2008－HPC－116 (34)
2008／8／7

GPUを効率的に利用するための言語拡張と自動最適化手法

佐藤功人† 滝沢寛之† 小林広明††

GPUは高性能グラフィックスプロセッサでありながら，汎用演算の高速化に大きな効果があり，様々なアプリケーションでの利用が試みられている．GPUは特有のハードウェア構成のために，高い演算能力を得るためには様々な制限を満たさなければならない．我々は異種プロセッサを搭載する計算システムに対して *SPRAT* を提案してきたが，プログラムの可搬性と実行効率を両立するためにはプロセッサ特性に合わせた言語の拡張とその自動最適化を行う必要がある．本論文では，GPU用にコードを自動的に最適化するための共有メモリの活用とミスアラインメントの影響を軽減する手法を提案し，メモリアクセスを調整することでエッジ検出処理とLU分解において実効性能を向上させることが可能であることを示した．

A programming language extension and its automatic optimization techniques for exploiting the potential of GPUs

Katuto Sato ,† Hiroyuki Takizawa † and Hiroaki Kobayashi††

GPUs have a great potencial of high-performance computing and have been used in various applications in addition to graphics processing. In order to achieve high-performance with GPUs, we have to carry out architecture-aware optimizations because of their unique architecture. We have proposed *SPRAT*, a programming language for hybrid systems of CPUs and GPUs, to realize both the portability of programs and the high computation effeciency. This paper proposes some automatic optimization techniques based on memory access adjustments. The results shows significant performance improvements in the executions of Edge detection and LU decomposition.

1. はじめに

近年，広く普及している描画処理用プロセッサ (Graphics Processing Unit, GPU) が持つ高い浮動小数点演算能力とメモリバンド幅に着目し，描画処理以外の汎用演算にもGPUを活用する研究が行われている[1]．一般的に汎用処理に用いられる汎用プロセッサ (CPU) と比較して，GPUはデータ並列処理に適した構造を持ち，様々なアプリケーションの高速化に大きな効果があることが知られている．

GPUはデータ並列処理では非常に高い性能を達成できる一方で，不得意な処理に対しては実効性能が大きく低下する．一般に，処理対象のデータ量が少ない場合には，立ち上げのオーバーヘッドが大きいGPUよりもCPUの方が実行時間が短くなり，データ量が大きい場合には並列処理に向くGPUの方が実行時間が短くなる傾向がある．これら2つを適切に使い分けるためには，実行時のランタイム支援によるプロセッサの自動選択機能を組み合わせることが有効である．我々はこの手法を実現するプラットフォームとして，データ並列処理を記述可能なプログラミング言語とプロセッサの自動選択機能を組み合わせた，SPRAT(Stream Programming with Runtime Auto-Tuning) を提案している[2]．SPRATでは，抽象度の高い拡張プログラミング言語で記述されたコードから，CPU用とGPU用のコードが自動生成される．SPRATはそれらのコードを実行時に使い分けることで，各処理に対して適切なプロセッサを割り当てる．

SPRATはハードウェアに依存しない抽象的なプログラミングモデルとして，ストリームプログラミングモデルを採用している．SPRATにより抽象度の高いコードから各プロセッサ向けのコードが自動生成されるため，プログラマがCPUやGPUの存在を意識する必要はない．しかし，実際にGPUの高い演算性能を引き出すためには，そのハードウェアの制限に起因する多くの条件や制約を満たすようにコードを記述しなければならず，それらの制約を満たせない場合には深刻な性能低下を招く．したがって，SPRATでGPUの演算性能を活用するためには，SPRATプログラミ

† 東北大学大学院情報科学研究科
Graduate School of Information Sciences. Tohoku University
†† 東北大学サイバーサイエンスセンター
Cyberscience Center. Tohoku University

