で制約される単体性能にノード数を乗じたものが並列実効性能となる．

Application for Sparse Matrix Computation of a Memory Accelerator with Gather Functions

NOBORU TANABE [†], JUNKO KOGOU [‡], YUKA OGAWA [‡],
MASAMI TAKATA [‡] and KAZUKI JOE [‡]

In this paper, we propose a parallel processing strategy for huge scale sparse matrix-vector product whose vector cannot be held on a device memory. The strategy uses a system with GPUs and functional memories named Memory Accelerator with gather function. Proposed algorithm named "Fold method" improves load distribution and parallelism. Transposing matrix produced by it improves access sequence for GPU. We evaluate the performance of proposed strategy with University of Florida Sparse Matrix Collection. The result shows the 4.1 times acceleration over the existing performance record with a GPU in the maximum case. There is no risk of performance degradation by overflowing cache capacity on GPU. Because of the architecture without inter-GPU communications, scalability is guaranteed. Therefore, parallel effective performance is the product of number of nodes and single GPU performance limited by burst transfer bandwidth of interface of functional memory.

1. はじめに

ベクトル型スーパーコンピュータの演算能力は COTS(commercial off-the-shelf)の CPU や GPU で代替可能なケースが多い．GPU の演算能力は既に 1 TFLOPS を超えており，それを生かした GPGPU 研究の成功例[1]は数多く報告されている．一方，キャッシュや GPU の統合メモリアクセスでは救済できない大容量メモリに対するランダムアクセスを主体にするアプリケーションでは, 必ずしも COTS がベクトル型スーパーコンピュータを代替できない．GPU 基板上のデバイスメモリの容量は現状では最大でも 6GB にすぎない．それを超える大規模データを処理する場合、バースト転送しか効率的に実行できない通信経路(PCI express)がボトルネックになっていた．

上記の問題を解決するため，筆者らは Scatter/Gather 機能を有する拡張大容量機能メモリ(メモリアクセラレータ)と GPU を併用したヘテロジニアスシステムを提案する．さらに，その上での疎

† (株)東芝
Toshiba corporation
‡ 奈良女子大学
Nara women's university

行列ベクトル積のスケーラブルな高速化を提案する．

以下､本論文では第 2 章で解決すべき課題を示し，第 3 章で提案システムのアーキテクチャを述べる．第 4 章では提案システム向けの疎行列ベクトル積アルゴリズムを示す．第 5 章では従来の GPU クラスタや提案システムで想定される性能ボトルネックについて述べる．第 6 章では性能評価を示し，最後に第 7 章でまとめる．

2. 解決すべき課題

本研究ではアプリケーションとして疎行列ベクトル積を検討対象とする．疎行列ベクトル積は連立一次方程式や固有値求解において最もよく使われる CG 法を代表とするクリロフ部分空間法の中核的処理である．よって非常に広範囲の科学技術計算アプリケーション上で実行時間の大半を占める．このため，数多くの研究がこの高速化に向けて行われてきた．しかしながら，とりわけランダムに近い非零要素配置を有する行列を扱う場合，キャッシュが効きにくく，メモリバンド幅がボトルネックとなる．このためベクトル型スーパーコンピュータ以外での効率的な処理は容易ではなかった．一方，近年では広大なメモリバンド幅を背景に GPGPU でも複数の実装成功例[1][2][3][4]が報告されるようになってきた．ただし，CPU よりも GPU の方がメモリ容量と引き換えに広いメモリバンド幅が実装されていることが性能向上の要因であり，GPU 内部の演算器は遊んでいる状態にある．

図 1 に示すように疎行列ベクトル積の処理は疎行列を構成する行ベクトル群と列ベクトルの積に分解できる．行間にはデータ依存関係が無いため，メモリ容量の制約に合わせ疎行列を行単位で GPU に分割することは基本的には容易である．

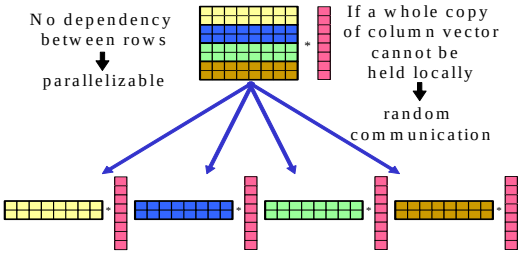

図 1　疎行列ベクトル積における並列性

ただし，GPU における従来の実装においては列ベクトルの大きさとデバイスメモリの間の大小関係における制約が存在する．つまり，入力された列ベクトル全てを GPU がローカルにコピーを保持できないと，行ベクトルの非零要素の位置に対応する列ベクトルの要素を読み出す際に，ランダムでバースト長が短い GPU 間通信が発生してしまう．非零要素の配置パターンは多様であるため，アプリケーションへの汎用性を保ったまま効率的にタイリングを行うことは困難である．

本研究では疎行列そのもののみならず，疎行列に乗じられる密な列ベクトルすら 1 個の GPU のデバイスメモリに入りきらない大きな問題を対象とする．例えば 6GB のデバイスメモリがある GPU において列ベクトルと行ベクトルを半分ずつ使って格納した場合，倍精度浮動小数の要素数が 375M 個を超えるベクトルを扱うとランダムアクセスが GPU 外部に溢れる．つまり 1000^3 以上の格子点を扱う大規模な行列ベクトル積を単純な GPU クラスタは現実的には並列処理することが困難である．

3. 提案システムアーキテクチャ

3.1 基本コンセプト

機能メモリ(メモリアクセラレータ)と GPU 等のアクセラレータを組み合わせることにより，従来のシステムでは効率が悪かったメモリアクセスを効率化し，その結果として高い実行性能を得る方式を提案する．図 2 に機能メモリのインタフェースに PCI express 等の高バンド幅な標準 I/O を用いる場合の提案アーキテクチャの基本概念を示す．

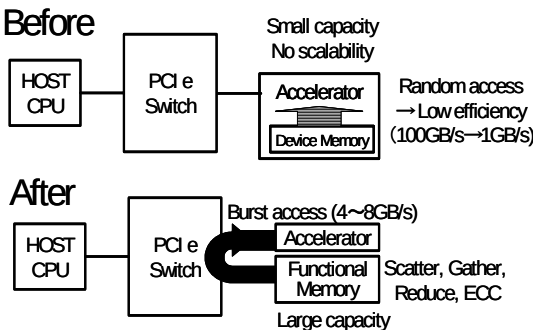

図 2　提案アーキテクチャの基本概念

機能メモリはアクセラレータの外付けデバイスとして，メモリ容量の厳しい制限を解消し，エラー訂正機能が付いた拡張メモリとして用いられる．さらに機能メモリはホストの主記憶と異なり，PCI express 等の標準 I/O を通過するデータ量を削減する機能や転送効率を向上させるための機能を有する．機能メモリの具体的な機能として代表的なものは，DIMMnet-2[5]や DIMMnet-3 [6]に実装されている Scatter/Gather(分散/収集)機能である．

