情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 2　207–220 (Aug. 2008)

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

滝沢寛之[†1]　白取寛貴[†1,*1]
佐藤功人[†1]　小林広明[†2]

本論文では，ストリーム処理記述用言語とその実行時環境との連携により，アーキテクチャ固有の記述を必要とせず，しかも計算システムに搭載されている異種複数のプロセッサの中から実行時に適切なプロセッサを選択する機能を実現する．そのような実行時の自動チューニング機能を実現するために，本論文は比較的容易に実行時間を予測可能なストリーム処理に焦点を絞り，実行時性能予測に基づいてプロセッサを適切に切り替える手法を提案する．本論文では，利用可能なプロセッサとして CPU と GPU を想定し，両者を適切に切り替えることによって，高い抽象レベルで記述されたコードでも両者の特長を生かした高性能計算が可能であることを明らかにする．評価実験の結果より，搭載されている CPU と GPU の性能差に応じて，両者を切り替えることの有効性が示された．また，処理データのサイズに依存して CPU と GPU の演算性能が逆転するという特性を，提案手法では自動的に利用できることが明らかになった．

SPRAT: A Stream Programming Language with Runtime Auto-tuning

Hiroyuki Takizawa,[†1] Hiroki Shiratori,[†1,*1]
Katuto Sato[†1] and Hiroaki Kobayashi[†2]

This paper realizes capabilities to program without any architecture-specific descriptions and also to select an appropriate processor from different processors of a computing system at runtime, by cooperation between a stream programming language and its runtime environment. To realize such a runtime auto-tuning capability, this paper focuses on stream processing whose execution time can be estimated with a simple linear performance model, and proposes a method to switch between different processors based on runtime performance prediction. This paper shows that appropriate switching between CPU and GPU in a PC allows even a code written in a high abstraction level to achieve high-performance computing, which makes use of the characteristics of each processor. The evaluation results demonstrate the effectiveness of switching between CPU and GPU according to their performance difference. The results also show that the proposed method can automatically select an appropriate processor, which may change depending on the data size.

1. はじめに

近年，高い浮動小数点演算性能とメモリバンド幅をあわせ持つ描画処理用ユニット（Graphics Processing Unit，GPU）を描画以外の処理にも利用する研究がさかんに行われている[1)]．現在の一般的な PC は，従来の CPU に加えてプログラム可能な高性能 GPU を搭載している．このため，PC を性能や特性の異なる複数のプロセッサを搭載するヘテロジニアス計算システムと見なすことができる．GPU をデータ並列計算アクセラレータとして効果的に利用することによって，様々なアプリケーションの実行時間を大幅に短縮できることが，これまでに多数報告されている．

しかし，元来 GPU は描画処理に特化して設計されたアーキテクチャであり，もともと描画処理以外の汎用計算を行うために作られてはいない．そのため，GPU を用いた汎用計算（General-Purpose computation on GPUs，GPGPU）のためには，従来の CPU とは異なる複雑なプログラミング技法が必要となる．近年では Compute Unified Device Architecture（CUDA）[2)] や Close-To-Metal（CTM）[3)] といった GPGPU 向けの開発環境も整いつつあるが，それらの環境においても依然として GPU の特殊なアーキテクチャを強く意識したコードを記述する必要がある．このため，CPU のみを使う場合と比較すると，コードの移植性と保守性が著しく低下する可能性がある．

一般的にソフトウェアの寿命はハードウェアのそれよりも長いとされており，特定のハードウェア向けに特化したコードを直接記述するのは好ましくない．このため，GPU を含む複数のプラットホーム向けのソフトウェアを統一的な記述で作成するための開発環境が研究・開発されている[4),5)]．これらの開発環境では CUDA や CTM よりも抽象度の高いレベ

†1 東北大学大学院情報科学研究科
Graduate School of Information Sciences, Tohoku University
†2 東北大学サイバーサイエンスセンター
Cyberscience Center, Tohoku University
*1 現在，新日鉄ソリューションズ株式会社
Presently with NS Solutions Corporation