ング言語によって記述された抽象度の高いコードから，GPU の特性に合わせたコードを自動生成する必要がある．

本論文では，抽象的な SPRAT 言語のコードから，GPU の特性を考慮したコードへと変換する手法を提案する．GPU ではメモリアクセス性能が実行時間に大きな影響を及ぼすため，本論文では GPU のオンチップメモリの活用とミスアラインメントの影響を軽減することを考える．そして提案する変換手法によって自動生成されたコードの実効性能を評価する．

2. CUDA 環境

CUDA(Compute Unified Device Architecture) は GPU 上でより柔軟なプログラミングを可能とし，同じプログラムを複数のプロセッサで実行してデータ処理を行う SPMD(Single Program, Multiple Data) モデルを基本としたプログラミングモデルを採用している[3]．また，範囲が限定されながらも，スレッド間の同期やデータの共有が可能であり，メモリアクセスの自由度も高く設計されている．

以下，NVIDIA 社製の GPU である GeForce8800GTX を例として，CUDA を利用可能な GPU のアーキテクチャについて述べる．GeForce8800GTX のアーキテクチャを図 1 に示す．CUDA では，GPU は Device と呼ばれる．Device はグローバルメモリと非常に高いメモリバンド幅 (86.4GB/s) で接続されている．Device は 16 基の Stream Multiprocessor(以下，SM) から構成されており，SM には一般的な演算を行う 8 基の Stream Processor(以下，SP) と，初等関数演算などを行う 2 つの Super Function Unit(以下，SFU) によって構成されている．GeForce 8800GTX の最大の単精度浮動小数点演算性能は 345.6GFLOPS となっている．8 つの SP と 2 つの SFU は，1 つの命令デコーダを共有しており，SM に含まれる全ての SP は同時に同じ命令を実行する，SIMD 型の実行方式をとる．

CUDA では，プログラムの実体一つをスレッドと定義し，1 つの SP が 1 つのスレッドを処理する形をとる．各スレッドは論理的に独立に処理されるが，実装上は 32 スレッドを 1 単位とした**ワープ**毎に実行される．ワープは幾つかまとめられて**スレッドブロック**を構成し，このスレッドブロックが 1 つの SM に割り当てられる．スレッド間での同期やデータ共有は，このスレッドブロック単位で行うことができるが，スレッドブロックを超えた同期やデータ共有を行う手段は提供されていない．GPU での実行は，このスレッドブロックが幾つか集まったグリッド単位で行われ，全スレッドを同期させるためには，GPU 上での実行を完了することによってのみ行うことができる．

GPU には，プログラマが明示的に利用可能な幾つかのメモリを搭載しており，SM 毎に 8192 本のレジスタ，16KB の共有メモリを利用可能である．グローバルメモリへのアクセスが 200〜300 サイクルかかるのに対して，共有メモリはバンク競合しない限りレジスタと同等のサイクル数で読み書き可能であり，高い演算性能を達成するためには，この共有メモリの活用が重要となる．

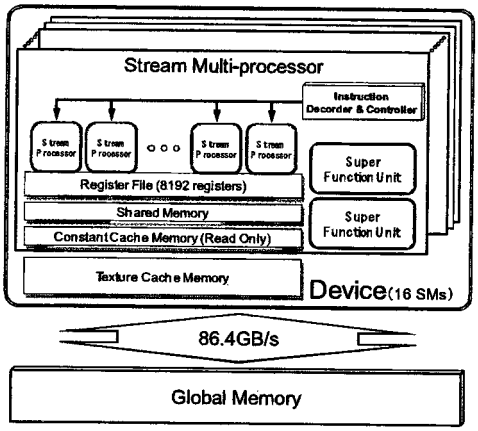


図 1 GeForce 8800GTX のアーキテクチャ

GPU の高いメモリアクセス性能を達成できるのは，ワープ内のスレッドが連続するメモリ領域にアクセスし，かつ先頭スレッドのアクセスするアドレスが適切にアラインメントされている (**アライン境界条件**) 場合に限られる．この条件が満たされている時，8 基の SP から発行されるメモリアクセス命令が融合され，1 回のメモリアクセス操作として実行される．