筆者らが行った予備評価[18]では従来の GPU ではランダムアクセスバンド幅が 1GB/s 程度しか得られなかった．提案システムでは Scatter/Gather 機能によってランダムアクセスがバーストアクセスに変換される．PCI express を経由してデバイスメモリに転送する場合は，大きいサイズの疎行列ベクトル積の列ベクトルのリストアクセスのように Scatter/Gather 機能付き転送コマンドの起動頻度を抑制できるアプリケーションでは 4～8GB/s のピークバンド幅に近いバンド幅が得られるようになる．

なお，機能メモリのインタフェースとしては，上記の説明では一例として PCI express を用いるものについて説明した．他にも GPU ベンダー側での対応が必要になるものの GPU 向けには SLI (Scalable Line Interconnect)や GDDR5 デバイスメモリインタフェースなどの高バンド幅なインタフェースの利用が考えられる．高バンド幅なインタフェースは一般的にバースト長が長くないと本来の性能を発揮できないことが多い．しかし，機能メモリの分散/収集機能によりバースト長が飛躍的に伸び，転送効率の向上が期待できる．特に GDDR5 DRAM は 1 チップあたり現段階で 7Gbps ×32 ビット幅で 28GB/s が得られる．

NVIDIA® Tesla™ C2050/2070 は GDDR5 ポートを 384 ビット分(32 ビット×12)に設置している．この GDDR5 ポートを将来の GPU 上で機能メモリ用に 1 チップ分増設または切換え可能に改良することで PCI express より高いバンド幅を機能メモリ側に提供することが可能であると考えられる．理論上，GDDR5 はさらに 4 倍の 28Gbps まで向上できるとされており，本用途への応用は有望と思われる．

さらに，将来展望としては NVIDIA が提案している Exa FLOPS マシン用アーキテクチャである Echelon は Hybrid Memory Cube(HMC)[7]を GPU の発展形のメニーコアプロセッサの主記憶とする．Micron 社により Hotchips23 で詳細が発表された HMC は複数の多バンクの DRAM とコントローラを三次元的に積層実装したメモリモジュールである．米国の Exa FLOPS マシンの開発機関である IAA が公開している資料[8]には Scatter/Gather 機能を有するメモリシステムを開発する Memory project が明記されており，その開発機関に Micron 社が挙げられている．以上から，将来の HPC 向け HMC の中に Scatter/Gather 機能が入る可能性は大きいと考えられる．Scatter/Gather 機能が入った HMC を GPU に接続する場合は，GPU のデバイスメモリが本研究で提案している機能メモリと同等になる．

3.2 処理の流れ

図 3 に基本概念に基づく機能メモリアクセス処理

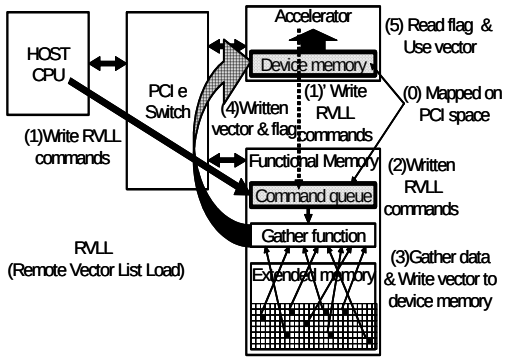

図 3. 機能メモリアクセス処理の流れ.

の流れを示す.

(1) 機能メモリ(例えば DIMMnet-3)へのコマンドキューはメモリ空間上にマップされる．よって，ホスト（またはアクセラレータ）はアクセスしたい機能メモリに割り当てられた上記のコマンドキューに対応するアドレスに所定フォーマットでコマンドを書き込む.

(2) 上記書き込みトランザクションは実行され，アクセスする機能メモリのコマンドキューにコマンドが書き込まれる.

(3) 機能メモリはコマンドキューからコマンドを取り出して，記載された内容の機能(例えば遠隔リストベクトルロード : RVLL)を実行し，指定があれば応答データや完了フラグをコマンドに記述されたアドレスに書き込む.

(4) 上記書き込みトランザクションは実行され，コマンドは終了する.

(5) ホスト（またはアクセラレータ）は十分に余裕のあるタイミングでコマンドを起動できない場合は必要に応じて上記完了フラグをポーリングする．もしコマンドが完了していれば，デバイスメモリ上に連続化かつアライメント調整されて格納済みのベクトルデータに対する後続の処理を実行する.

3.3 実効メモリバンド幅のスケーラビリティ

疎行列ベクトル積等は実効メモリバンド幅が性能を決めるので，多数のアクセラレータで処理する場合，演算能力の増加に見合った実効メモリバンド幅が得られないと実効 FLOPS 値は向上しない．図 1 に示されたような疎行列ベクトル積の並列化を行う場合を想定すると，アクセラレータの処理能力と機能メモリの転送能力のバランスを考慮してアクセラレータと機能メモリの個数の比率を決め，列ベクトルのコピーを複数の機能メモリに保持することで実効バンド幅のスケーラビリティを維持できる.

4. 提案システム向け疎行列ベクトル積

本章では，GPU のデバイスメモリに入りきらないほど大きなベクトルに対する疎行列ベクトル積の提案システム向けの手順について論じる.

4.1 基本方針

GPU 間の細粒度な 1 対 1 通信を排除することでスケーラビリティを得ることを目指し，図 1 に示された並列化方針で提案システムを用いて疎行列ベクトル積を実行するものとする．列ベクトルは全機能メモリにコピーを保持する.

行列ベクトル積においては，行列データは 1 回の積和演算にしか用いられないため再利用性が無い．つまり行列データを共有メモリやキャッシュによって再利用する意義はない．行列に乗ずるベクトルには多少の再利用性が存在する．しかし，帯幅が大きくない帯行列的な非零要素の配置でない限り，GPU 上の小容量なキャッシュではデバイスメモリ（キャッシングされる Texture メモリ）に入りきらないほどの大きなベクトルを効率的に再利用することは困難であると考えられる．よって，行列データや行列に乗ずるベクトルデータの格納法やアクセス方法は，再利用よりもグローバルメモリからの転送を効率化することを優先して考える.

GPU 上で実行する前の前処理として，GPU が扱いやすい状態にデータ構造を整える．これに加えて，GPU 向けの最適化を促進するためには，実効バンド幅が高い Coalesced access になるようにする点，最内側ループ内に IF 文が来ないようにする点，スレッドを多数起動して負荷が均衡するようにする点などを考慮する必要がある.