ルでのソフトウェア開発が可能であり，同一の記述から各種プロセッサ向けのコードが自動生成されるため，各種プロセッサ向けに個別にコードを記述する必要はない．

このように同一の記述から各種プロセッサ向けのコードを自動生成できる場合，利用可能な異種複数のプロセッサの中から各処理を担当するプロセッサを選択可能となる．しかし，各プロセッサは異なる特性や性能を持ち，その結果として処理の得手不得手がある．特にGPU は得手不得手の差が大きく，GPU に対して不得手な処理が割り当てられた場合には，アプリケーション全体として深刻な性能低下を引き起こす可能性がある．また，適切なプロセッサの選択は，それぞれのプロセッサの得手不得手だけでなく，計算システムに搭載されている他のプロセッサとの性能差や処理されるデータのサイズなど，実行時に定まるパラメータにも強く依存している．このため，これらのパラメータが定まる以前に，プログラマやコンパイラが各処理に対して適切なプロセッサを指定するのは困難である．

本論文では，GPU による高速化が期待できるストリーム処理に着目する．C 言語にストリーム処理を記述するための拡張文法を加え，その拡張文法で書かれた部分に関しては自動的に担当プロセッサを判断することを可能にする言語処理系およびその実行時環境としてStream Programming with Runtime Auto-Tuning（SPRAT）を提案する．適切な担当プロセッサの選択はハードウェア構成やデータサイズに強く依存しており，多くの場合これらはコンパイルの時点では未知である．このため，SPRAT は実行時にデータサイズに基づいて各プロセッサによる実行時間を予測し，自動的に担当プロセッサを調整する実行時自動チューニング機能[6] を備えている．

2. 関 連 研 究

2.1 プログラム開発言語/ツール/ライブラリ

従来の GPU を汎用計算に用いる研究（General-Purpose computation on GPUs, GPGPU）では，OpenGL や DierctX などの描画用 API（Application Programming Interface）を介して GPU が利用されてきた．GPU 上での処理の記述には当初アセンブリ言語が用いられ，後に C for graphics（Cg）や High-Level Shader Language（HLSL），OpenGL Shader Language（GLSL）などの高級言語も登場している[1]．しかし，これらで記述できるのはあくまでも描画処理の一部であり，対象アプリケーションの計算を描画処理として記述することで，間接的に GPU の演算能力が利用されてきた．

Buck らは GPU を汎用ストリームプロセッサとして抽象化し，プログラマが描画用 API を意識することなくストリーム処理を記述できるプログラミング言語として BrookGPU を提案している[7]．BrookGPU は C 言語にいくつかの予約語を追加した言語であり，それらの予約語を使って書かれたコードの一部が Brook コンパイラによって描画用 API で記述されたコードに変換される．BrookGPU では実行時バックエンド（runtime backend）を環境変数 `BRT_RUNTIME` によって事前に静的に指定することが可能であり，BrookGPU version 0.4 では CPU，OpenGL，DirectX の 3 つのうちいずれかの実行時環境を指定できる．ただし，BrookGPU では CPU での実行を指定できるものの，言語仕様が GPU での実行に合わせて考えられているため，CPU での高速な実行は期待できない．また，GPU 上での実行においても，抽象化のための冗長な処理に起因するオーバヘッドがあるため，描画用 API を使って直接記述された GPGPU アプリケーションよりも性能が低くなる場合が多い．さらに，プログラムの実行開始時に環境変数によって担当プロセッサが定められ，その後の切替えはできないため，プログラムを構成するそれぞれのカーネルに対して適切なプロセッサを個別に割り当てることはできない．

近年，GPU のハードウェアをより柔軟に制御し，さらに GPGPU における描画用 API のオーバヘッドを排除することを目的として，CUDA や CTM が GPU ベンダから提供されている[2),3]．これらは OpenGL や DirectX よりも低レベルで GPU のハードウェアへのアクセスを可能にしており，GPU の持つ演算性能をより効率的に引き出すことができる．しかし，抽象度が低いために GPU のアーキテクチャ，仕様変更や拡張の影響を受けやすい．