3. C を拡張したプログラミング言語 SPRAT

関連研究である BrookGPU[4] と同様に，SPRAT は GPU をストリーム処理用プロセッサとして抽象化することで，プログラムに可搬性を持たせている[2]．ストリームプログラミングモデルを導入するために，SPRAT ではストリームを宣言するための予約語である `stream` と，カーネル機能を提供するために関数を修飾する `kernel` を C 言語に追加した．例として，単精度浮動小数点型の 2 次元ストリーム (10 × 10) を宣言するには，

```
stream<float> a(10, 10);
```

と記述する．ストリームに格納するデータ型は，宣言時に< >の中に型名を記述することで指定する．ストリーム宣言時にグローバルメモリ上に暗黙的に領域が確保され，その先頭アドレスは常にアライン境界条件を満たしている．

ストリームには，他のストリームの一部を参照する**参照ストリーム**も存在する．参照ストリームは

```
stream<float>& ref = a[2][2](4, 4);
```

と記述し，ストリームの一部を切り出すことによって生成される．参照ストリームは，通常のストリームと同様に扱えるが，ストリームの一部を切り出すために，参照ストリームが対応する範囲のメモリ領域では，先頭アドレスはアライン境界条件を満たしているとは限らない．

ストリームを引数としてカーネル関数に渡すことで，ストリーム処理を実現する．引数にストリームを記述するときには，in/out/inout/gather によって修飾することで，データの入出力方向を規定する．in は入力ストリームであることを示しており，カーネル関数中では読み取り専用のストリームとしてアクセスされる．out は出力ストリームであることを示し，書き込み専用ストリームとしてアクセスされる．inout は入出力ストリームであり，カーネル関数中で読み書きすることが可能である．gather は特別な入力ストリームであり，ストリーム中の任意の位置から読み込むことができる．これらの修飾子を指定しないストリームは，暗黙裏に in が指定されているとして処理される．gather ストリーム以外は，カーネル関数内でアクセス可能な要素は暗黙裏に決定されており，その要素以外にアクセスすることはできない．

カーネル関数から，出力ストリームの各要素に対応するスレッド (**カーネルスレッド**) が生成される．各カーネルスレッドには固有の座標が割り当てられ，その座標に応じて処理内容を変えることも可能である．カーネルスレッドの実行順序は実行環境に依存するため，実行順序に依存して処理結果が変わるようなプログラムを記述した場合には，その出力結果は保証されない．

上記の予約語を用いた saxpy カーネルの記述例を以下に示す．

```
kernel map saxpy(float pi, in stream<float> x,
      in stream<float> y, out stream<float> z)
{
      z = x * pi + y;
}
```

saxpy カーネルは，2 つの入力ストリームから要素を取得し，片方を pi 倍した後で，それらの和を出力ストリームへと書き込む．入力および出力カーネルのアクセス要素は暗黙裏に決定されるため，カーネル関数中ではストリーム変数名を用いて要素にアクセスする．

一方，gather 型入力ストリームはカーネル関数中で任意の要素にアクセス可能なストリームであり，アクセス位置の指定は C 言語における配列インデックス指定の構文を用いて行う．カーネル関数内のストリームに対するインデックス指定には，相対位置アクセス指定と絶対位置アクセス指定が存在する．通常 C 言語で用いられるインデックス指定 [n] は，相対位置アクセス指定として認識され，カーネルスレッド座標に対しての相対座標で対象要素位置を指定する．一方，SPRAT 言語では，カーネル関数内部でのみ利用可能な絶対インデックス指定が拡張されている．絶対位置アクセス指定は，[[n]] のように記述され，カーネルスレッドの座標に関係なく，ストリーム上の絶対位置によってアクセス対象要素を指定することが可能である．gather 型のストリームを用いた filter カーネル関数の例を以下に示す．

```
kernel map filter(gather stream<float> x,
                  out stream<float> y)
{
      y = x[-1][0] + x[0][-1]
        + x[1][0] + x[0][1];
}
```

filter カーネルは，入力ストリームに対して現在のカーネルスレッド座標から上下左右に 1 だけずれた座標位置にある要素を読み込み，それらの和を出力ストリームに書き出す．データ並列処理では，ある要素を演算するために周囲の要素を参照する処理が頻繁にありこれらを効率よく扱うために，相対位置指定を通常のアクセス要素指定方法として用意している．このように，アクセス対象要素を定数でプログラムに記述することにより，データの依存関係解析も容易になる効果が期待できる．