4.2 前処理

以上を踏まえて，以下の前処理を行う．前処理における行列の整形と転置の流れを図 4 示す.

(1) 行列の整形

アプリケーションによって一行あたりの非零要素数には差があるとともに，その値のばらつき方も異なる．単純な行分割による負荷分散では非零要素数最大の行のみによって実効時間が決まってしまう．これを回避するために，行列の形を整形する必要がある.

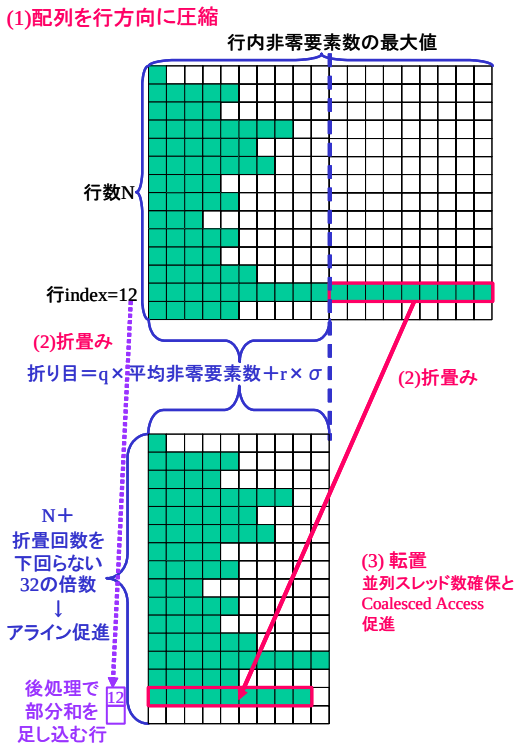

図 4　前処理における行列の整形と転置の流れ

その方法にはいくつか考えられるが，例えばホスト上で適宜 0 パディング（CRS 形式などで省略されていた零要素の位置に記憶領域を割り当て，そこを 0 で初期化すること）や折り畳み（非零要素が多い行を分割し，複数スレッドに割り当てること）を行うことで行列の形を整形する．この例では，大半の行で折り畳みが生じず，かつ，一行あたりの平均非零要素数にできるだけ近い列数を持つ縦長の二次元配列に整形する．この列数が全スレッドの最内ループの回数となるため，これを最適化することが実行時間短縮につながる．

例えば，折り目の位置を行内非零要素数の平均に係数 q を乗じたものとし，最適な q の値を経験的に探す．本論文の後半の評価においては q=1.5 の場合について評価を行った．

他の例としては，折り目を行あたり平均非零要素数＋行あたり非零要素数の標準偏差 σ ×r とする方式もありうる．ここで r=2 とすれば，分布を正規分布と仮定した場合には 95.4%の行で折り畳みが生じないようにできるので概ね 10%以下の行数増加に留めつつ，実行時間に直結するカーネルの最内ループ回数を行内最大非零要素数から行内平均非零要素数＋2 σ に短縮できる．σ の導入は分布の違いをある程度反映した折り目を与えると考えられる．しかし，平均値以下の位置で積極的に折り畳むと良い場合をこのままでは反映できない．

より汎用な最適化指標を与えるべく，上記の二つを併合した q×行内非零要素数の平均＋r× σ を折り目として，最適値を与える係数(q,r)の探索は今後の課題とする．

なお，GPU での最適化に関して，アラインメントを考慮する必要があるが，次のステップ（転置）に伴い，上記の列数はアラインメントには影響しない．

一方，整形後の配列の行数はスレッド数に対応する．これが半端な値であると次のステップ（転置）によって行の先頭位置のアラインメントがずれてしまう．よって，折り畳み分を加算した行数より大きく，かつ 32（複数 GPU で実行する場合は GPU 数×32）で割り切れる行数に，0 パディングによって整形する．ここまでの前処理アルゴリズムを Fold 法と名づける．上記の Fold 法では機能メモリを一切使っておらず，通常の GPU のみにも適用することができ，負荷分散等の効果が期待できる．本論文の主眼は機能メモリを用いる場合の評価にあり，用いない場合の評価は今後の課題とする．

(2)　行列およびインデックスの転置と転送

通常の CRS 形式による行列の格納方式によれば，行列の非零要素は同じ行の非零要素がアドレス連続方向に並ぶ．一方，GPU は隣接スレッドが同時にアクセスするデータが隣接アドレスに並ぶ時に最も効率的なメモリアクセスになる．各スレッドが行または部分行を担当するようにカーネル処理を割り当てた場合，上記の条件を満たすために(1)において整形した配列を転置する．その結果，横長の二次元配列となる．

複数 GPU で実行する場合は上記の横長配列を縦方向に等分した配列を各 GPU に分配する．一行あたりの平均非零要素数が多い行列の場合，転置処理自体がキャッシュベースのホスト CPU には苦手な処理になる．その場合は，(1)でできた配列を機能メモリに転送し，機能メモリ上で等間隔アクセスによる並べ替えを行なう．その上で，GPU のグローバルメモリにバースト転送することで問題を回避できる．

(3)　機能メモリによるベクトルのプリロード

CRS形式やJDS形式などによって非零要素のみを用いた行列ベクトル積を行う場合，カーネルの最内側ループには通常だと間接参照が必要になる．つまり，乗ずるベクトルを格納する配列のインデックスが配列になっているループである．この配列が GPU のグローバルメモリに入りきらない場合には，ランダムな GPU 間通信が発生してしまい，GPU 台数を大きくした場合のスケーラビリティに重大な問題が発生するケースが多くなると考えられる．

そこで，容量の制約が GPU より緩い拡張機能メモリを適切な台数の GPU ごとに設置する．そこに乗ずるベクトルを格納することで，この小規模クラスタ内部に全ての 1 対 1 通信を閉じ込める．

図 5 に機能メモリによるベクトルのプリロードの流れを示す．機能メモリには(2)において転置したインデックス配列(複数の機能メモリを有する場合は，担当する GPU 向けに(2)において分割されたインデックス配列)を指定した間接ベクトルロードコマンドを各機能メモリ上で実行することで，必要なデータを機能メモリの buffer 上に Gather し，GPU のデバイスメモリ(グローバルメモリ)に PCI express バスを介してバースト転送する．こうして GPU の PCI express バスは効率的に動作するようになる．その結果，GPU 上では隣接スレッドがグローバルメモリ上の連続アドレスをアクセスするようにデータが並び，メモリアクセスが高速化する．