これまでに，抽象度の高いレベルでのプログラミングを可能とするための研究開発もなされている[4),5),8]．それらの開発環境やライブラリでは，GPU の存在を意識することなく GPU のすぐれた演算能力を利用可能である．また，処理を担当するプロセッサを明示的に指定する手段が提供されている開発環境もある．しかし，性能予測に基づいて自動的に適切なプロセッサを選択する機能はない．

ほかにも，データ並列処理を適切な割合で分割し，それらを CPU と GPU で分担して実行することによって両者の演算能力を同時利用する研究[9] や，GPU による行列積の自動チューニング[10] も報告されている．しかし，その適用範囲は行列積などの特定の処理に限定されており，言語処理系を対象とする本論文とは目的が異なっている．

2.2 性能モデリング

GPU を汎用計算に用いるうえで，その実効性能を評価するための研究がいくつか行われている．

Buck らは GPU の汎用計算における性能を計測するための GPUBench を開発・公開し，GPU の実効性能がほぼつねに GPU とビデオメモリ間のデータ転送の実効バンド幅によっ

て決定されることを指摘している[11]．さらに，Buck らは文献 7) において，CPU および GPU によるストリーム処理の実行時間を一次式でモデル化し，GPU 上での実行によって処理の高速化が期待できる条件について議論している．

Trancoso ら[12] は，GPU の演算性能がプログラムの特徴に依存していることを実験的に調査した結果を報告している．プログラム中の演算命令の数や入力データのサイズ，入力データの種類の変化によって，GPU の計算性能が変化することを明らかにしている．ただし，性能評価には BrookGPU を用いており，単純に環境変数 BRT_RUNTIME の値の変更によって CPU と GPU を切り替えているため，前述のように特に CPU の演算性能が不当に低く評価されている．

伊藤ら[13] は，GPGPU アプリケーションの実行時間の予測方法を提案している．GPU 固有の性能パラメータとして，テクスチャデータのメインメモリからビデオメモリへの転送（ダウンロード）と，ビデオメモリからメインメモリへの転送（リードバック），およびビデオメモリとフラグメントプロセッサ間の転送のそれぞれについて実効バンド幅と立ち上り時間（startup time）を事前に計測することで，典型的な GPGPU アプリケーションの実行時間を予測している．しかし，彼ら自身も言及しているとおり，実効バンド幅はメモリアクセスパターンに強く依存しているため，GPGPU アプリケーションのアクセスパターンが事前の計測時と大きく異なる場合には，性能予測が大きく外れる可能性がある．この性能依存性を考慮し，He らは GPU の実効メモリバンド幅を逐次アクセスとランダムアクセスの場合に分けてモデル化している[14]．

3. 実行時自動チューニング機能を備えるストリーム処理用言語

本論文では，GPU による大幅な高速化を期待できるストリーム処理を明示的に記述するために C 言語の拡張文法を定義し，さらにその処理を担当するプロセッサを自動的に選択する機能を備えるプログラミング言語として Stream Programming with Runtime Auto-Tuning（SPRAT）を提案する．ストリームの各要素は相互に依存せず，独立に処理される．このためストリーム処理には，ハードウェア構成をプログラマに公開することなく，並列実行可能なコードを記述できるという特徴がある[15]．

3.1 ストリーム処理記述用拡張文法

GPU のように性能を発揮できるパターンが限られているプロセッサを対象として高性能計算アプリケーションを開発することを考え，SPRAT 言語ではプログラム記述の自由度をあえて制限し，性能を発揮できる処理のみを記述できる構文を採用する．このことによって，最適化時のパラメータ探索の労力を軽減するとともに，実行時性能予測を容易にしている．SPRAT 言語では特に BrookGPU の文法や予約語を参考にしているが，CPU 上での高速実行の障害となる仕様を避け，CPU と GPU の効果的な使い分けを目指している．