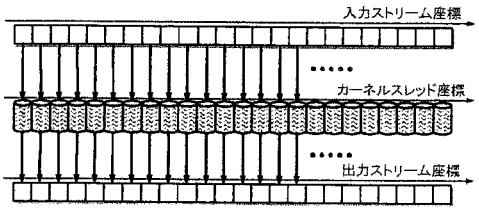
4. SPRAT 言語の自動最適化

SPRAT 言語では，抽象的なストリーム処理としてデータ並列処理を記述しているため，そのままではプロセッサの持つ演算能力を最大限活用できるとは限らない．そこで，プロセッサのアーキテキチャに合わせた最適化を行う必要がある．SPRAT では，CPU 用のコードは標準的な C で記述されているため，従来提案されてきた様々な最適化手法を利用可能である．一方，GPU 向けのコードに関しては自動最適化手法が確立されていないため，SPRAT コードから CUDA コードへの変換規則を検討する必要がある．

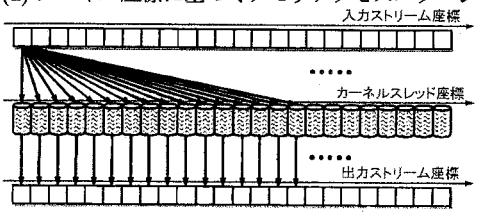
本論文では，SPRAT 言語の記述から GPU のメモリアクセス性能を引き出すことが可能なコードへの変換手法を提案する．提案手法では，各ストリーム要素へのメモリアクセス頻度を推定し，頻繁にアクセスされるストリーム要素を事前に共有メモリに複製することで高速化を実現する．また，アライン境界条件を満足していないメモリアクセスが必要なカーネル関数では，同等のメモリアクセスをアライン境界条件を満足するアクセスの組み合わせで実現することにより，メモリアクセス性能の低下を軽減することが可能である

4.1 アクセス頻度推定によるデータのプリフェッチ

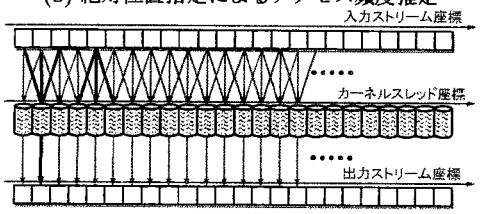
GPU は演算命令の実行時間に対し，グローバルメモリアクセスの遅延時間が非常に大きいため，可能な限りグローバルメモリアクセス回数を削減しなければならない．本手法では，グローバルメモリアクセス回



(a) カーネル座標に基づくメモリアクセスパターン



(b) 絶対位置指定によるアクセス頻度推定



(c) 相対位置指定によるアクセス頻度推定

図 2 アクセス頻度推定

数の削減のために，繰り返し利用されるデータをあらかじめ共有メモリに複製 (プリフェッチ) する CUDA コードを生成する．プリフェッチしたデータを格納する共有メモリの容量は限られているために，プリフェッチ対象データを適切に選択する必要がある．プリフェッチを行う対象データは，再利用回数が多く，グローバルメモリアクセスの回数が多く削減できる要素にするべきであり，適切な対象データを選ぶためにはアクセス頻度の推定が必要である．

SPRAT が定義する拡張プログラミング言語では，カーネル関数内においてアクセス対象要素を定数相対座標や定数絶対座標で指定するために，各要素へのアクセス頻度の推定が他のプログラミング言語と比較して容易である．例として，以下のカーネル関数 `func1` を考える．

```
kernel map func1(in stream<float> x,
                 out stream<float> y)
{
    y = x;
}
```