なお，上記の動作は 3.2(3)に示されるように、PCI express 接続を用いる場合は，RVLL コマンドにより起動される機能メモリ側のハード(DMA)が PCI ア

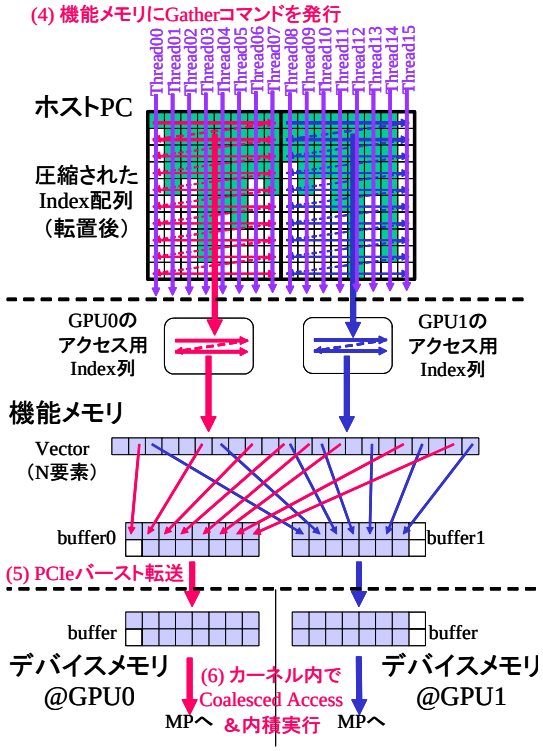

図５　機能メモリによるベクトルのプリロードの流れ

ドレス空間上にマップされたデバイスメモリ上のバッファ領域に、バースト書き込み転送を行う。1 回の疎行列ベクトル積では各 GPU が担当するパディングを含む非零要素数(＝ベクトルへの間接参照の全アクセス数)×データ 1 個のサイズ(例えばバッファをデバイスメモリの半分以下の 1GB に設定したなら 1GB)のバースト長で 1 回転送を行うことになる。このため，ほとんどのタイミングでソフトウェアは介在せず，DMA セットアップに関するオーバーヘッドは無視でき，PCI express のピークに近いバンド幅が期待できる。この動作はホスト OS が介在する可能性のある cudaMemcpy 関数で GB 単位のデータをホスト～GPU 間転送する場合と同等以上のバンド幅が出るように実装可能と考えられる．

4.3　カーネル部

GPU で実行されるカーネル部は，上記の前処理によって同じ長さの短い密ベクトルと密ベクトルの内積処理を多数のスレッドが実行する状態に置き換えられる．行列およびベクトルへのアクセスはアラインメントされた位置からのスレッド番号順に連続するグローバルメモリへの直接参照となり，全アクセスが Coalesced access となる．

4.4　後処理

カーネル処理を終えたところで，各スレッドが累積したスカラ値からなる部分ベクトルをホストの主記憶に転送する．折り畳んだ行については部分和を足しこんで，最終的な結果ベクトルの値を計算する．この計算を複数の GPU で行うと別 GPU にある部分和との加算が発生してスケーラビリティが低下する可能性がある．さらに GPU は IF 文の実行がホスト CPU に比べて得意ではない．このため，部分和を全てホストに転送し，ホスト CPU 上で折り畳んだ行の値を足しこむのが望ましいと考えられる．

5.　想定されるボトルネック

本章では，提案方式やそれを用いない GPU クラスタで想定されるボトルネックについて考察する．

5.1　デバイスメモリバンド幅

単精度浮動小数の密ベクトルと密ベクトルの内積に要するメモリバンド幅と演算の比率は 4 バイト/FLOP である．提案方式ではデバイスメモリ上に両方のベクトルが存在することになるので，デバイスメモリバンド幅がボトルネックになる場合の FLOPS 値は，評価の章で用いた Tesla C1060 の場合は 103GB/s/4B/FLOP＝25.75 GFLOPS となる．

一方，提案システムを用いない単純な GPU クラスタの場合，特に非零要素の位置にあるベクトルの要素を別の GPU から集めてこなければならない．このため，その実効バンド幅は上記のデバイスメモリバンド幅とはかけ離れたものになる．ローカルのデバイスメモリアクセスで済む場合と済まない場合の比率によって，ベクトルに対する実効デバイスメモリバンド幅は大きく変動する．

5.2　GPU～機能メモリ間インタフェースバンド幅

PCI express Gen.2 x16 のピークバンド幅はどちらの GPU でも片方向あたり 8GB/s である．提案方式ではホストまたは機能メモリからの行列の転送や乗ずるベクトルの転送はこの経路で行われるので，デバイスメモリへのアクセスが連続化された場合は，PCI express のバンド幅がボトルネックとなる．ただし，ここで発生する転送はバースト転送であるため転送効率は高い．

一方，提案システムを用いない単純な GPU クラスタの場合，他の GPU との通信がこの経路を用いることになる．その際のバースト長は長く取ることが困難なので，実効バンド幅は提案システムを用いるよりも大幅に低くなると予想される．さらにその通信は Infiniband などのノード間結合網を介するので，通常そのバンド幅は PCI express のバンド幅よりも低い．

5.3　機能メモリ実効バンド幅

機能メモリにおける不連続アクセス時の実効バンド幅が上記のバンド幅を維持できない場合はこれがボトルネックとなる．これは機能メモリを並列に用いることで補うことも可能である．

通常の PC の主記憶はキャッシュライン単位のアクセスに対して最適化されたメモリシステムである．つまり連続アクセスに対するバンド幅は効率的であるが，不連続アクセスに対するバンド幅は低い．一方，本用途に用いられる機能メモリは不連続アクセスのスループットを高める構成を取る．具体的にはベクトル型スーパーコンピュータのメモリシステムのように，多数のバンクから構成されるインターリーブドメモリに近い構成にすれば，不連続アクセススループットが高まる．他にも，Cell/B.E.の主記憶として有名な XDR-DRAM や，ネットワーク機器向けの FC-RAM は DRAM チップの内部に多くのバンクが存在する．このためこれらは不連続アクセススループットが高い機能メモリの構成に適していると考えられる．さらに Hybrid Memory Cube(HMC)は

表 1　測定環境(C1060 環境)

ホスト	Intel® Core(TM) i7 CPU 920 2.67GHz
GPU	Nvidia Tesla C1060(MP 数 30)
デバイスメモリ	メモリバンド幅 103GB/s,　4GB
ホスト I/F	PCI express x16 Gen.2 (最大バンド幅 8GB/s)
OS	Fedora10
CUDA	Cuda3.0

表 2　測定環境(C2050 環境)