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

int main(int argc, char** argv){
  stream<float> sX(N,M), sY(N,M), sZ(N,M);
  float x[N*M], y[N*M], z[N*M], pi=3.14f;

  init_array( x, y);
  streamRead(sX, x);
  streamRead(sY, y);
  saxpy(pi, sX, sY, sZ);
  streamWrite(sZ, z);
  print_array(z);
  return 0;
}
```

図 1　SPART 言語のサンプルコード
Fig. 1　Sample code of the SPRAT language.

SPRAT 言語では，ストリーム処理の対象となるデータ集合全体（**ストリーム**）を stream 型変数に保持する．ストリームの各要素に対する処理の内容は，修飾子 kernel を付けて定義される**カーネル関数**内に記述される．ストリームは並列に処理可能なデータの集合であり，カーネル関数内でのみ各要素にアクセス可能である．このような構文は他のストリーム処理記述言語でも採用されており[7]，ストリーム処理の記述のための標準的な構文であるといえる．

図 1 に SPRAT 言語によって記述されたサンプルコードを示す．このサンプルコードでは，2 つの入力ストリーム sX と sY と 1 つのスカラ値 pi がカーネル関数 saxpy に渡され，pi 倍された sX の要素と sY の要素との和が出力ストリーム sZ の各要素に代入される．カーネル関数外ではストリームの各要素にアクセスできないため，関数 init_array で値を設定された配列 x と y を，標準関数 streamRead を使って stream 型変数 sX と sY に

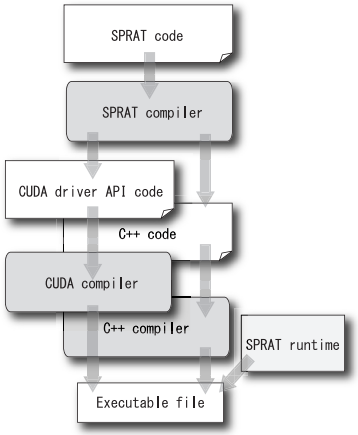

図 2　SPRAT 言語の変換
Fig. 2　SPRAT language translation.

複製している．同様に，ストリーム処理の結果を保持する sZ が，streamWrite を使って配列 z に複製されている．

SPRAT 言語から実行ファイルに変換される過程を**図 2** に示す．プログラマによって記述された SPART 言語のソースコードは，SPRAT コンパイラ（トランスレータ）によって標準的な C++言語と GPU 向けのコード[⋆1]に変換される．このようにすることで，特に高速化が求められる処理をハードウェアを意識して手動で最適化する余地を残している．C++および GPU 向けのコードは，それぞれのコンパイラによってオブジェクトコードに変換されたのち，SPRAT 実行時ライブラリやその他必要なライブラリとリンクされ，最終的に実行ファイルが生成される．

以下では，SPRAT 言語の stream 型とカーネル関数（kernel map）について主な役割と機能を述べる．

3.1.1　stream

SPRAT 言語では，ストリームデータのコンテナとして stream 型を提供している．この stream 型変数の宣言時に，各要素の変数型とストリームに含まれる要素数を指定する．要素数を複数指定することによって，多次元ストリームを宣言することも可能である．

ストリームへのデータ入出力のために，関数 streamRead と streamWrite が定義されている．前者は配列の各要素をストリームの各要素へと代入する．逆に後者はストリームの各要素を配列の各要素へと代入する．

カーネル関数の仮引数で指定可能な予約語として in，out，inout，および gather が用意されている．ここで in はストリームを読み取り専用の入力ストリームとして扱うための修飾子である．デフォルトで stream 型の引数は読み取り専用として扱われる．一方，out や inout は stream 型引数を書き込み専用および読み書き可能の出力ストリームにするための修飾子である．後述のように，配列のように添え字を使って stream 型データにランダムアクセスしたい場合には gather を指定する．

ほかにもストリームデータの任意の範囲を切り出すための文法が提供されており，