`func1` では，アクセス対象要素を明示的に指示していないが，暗黙裏にカーネル座標と同じ座標に存在するストリーム要素が参照され，図 2(a) のようなアクセスパターンとなる．この場合，データが再利用されないため，プリフェッチ最適化は行われない．

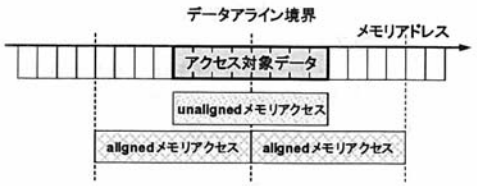


図 3 アラインメントを考慮したメモリアクセス

```
kernel map func2(gather stream<float> x,
                 out stream<float> y)
{
    y = x[[0]];
}
```

`func2` では，インデックスが絶対位置指定となっているため，全てのカーネルスレッドは，自身の座標にかかわらずストリームの特定の要素にアクセスが集中する (図 2(b))．このため，この要素をプリフェッチすることで性能向上を図ることが可能である．

```
kernel map func3(gather stream<float> x,
                 out stream<float> y)
{
    y = x[-1] + x[0] + x[1];
}
```

`func3` では絶対位置指定が存在しないが，定数による相対位置指定によって，図 2(c) のようにカーネルスレッド座標が隣接しているスレッドで重複したデータ範囲をアクセスしている．このような重複して用いるデータもプリフェッチを行い，アクセスを共有メモリへと振り替えることにより，性能向上を図ることが可能である．

以上の解析によって，再利用回数が多いと予測されるデータから優先的にプリフェッチを行うことが可能となる．これにより，容量が制限されている共有メモリを有効に活用できる．

4.2 データアラインメントの影響軽減

GPU では，アライン境界条件を満たさない 1 回のメモリアクセスの方が，アライン境界条件を満たす 2 回のメモリアクセスよりも遅延時間が大きい．したがって，図 3 に示す様に，アライン境界条件を満たしていないデータへのアクセスについては，2 回のアライン境界条件を満たしたアクセスによって共有メモリに一度格納し，共有メモリからデータを取り出すことで遅延時間を短くすることができる この場合．グローバルメモリへのアクセス回数は 2 回に増加し，さらに共有メモリへのアクセスが発生するが，アライン境界条件が満たされていないメモリアクセス遅延時間はこれらの操作の遅延時間以上に大きいため，全体としてはアクセス時間が短縮される．

この方法では，アクセス対象データがアライン境界条件を満たしている場合には，2 度のメモリアクセス

によって性能が低下する．しかし，SPRAT ではプログラマが直接ストリームのメモリ領域を確保することはないため，メモリ領域がアライン境界条件を満たさない可能性があるのは参照ストリームのみである．参照ストリームの宣言時にアライン境界条件を満たしているか否かを判定することが可能であることから，アライン境界条件を満足している場合には 1 回のメモリアクセスしか行わないことで性能低下を回避できる．

5. 性能評価

第 4 章で提案した 2 種類の最適化手法を評価するために，2 種類のサンプルプログラムを実装し，最適化手法を適用しない場合とした場合で，どれほど性能が改善するか評価を行った．評価に用いたサンプルプログラムは，画像のエッジ検出処理プログラム[5] と LU 分解プログラム[6] である．評価環境では，GPU として NVIDIA GeForce 8800GTX を用い，OS として Fedora7(Linux 2.6.23.17-88.fc7)，コンパイラとして GCC 4.1.2 と NVIDIA Cuda compiler driver release 1.1 V0.2.1221 を使用する．