CPU	Intel® Xeon(R) CPU X5670 2.93GHz
GPU	Nvidia Tesla C2050(コア数 448)
デバイスメモリ	メモリバンド幅 144GB/s，3GB
ホスト I/F	PCI express x16 Gen.2 (最大バンド幅 8GB/s)
OS	Red Hat Enterprise Linux Client release 5.5
CUDA	Cuda3.2

多数のメモリチャネルを内在していることから不連続アクセススループットが高いことが予想される．前述の通り，HMC の中に Scatter/Gather 機能が入る可能性が大きいと考えられる．

具体的な機能メモリの構成や，そこで得られる不連続アクセスの実効バンド幅の評価は別論文で扱うものとする．

6.　評価

6.1　実験環境とテスト行列

今回の実験に用いた計算機環境を表 1(C1060 環境*)および表 2(C2050 環境)に示す．また，実験に用いた行列を表 3 に示す．

行列は University of Florida Sparse Matrix Collection[8]から抜粋した．これらは本研究が想定する「乗ずるベクトルが GPU のデバイスメモリに入りきらないほど大きい問題」ではない．しかし，本評価ではそのような大きな問題を提案システム上の複数 GPU に分割して実行する場合に，各 GPU に分配されるデータが上記の行列集と同等の性質を保持していると仮定する．先行研究である Cevahir らの研究[4]でも同様の行列を用いて疎行列ベクトル積の実測値を公開しているが，今回は特に Cevahir らのプログラムであまり高速化しなかったものを中心に抜粋した．

ここで用いられている行列のサイズでは最大でも乗ずるベクトルは 6.3MB にすぎない．これはデバイスメモリ容量に比べると微々たるものである．よって，本来想定する状況よりもかなりキャッシュが効きやすい状況（先行研究に有利な状況設定）での評価であり，キャッシュを用いない提案方式には不利な状況設定での評価になる．

本研究は Cevahir らの研究と同様に Mixed Precision iterative refinement アルゴリズム[9]を使うことを想定している．このアルゴリズムは preconditioner として単精度の CG ソルバーを用いる．全てを倍精度で計算するよりも高速であることが知られている．このため，計算時間の大半を単精度の疎行列ベクトル積が占めることを本研究は想定しており，評価も単精度で行った．

また，本研究における性能評価の範囲については，CG 法のように同じ入力行列を用いて何度も繰り返し疎行列ベクトル積を計算することを想定している．よって，CPU から GPU に対して演算開始の指示を送る時点を開始時刻，GPU 上で演算が終了し CPU へ演算結果を書き戻すことが可能となる時点を終了時刻として扱い，計算前後の CPU～GPU 間の通信時間については性能評価の範囲に含めないこととする．

なお，本研究においては，機能メモリにおける Gather 操作のスループットは十分であるとする．その根拠は他研究[14] [15]において現実的なハードウェア量により PCI express のバンド幅と比較して十分なスループットが得られていることによる．

提案方式では機能メモリ→GPU 間の DMA 転送を GPU での計算とほぼオーバーラップして継続実行することが可能である．よって，本評価では機能メモリから GPU のデバイスメモリへの DMA 転送が終わっている整列済みのデータを GPU が使い続ける状況の性能を測定する．

また，前処理で整形された配列を生成する部分の時間は行列 1 個に対して 1 回だけかかるのみで，多数回実行される反復計算の実行時間に比べると少ない時間(高々数十秒程度)であるので，ホスト PC 上で別途行なっている．

表 3　評価に用いた行列

行列名	行数	非零要素数	非零要素数/行		
		総数	平均	最大	σ
Na5	5,832	155,731	26	185	35.7
msc10848	10,848	620,313	57	300	49.4
exdata_1	6,001	1,137,751	189	1501	390
G3_circuiit	1,585,478	4,623,152	2	4	2.2
thermal2	147,900	3,489,300	23	27	6.9
hood	220,542	5,494,489	24	51	13.3
F1	343,791	13,590,452	39	306	20.0
ldoor	952,203	23,737,339	24	49	12.9

6.2　評価プログラム

本章の評価に用いた CG 法のプログラムはカーネル部の列ベクトルアクセス手法が異なる以下の 3 種類である．いずれも 4 章で紹介した前処理を行ったものに対して疎行列ベクトル積を行うようなプログラムになっている．これらによってカーネル部の列ベクトルアクセス手法の違いのみがどのように処理速度に反映されるのかを知ることができる．

(1) テクスチャメモリ版
本プログラムは GPU のテクスチャメモリに列ベクトルを格納し，Tex2D 関数によってアクセスすることでテクスチャキャッシュの効果を利用するものである．性能の基準として用いるとともに，収束するまでの反復回数の採取も行い，後述する(3)の反復回数としてその値を用いる．

(2) 共有メモリ版
本プログラムは共有メモリを介してデバイスメモ

* Intel , Intel Core は、米国およびその他の国における Intel Corporation の商標です。

リ上の列ベクトルをアクセスするものである．Fermi(C2050 環境では上記のアクセスが汎用キャッシュ（L1 および L2 キャッシュ）によって加速される．

(3) 提案システム版

本プログラムは提案システムによってデバイスメモリ上に使用する前に整列された列ベクトルをアクセスして計算に用いるものである．ソースコード上は間接参照のかわりにアラインされた位置への直接参照によるバーストアクセスとなり，Coalesced アクセスとなる．文献[14] [15]の技術で十分なメモリアクセススループットが得られるものと仮定する．

6.3 テクスチャキャッシュにおけるヒット率

C1060 環境における(1)のプログラムのテクスチャキャッシュのヒット率を測定した．測定にはプロファイラを用いた．CUDA3.0 においては tex_cache_hit および tex_cache_miss という性能カウンタの値を計測することができる．

図 6 に行列サイズ(行数)とテクスチャキャッシュヒット率の関係を示す．行数が多くなると小さなテクスチャキャッシュから列ベクトルアクセスがはみ出すため，ヒット率が悪くなっていく傾向がわかる．線形近似を行った場合，急峻な傾斜で右下がりであり，行数が大きくなった時にこの勢いでヒット率が下がると今回の測定範囲以上の大きさの行列ではキャッシュの効果はほとんど期待できない．

測定に用いた行列の中で最も行数が多い G3_circuit(行数 1,585,478)ではヒット率は 7.74%に過ぎず，この程度の大きさの行列でもテクスチャキャッシュでは扱いきれず溢れている状態といわざるを得ない．G3_circuit は 1 行あたりの非零要素が平均 2 と少ないため，ライン内に再利用されるデータがほとんど載っていないこともヒット率を低くする原因と考えられる．

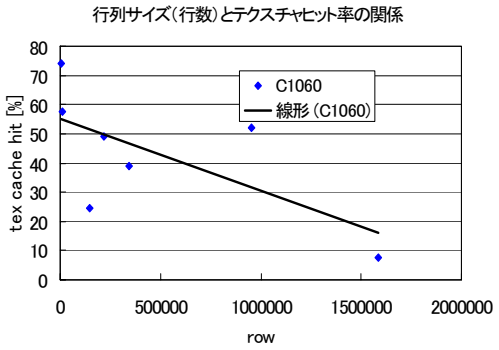

図 6. 行列の行数と Texture キャッシュヒット率の関係.