```
stream<float>& ref = strm[i][j](w,h);
```

と記述することで (i,j) を始点とし，幅 w 高さ h の領域を切り出すことができる．ただし，ref は strm の一部を示す参照変数であり，ストリーム要素用のメモリ空間を strm と共有している．

また，ストリーム中の最大要素を返す streamMax や最大要素の添え字を返す streamMaxi などストリームに対して多用される処理については，標準関数が用意されている．

3.1.2　kernel

SPRAT では，ストリーム処理の内容を記述するための特別な関数としてカーネル関数を提供している．このカーネル関数の実行を担当するプロセッサは，SPRAT の実行時環境によって実行時に選択される．カーネル関数内には個々のストリーム要素への処理内容のみが記述されており，各要素の処理される順番を制御することはできない．

現在の SPRAT 言語では，図 1 に示すとおり，カーネル関数の属性として map か reduce のいずれかを予約語 kernel の後に指定する．前者は入力ストリーム要素間の演算から，出力ストリームの各要素を算出する．後者は，入力ストリーム中の各要素をよりサイズの小さい出力ストリームやスカラ値へとリダクションする．

属性として map を指定されたカーネル関数は，引数として out 修飾子あるいは inout 修飾子のついたストリームを必ず 1 つ以上持つ．その出力ストリームのサイズに合わせて，データ並列処理が実行される．たとえば，入力ストリームは出力ストリームと同じサイズである場合，図 1 の saxpy は SPRAT コンパイラによって以下の処理を意味するコードに暗黙のうちに変換される．

⋆1 本論文執筆時点の実装では NVIDIA 社の CUDA driver API を用いて記述されたコードのみをサポート．