エッジ検出処理[5] は，画像から境界部分を抽出するプログラムであり，計算過程で近傍の要素から中心要素の値を求める処理を行う．近傍の要素に格納されている値を用いて中心要素の値を求める処理は，流体シミュレーションや有限要素法などで広く行われている処理であり，そのようなアプリケーションにおいての効果を見積もるための単純な例として，このプログラムを採用した．隣接する要素の値を用いるため，同じデータを再利用する部分が多く，プリフェッチによる自動最適化が効果を発揮しやすいと考えられる．

LU 分解は，正方行列を上三角行列と下三角行列に分解するプログラムである．連立方程式の解を求める場合や，科学技術演算内部で広く利用されているアルゴリズムであり，計算量が行列サイズ N の三乗に比例する．LU 分解のアルゴリズムでは，演算対象データ範囲を縮小しながら同じ処理を適用するため，データの先頭位置が常に変化し，アライン境界条件を満たすことができない場合が多い．本論文では，特にアライン境界条件による性能への影響を評価するために，可能な限り単純なピボット選択無しの LU 分解アルゴリズム[6] を用いて評価を行う．

5.1 エッジ検出処理による評価

本評価で用いるエッジ検出処理では，図 4 に示す 5 つのカーネルによって処理が行われる．この中で，2 と 3 のカーネルは，隣接ピクセルの値を用いて出力値を算出するカーネルであり，データの再利用が多く発生する．このサンプルプログラムに対し，入力画像サイズを変化させ，単位時間あたりに処理したピクセル数によって性能を評価する．

図 5 に示したのは，エッジ検出処理における実効スループットである．プリフェッチ最適化を行っていない場合 (Not optimized) に比べて，行った場合 (Optimized) では，最大で 2.7 倍の性能向上が得られた．

図 4 エッジ検出処理

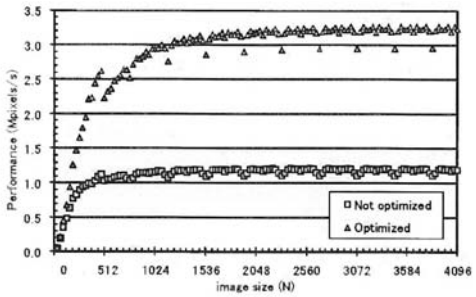


図 5 エッジ検出処理のスループット

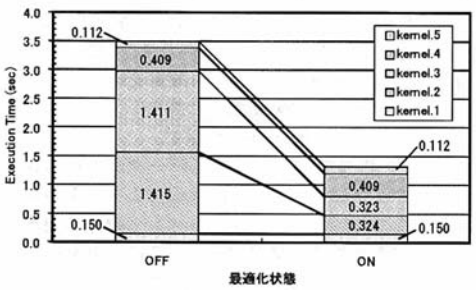


図 6 N=2048 におけるカーネルごとの実行時間

図 6 は，正方画像サイズ N=2048 における，各カーネルの実行時間を示している．データ再利用が多く発生するカーネル 2(平均値フィルタ) およびカーネル 3(Laplacian フィルタ) では プリフェッチ最適化により約 4 倍の性能向上を達成できることが示された．

以上のエッジ検出処理の実験結果から，データ再利用が頻繁に起こる処理においてはプリフェッチ最適化によって大幅な速度向上が達成できることが明らかになった．

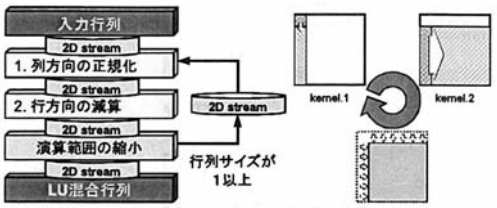


図 7 LU 分解 (ピボットなし)