6.4 汎用キャッシュにおけるヒット率

C2050 環境における(1)のプログラムの汎用キャッシュ（L1 キャッシュ）のヒット率を測定した．測定にはプロファイラを用いた．CUDA3.0 においては l1_global_load_hit および l1_global_load_miss という性能カウンタの値を計測することができる．

図 7 に行列サイズ(行数)と L1 キャッシュヒット率の関係を示す．キャッシュの Preference は L1 キャッシュが大き目(L1 が 48KB，共有メモリが 16KB)となる設定における結果である．行数が多くなるとテクスチャキャッシュの場合と同様に，ヒット率が悪くなっていく傾向がわかる．G3_circuit が 26.5%であり，テクスチャキャッシュを C1060 上で用いる 7.74%よりは改善している．注意深く図 6 と図 7 を観察すると F1 はテクスチャキャッシュを用いる場合はヒット率がさほど低くないが，汎用キャッシュを用いる場合は 23.9%とヒット率が下がる．この現象は汎用キャッシュの場合，再利用性が無い行列データまでキャッシュを経由してしまっており，非零要素総数が多い F1 ではヒット率が減少する結果になったと考えられる．

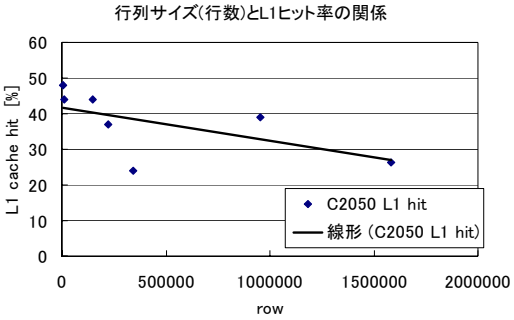

図 7. 行列の行数と L1 キャッシュヒット率の関係.

6.5 1GPU 内に収まる疎行列ベクトル積性能

上記の疎行列に対する疎行列ベクトル積の処理性能を測定した．提案手法の結果として示されている値は，提案システムによって GPU が使用するタイミングより前に GPU のデバイスメモリ上に機能メモリによるプリロードが完了していると仮定した場合の処理速度である．精度は単精度浮動小数とした．提案手法については折り畳みを行平均の 1.5 倍(q=1.5)で行う場合について測定した．その結果を表 4 に示す．

表 4 疎行列ベクトル積性能 [GFLOPS]の他実装との比較

	JDS[4]	提案手法
使用 GPU	S1070	C1060
Na5	3	5.31
msc10848	3.5	8.38
exdata_1	3.4	8.01
G3_circuiit	9	15.08
thermal2	3.3	13.54
hood	11.5	13.18
F1	7.1	11.25
ldoor	9.8	13.30

ここでは折り畳んだことにより発生する累積加算時間は隠蔽されるか，または全体の計算時間に比べ十分に小さいものと近似している．その根拠は，行数の 20%でしか累積加算(スカラ加算)は発生しておらず，q の最適化をすれば様々な形状の行列でそのような水準を維持するようにできること，発生した行については行間に依存関係が無く並列処理でき，その累積加算数より大きい一行内の非零要素数の長さ

の内積が実行時間の大半となることから，大半の行列では累積加算時間を外して考えても議論の大筋を外さないと考えられるためである．

比較対象として Cevahir らの研究における実測値を文献[4]のグラフから読み取り，併記している．S1070 の中身は C1060 と同等品が 4 個入っている製品であるため性能を直接比較可能である．上記は JDS 形式の行列格納法を基にしており，我々の最適化方針に近い方向性を有している．しかし，JDS 形式では GPU に送るべき配列が 4 種類になっており，我々の 2 種類より多い．さらに，Texture メモリに対するキャッシングによってバンド幅を改善しているもののベクトルへのアクセスは間接参照であるため，差が生じていると思われる．全ての行列で文献[4]の性能より高速化している．最も高速化したものは thermal2 で，キャッシュの効果を全く使っていないにもかかわらず，文献[4]の 4.1 倍の性能が得られた．

また，JDS 形式では結果の書き込みにおいても間接参照になっており，この部分が Coalesced access にならない．元来，JDS 形式は間接参照にも強いベクトルプロセッサ向けに開発された方式であり，この点で必ずしも GPU 向けになっていない．これに対して提案方式では結果の書き込みも全て Coalesced access になっており，この点も差が生じている要因の一つと考えられる．

平均の 1.5 倍の位置での折り畳みを適用した提案アルゴリズムによって表 5 のように列数(最内側ループ数，実行時間に対応)は最低で 1/5.3，平均 1/3.0 となるのに対し，行数(スレッド数)は今回測定した全行列に対しては平均 1.2 倍にしか増加しなかった．非零要素数が多い上位 5 種類に着目すると平均 1.08 倍にしか行は増えない．行の増加率は列の減少率による高速化を鈍らせる方向に働くが，大きな行列では行増加率が低水準にある．よって，折り畳みは行列サイズの増加に対して好ましい傾向を示していることがわかる．

なお，元から負荷分散がうまく行っていた thermal2 のみについては 1.5 倍の位置の折り目が最大値を超え，折畳みは発生しなかった．よって，この場合の動作は最大値合わせと同じ(加速率 1)になる．

上記では折り目を平均の 1.5 倍に固定して測定を行った．しかし，この 1.5 という係数 q には何らか根拠があるわけではなく，傾向をつかむために最初に測定を試みた条件に過ぎない．つまり，最適化の余地を残している．

なお，機能メモリのインタフェースとして 32 ビット幅の GDDR5 メモリポート 1 ポート分が追加された場合は，28GB/s すなわち 14GFLOPS 相当，PCI express Gen.3 x16 の 2 倍弱のバンド幅を機能メモリアクセスに用いることができる．その場合，機能メモリのインタフェースのバンド幅はボトルネックにならないと考えられる．

6.6 列ベクトルアクセス法の違いによるカーネル実行時間

前述の 3 種類の評価プログラムのカーネル実行時間を表 5 に示す．実行時間の積算値を反復回数で割り算した平均値を示している．時間測定には CUDA Event を用いる方法で行った．表中の数値の全て単位はミリ秒である．現状のカーネルは疎行列ベクトル積の大半の計算を担っているが，行折り畳みの部分和の足し込みを行う後処理についてはカーネル外（ホストでの実行）となっている．

表5　列ベクトルアクセス法の違いによる実行時間［ms］

	C1060			C2050	
	texture	共有	提案	共有	提案
Na5	0.082	0.110	0.037	0.048	0.040
msc1848	0.205	0.392	0.119	0.128	0.113
G3_circuit	1.281	1.558	0.750	0.757	0.583
thermal2	0.942	1.451	0.518	0.495	0.440
hood	1.176	2.338	0.812	0.827	0.740
F1	4.770	7.404	3.596	2.890	1.981
ldoor	4.987	10.33	3.568	3.577	3.188

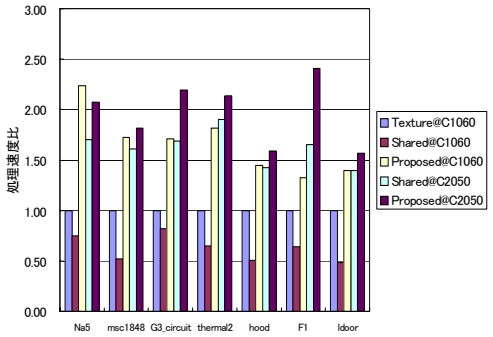

図 8.　列ベクトルアクセス法の違いによる処理速度比

C1060 のテクスチャキャッシュを用いたもの(Texture)を 1.0 とした際の速度比を図 8 に示す．結果としては，全ての行列において提案方式が高速である．ただし，追加ハードウェアの効果はこれらの小さめの行列に対しては限定的であることがわかる．この結果を言い換えると，提案ハードウェアは小さい行列におけるテクスチャキャッシュや汎用キャッシュがある程度効いている状態の GPU のスループットと同等以上のスループットをキャッシュの効果に頼らずに置き換え可能であることがわかる．前節の実験より行列の巨大化はキャッシュの効果を減らすことと合わせると，今回の実験より大きな行列を用いると提案ハードウェアの有無による差は広がると考えられる．

C2050 上では C1060 上であまり高速化しなかった Shared のプログラムがデバイスメモリバンド幅の向上と汎用キャッシュの効果によって C1060 上の提案方式なみに高速化した．この範囲の行列サイズでは L1 はミスだが，まだ L2 ではヒットしていると考えられる．しかし L1 全体に比べて L2 の容量は極端に大きいわけではないので，L1 と同様な限界性は行列をさらに大きくしていくと顕在化すると考えられる．

なお，文献[16]には F1 と ldoor の 2 つの行列には倍精度の場合の C2050 上での疎行列ベクトル積の JDS 形式等を用いた場合の処理時間が記載されている．提案システムを用いず前処理のみ用いている表 4 中の時間(C2050 の Shared)は，F1 の場合は文献[16]の JDS の 1/4.1，ldoor の場合は ELL の 1/2.74 の時間である．倍精度と単精度の違いでバンド幅が半分で済むことにより 2 倍弱の性能差が出ることを考慮しても，前処理アルゴリズム(Fold 法)だけでも従来方式より高速化が得られていると考えられる．Fold 法は Segmented Scan(SS)法[17]と同様に負荷分散

を改善するが，上記で取り上げた性能は文献[16]上で測定された SS 法よりも大幅に高速である．Fold 法は整形によりアラインさせコアレスドアクセスを促進した効果もあるが，主な理由は SS 法が本質的に GPU の演算器(乗算と加算の並列実行能力)を有効活用できないのに対し，Fold 法は内積演算が主体になるため有効活用できることが 2 倍の性能差を生むことによると考えられる．

6.7 1GPU 内に収まらない疎行列ベクトル積性能

まず，提案システム上で機能メモリにデータがセットされた状態から行列ベクトル積の結果を 1 回 GPU 上で計算し終わるまでの性能を考える．提案システムは 1 台の機能メモリとそれに接続している GPU をノードとして，それらが行分割によってノード間の通信を全く行わず，完全に並列に動作する．よって，機能メモリに乗ずるべきベクトルが入りきる十分に大きな問題の場合には，GPU 間通信というスケーラビリティ制約要因を完全に排除している．このため，GPU 内に収まる疎行列ベクトル積の性能に，ノード数を乗じたものがシステム全体の性能となる．さらに，GPU 間通信が皆無であることから，提案システムのスケーラビリティと行列の形状は無関係である．

提案システム上では GPU が必要なバンド幅と機能メモリの実効バンド幅のバランスに応じて GPU と機能メモリの個数の比率を決めれば良い．機能メモリを複数用いた場合には図 1 のようにベクトルのコピーが機能メモリに保持される．このため複数の機能メモリを反復法の中で用いる場合は，結果ベクトルを全ての機能メモリに書き込む時間が固有のオーバーヘッドとして加わる．このオーバーヘッドが反復法のプログラムの中で隠蔽できない場合は前段落での FLOPS 値は若干劣化する. 提案手法の反復法のプログラムへの実装と評価は今後の課題とする．

このオーバーヘッドは結果のホストへの転送時間と，ホストから全機能メモリへの放送時間からなる．これらはいずれもバースト転送になるため PCI express や相互結合網上では効率的に転送され，計算時間全体と比較して小さいと考えられる．前者(ホストへの転送)については全てのノードで並列実行できるのでスケーラビリティに影響を与えない．後者(機能メモリへの放送)については，同じデータを全てに放送する通信はネットワーク側の対応によってスループットをノード数に非依存にできる．つまり，ノード数に非依存な若干の遅延付加はあったとしても，スケーラビリティに影響を与えないように実装することができる．

一方，Cevahir らの研究[4]では，上記の評価で用いた行列については PCI express スイッチで接続された TeslaC1070 内部の 4 台の GPU を用いても 1 台の GPU の場合の 0.8 倍から 1.1 倍程度のスケーラビリティしかない．さらに，乗ずるべきベクトルがデバイスメモリ上に乗り切らない場合は，GPU ごとに分割してそのベクトルを保持することになる．このため，他の GPU が保持している部分ベクトル上のデータをネットワーク経由で取りに行く必要がある．分割台数が大きくなればなるほど，ローカルのデバイスメモリにある確率は減るためスケーラビリティ問題が深刻化すると考えられる．よって，デバイスメモリ上に全てが載っている場合の測定値である上記の FLOPS 値からさらに絶対性能や, スケーラビリティが劣化するのは確実であると考えられる．