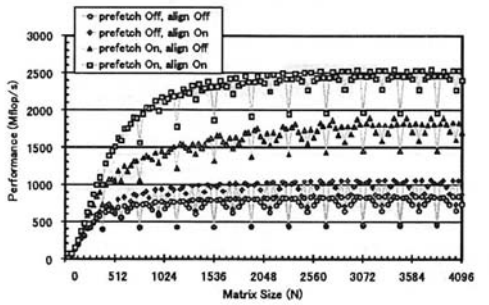


図 8 LU 分解における行列サイズと演算性能

5.2 LU 分解による評価

本論文で実装した LU 分解は，誤差を最小化するためのピボット選択を行わないアルゴリズムを採用した．図 7 に示される LU 分解は，2 つのカーネルから構成される．1 つめのカーネルは入力行列の $(1,1)$ 要素を用いて $(n,1)$ 要素全て $(n \neq 1)$ について除算を行い，正規化を行う．2 つめのカーネルは $(n,1)$ の要素値を (n,m) 要素全て $(m \neq 1)$ の値から減算を行う．その後に演算範囲を行と列方向に 1 ずつ縮小し，サイズが 0 になるまでこれらの操作を繰り返す．

LU 分解は，2 つのカーネルともデータが再利用されるため，プリフェッチによる最適化による効果が期待される．ここではデータアラインメント調整の効果のみを評価するために，プリフェッチ最適化とデータアラインメント調整を独立して適用した．

図 8 に示したのは，行列サイズと演算性能 (Mflop/s 値) を表したものである．この結果では，プリフェッチ最適化によって最大 3.25 倍，データアラインメント調整によって最大 1.27 倍の性能向上を達成した．両方とも適用した場合には最大 4.45 倍となり，これらの最適化手法が最適化手法を適用しない場合に比べて大きな効果を得られることが明らかになった．

6. おわりに

本論文では，SPRAT において抽象的に記述された拡張プログラミング言語から，プロセッサの特性に合わせて最適化を行う手法を考案・実装し，GPU 上で動作するプログラムに対して適用・評価を行った．GPU では，メモリアクセス性能が実行時間に大きな影響を及ぼすため，アクセス頻度推定と共有メモリの活用によるプリフェッチ最適化や，データアラインメントの調整による最適化が有効である．

評価実験の結果，エッジ検出処理に含まれるデータ再利用を多く行うカーネル関数では，プリフェッチ最適化によって約 4 倍の速度向上が達成できることが示された．LU 分解においてデータアラインメントの調整を評価した結果，約 1.27 倍の速度向上を得られ，さらにプリフェッチ最適化と組み合わせることで 4.45 倍に演算速度が改善可能であることが明らかになった．

SPRAT において，抽象的に高位言語で記述されたプログラムから，ハードウェアの潜在性能を引き出すことが可能なコードを自動的に生成可能であることは，プログラムの可搬性と絶対性能を両立させるために極めて重要である．GPU においてはメモリアクセス性能が実行速度に大きな影響を及ぼすが，同時に分岐などのプログラムフロー制御やスレッドブロックサイズも性能に大きな影響を与えることがわかっている．今後さらなる実行性能の向上を得るために，メモリアクセス性能の最適化だけでなく，フロー制御やスレッドブロックサイズの最適化も行っていく予定である．

謝　辞

本研究の一部は，文部科学省科研費若手研究 (B)(19700020)，特定領域研究 (18049003)，および総務省特定領域重点型研究開発 (061102002) の支援を受けている．

参考文献

1) Owens, J. D. et al.: A Survey of General-Purpose Computation on Graphics Hardware, *COMPUTER GRAPHICS forum*, Vol. 23, No.1, pp.80–113 (2007).
2) 滝沢寛之，白取寛貴，佐藤功人，小林広明：SPRAT: 実行時自動チューニング機能を備えるストリーム処理記述用言語，情報処理学会論文誌：コンピュータシステム (ACS23) (2008).
3) NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Architecture programming guide version 1.1 (2007).
4) Buck, I. et al.: Brook for GPUs: Stream Computing on Graphics Hardware, *ACM Trans. Graph.*, Vol.23, No.3, pp.777–786 (2004).
5) 高木幹夫，下田陽久：新編 画像解析ハンドブック，東京大学出版会 (2004).
6) Galoppo, N. et al.: LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, *the ACM/IEEE SC05* (2005).