さらに，Cevahir らの最近の別の研究[12]では，前処理として hypergraph-partitioning[13]を上記に追加して通信を抑制することで，スケーラビリティを改善した．その結果，32 ノードの PC クラスタ上で 64 台の Tesla を用いて 94GFLOPS を達成している．これを GPU1 台あたりにすると 1.47GFLOPS であり，1GPU での実行の FLOPS 値[4]よりかなり落ち込んでいる．より大きなクラスタに対してはパーティションの減少に伴い通信の増加が必然なため，更なる効率の低下が避けられないものと考えられる．さらに，パーティショニングは例えば分割した時に通信を発生させる境界接点数の全接点数に対する割合が少なくなる棒状のものを離散化したときのように本質的にうまく行く場合と，うまく行かない場合があり，スケーラビリティと行列の形状は敏感であると考えられる．これに対して，提案方式にはそのような欠点がない．

7. おわりに

本論文では，汎用キャッシュを搭載した GPU(C2050)において疎行列ベクトル積では 20～50%程度の L1 ヒット率しかなく, 行数が大きくなるほどヒット率が悪化する傾向を確認した．提案アーキテクチャは，メモリ容量とランダムアクセススループットを強化した機能メモリ(メモリアクセラレータ)がバーストアクセスにより GPU のデバイスメモリ上に整列したデータを書き込む．その結果キャッシュに依存せずに高いアクセス性能が得られる．

さらに，提案アーキテクチャ向けの疎行列ベクトル積のアルゴリズムを提案した．長い行を適切な折り目で折り畳む提案アルゴリズム（Fold 法）は負荷分散を改善し並列性を高める．これが生成した行列を転置して用いる方式は GPU 向けのアクセス順序にしている．Florida University Sparse Matrix Collection を用いて提案方式の性能評価を行った. その結果，単体性能においては，負荷分散が最初から取れている行列においては先行研究[4]の最大 4.1 倍の性能向上を観測した．行折り畳みを実装することで他の行列でも負荷分散が良くなり，測定に用いた全ての行列で加速が得られた．先行研究での測定はキャッシュがある程度効いている状態と考えられるが，本手法は先行研究とは異なり，キャッシュの効果を一切使っていない．よって，さらに大きな行列を扱う時のヒット率低下による性能低下の心配も無い．

GPU 側に GDDR5 メモリ 1～2 チップ分の専用ポート追加または切換え可能とする改良を加えたり，デバイスメモリとして Scatter/Gather 機能付きの Hybrid Memory Cube を採用することにより，PCI express を利用する構成におけるボトルネックを解消できると考えられる．

一方, 提案方式では細粒度でランダムな GPU 間通信がローカルな大容量メモリアクセラレータへのバーストアクセスに変換されている．このため，優れたスケーラビリティが確保されている．

今後の課題は行折り畳みの最適化を実装した評価, 加速率におけるアルゴリズム寄与分の分離評価，ストリーミングの実装と評価, CG 法などのクリロフ系

ソルバへの実装と評価，ライブラリやコンパイラ等のアプリ開発環境の整備，機能メモリの設計と評価などがある．

謝辞 本研究の一部(DIMMnet-3 の開発)は総務省戦略的情報通信研究開発推進制度(SCOPE)の一環として行われたものである．

参 考 文 献

[1] Nvidia : "CUDA Community Showcase", http://www.nvidia.com/object/cuda-apps-flash-new.html

[2] N. Bell, M. Garland : "Eficient Sparse Matrix-Vector Multiplication on CUDA", NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[3] M. M. Baskaran, R. Bordawekar : "Optimizing Sparse Matrix-Vector Multiplication on GPUs", IBM Research Report, RC24704, Apr. 2009.

[4] A. Cevahir, A. Nukada, S. Matsuoka : "An Efficient Conjugate Gradient Solver on Double Precision Multi-GPUSystems", Symposium on Advanced Computing Systems and Infrastructures (SACSIS2009), pp.353-360, May 2009.

[5] N. Tanabe, M. Nakatake, H. Hakozaki, Y. Dohi, H. Nakajo, H. Amano : "A New Memory Module for COTS-Based Personal Supercomputing", Innovative Architecture for Future Generation High-Performance Processors and Systems, pp.40-48, Jan. 2004.

[6] N. Tanabe, H. Hakozaki, Y. Dohi, Z. Luo, H. Nakajo : " An enhancer of memory and network for applications with large-capacity data and non-continuous data accessing", The Journal of Supercomputing, Vol. 51, No. 3, pp. 279-309, Dec. 2009.

[7] Micron Technology Inc. : “Hybrid Memory Cube ”, http://www.micron.com/innovations/hmc.html

[8] S. S. Dosanjh : “Exascale Computing and The Institute for Advanced Architectures and Algorithms (IAA) ”, http://www.hpcuserforum.com/presentations/Norfolk/Sandia%20IAA.hpcuser.ppt Apr. 2008.

[9] Tim Davis : " The University of Florida Sparse Matrix Collection", http://www.cise.ufl.edu/research/sparse/matrices/

[10] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. : "Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy", ACM Trans. Math. Softw. 34, 4, Article 17 Jul. 2008.

[11] A. Cevahir, A. Nukada, S. Matsuoka : " Fast Conjugate Gradients with Multiple GPUs", The International Conference on Computational Science 2009 (ICCS 2009), pp. 893-903, May, 2009.

[12] A. Cevahir, A. Nukada, S. Matsuoka : " High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning", Computer Science - Research and Development, Vol.25, No.1-2, pp.83-91, May 2010.

[13] U. V. Catalyurek and C. Aykanat, “Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication”, IEEE Transactions Parallel and Distributed Systems, vol. 10, no. 7, pp. 673. 693, 1999.

[14] 田邊, Nuttapon, 中條："Gather 機能付き拡張メモリのアクセス性能の評価", 情報処理学会 HPC 研究会, Vol.2010-HPC-128, Dec. 2010.

[15] 田邊, Nuttapon, 中條, 小川, 高田, 城："GPU と拡張メモリによる疎行列ベクトル積性能の行列形状依存性軽減", HPC 研究会研究報告 2010-HPC-129, Mar. 2011.

[16] 久保田, 高橋 :" GPU における格納形式自動選択による疎行列ベクトル積の高速化", HPC 研究会研究報告 2010-HPC-128, Dec. 2010.

[17] 大島，櫻井，片桐，中島，黒田，直野，猪貝，伊藤: "Segmented Scan 法の CUDA 向け最適化実装", HPC 研究会研究報告 2010-HPC-126, Aug.2010.

[18] 小川, 田邊, 高田, 城："GPU と機能メモリを用いたヘテロシステムによるスケーラブルな疎行列ベクトル積高速化の提案", SACSIS2010, pp.109-110, May 2010