```
for(int i=0; i<M; i++)
  for(int j=0; j<N; j++)
     z[i*N+j] = a*x[i*N+j] + y[i*N+j];
```

ただし，各ストリーム要素が処理される順序は保証されておらず，処理される順序を仮定したコードの動作は未定義である．

上記の対応関係とは異なる要素間の演算を行いたい場合には，gather ストリームを用いる．gather 修飾子が付加されたストリームでは，配列のように添え字を使ってストリーム要素にアクセス可能である．通常は，対応する出力ストリーム要素の座標からの相対座標で入力ストリームの添え字を指定する．たとえば，gather ストリーム strm に対して strm[0][0] と記述されている場合には，in ストリームの場合と同じ位置の要素を参照し，strm[-1][0]，strm[1][0]，strm[0][-1]，strm[0][1] であれば，それぞれ 2 次元配列上の上下左右の隣接ストリーム要素を参照する．ストリーム処理では隣接要素を参照する機会が多いため，このような位置指定方法を標準としている．一方，2 次元ストリーム strm の i 番目の行，j 番目の列の要素を参照するためには strm[[i]][[j]] と記述する．後述の LU 分解のコード（図 4 参照）のように，相対位置指定と絶対位置指定を組み合わせてストリーム要素を指定することも可能である．

3.2　実行時自動チューニング

多くの場合，ストリーム処理の速度はメモリバンド幅に律速されるため，その実行時間は処理対象となるストリームサイズに対して比例する傾向にある．特に map では，出力ストリームのサイズによって演算回数が定まり，しかも入力ストリームサイズも出力ストリームのサイズに合わせて定まることから，カーネル中でアクセスする全ストリーム要素数および処理回数ともに出力ストリームサイズに応じて増加する．このため，map カーネルの実行時間も出力ストリームサイズに比例する傾向が強く，線形近似によって比較的容易に予測することができる．

SPRAT では，データ転送時間およびカーネル実行時間の双方を線形近似によって予測する．それぞれ切片（立ち上り時間）と傾き（実効バンド幅あるいはスループット）を事前に計測し，実行時に任意のデータサイズの実行時間予測に用いる．実行時間予測に用いる性能パラメータの一覧を**表 1** に示す．表中の p および q はそれぞれプロセッサ（CPU あるいは GPU）を表し，k_i は i 番目のカーネル関数を表している．

3.2.1　データ転送時間予測

データサイズを変化させた場合の，CPU から GPU へのデータのダウンロード時間と，GPU から CPU へのデータのリードバック時間を測定し，性能予測のための性能パラメータを決定する．2 種類のプロセッサ p と q を搭載するシステムにおいて，p から q へのデータ転送のバンド幅 $B_{p,q}$ と立ち上り時間 $S_{p,q}$ を性能パラメータとして用いることにより，任意のデータサイズの転送時間を予測する．データ転送時間 $T_{p,q}$ は，式 (1) で定義される．

$$T_{p,q} = D_{p,q}/B_{p,q} + S_{p,q} \tag{1}$$

ここで $D_{p,q}$ は，プロセッサ p からプロセッサ q へ転送されるデータサイズである．逆方向のデータ転送に関しても，実効バンド幅 $B_{q,p}$ と立ち上り時間 $S_{q,p}$ を事前に計測し，同様の線形モデルによって転送時間が予測される．これらデータ転送に関する性能パラメータは，システム固有のパラメータとして言語処理系のインストール時に計測される．また，線形近似には最小二乗法が用いられる．

表 1　性能パラメータ
Table 1　Performance parameters.

Symbol	Description
$B_{p,q}$	Data transfer bandwidth from p to q.
$S_{p,q}$	Data transfer startup time from p to q.
B_{p,k_i}	Throughput of p for execution of k_i.
S_{p,k_i}	Startup time of p for execution of k_i.

SPRAT 言語では stream 型変数の宣言時にストリームサイズが指定される．ストリームサイズの指定が定数式とは限らないため，実際のストリームサイズが定まるのは実行時である．実行時にストリームサイズ（式 (1) 中の $D_{p.q}$ や $D_{q,p}$）が定まった後に，データ転送時間が式 (1) に従って予測される．

現在の SPRAT の実装では，実行を担当するプロセッサ側のストリームデータが最新であることをカーネル実行開始時に確認し，最新でない場合にのみストリーム全体が転送される．つねにストリーム全体を転送するために不要なデータ転送が発生する可能性も考えられるが，必要なストリーム要素のみを選択的に転送するために必要な実行時コストも高いことが予想されるため，現在は一律にストリームデータ全体を転送するように実装されている．

3.2.2　カーネル実行時間予測

多くの場合，カーネルの実行時間は出力ストリームのサイズ D_o に比例して増加する．しかし，そのスループットや立ち上り時間はカーネル中のメモリアクセスパターンや演算回数に依存する．このため，SPRAT ではプロセッサ p による i 番目のカーネル実行時のスループット B_{p,k_i} と立上り時間 S_{p,k_i} を記録し，以後の実行時の予測に利用する．すなわち，

カーネルの実行時間は式 (2) によって予測される．

$$T_{p,k_i} = D_{k_i}/B_{p,k_i} + S_{p,k_i} \quad (2)$$

ここで p は CPU か GPU のいずれかを示しており，CPU でのカーネル実行時間および GPU でのカーネル実行時間ともに，式 (2) によってモデル化される．また，標準では $D_{k_i} = D_o$ であり，カーネル実行時間は出力ストリームサイズに比例すると仮定して，実行時間の予測が行われる．

カーネル実行時間の予測に必要な性能パラメータ B_{p,k_i} と S_{p,k_i} は，カーネルを複数回実行し，それらの実行時間から最小二乗法によって算出される．実行時間を計測するための問題サイズに関しては，計測時にアプリケーション利用者から典型的な値が与えられるものと仮定している．カーネルの実行時間を計測するためにはそのカーネルの実行後に CPU と GPU の同期が必要であり，計測時には同期のオーバヘッドにより実効性能が低下する．このため，計測回数の増加によって予測誤差が十分小さくなった以降はそれ以上実行時間を計測せず，それまでに計測されたデータから算出された B_{p,k_i} と S_{p,k_i} に基づいて実行時間を予測する．

現在の SPRAT の実装では，最初の n 回のアプリケーション実行時に実行を担当するプロセッサを固定して，CPU および GPU の性能パラメータを計測し，その結果をデータベースに保存する．標準では $n = 10$ であり，CPU と GPU の性能パラメータがそれぞれ 5 回のアプリケーション実行における計測データから算出される．それ以降の実行時には，データベース中の性能パラメータを参照して，カーネル実行時間を予測する．また，予測誤差が十分小さくなるまでカーネル実行時間を計測するための手段が提供されており，アプリケーション利用者はデータベース情報を専用のビューアによって確認し，各アプリケーションの性能パラメータを管理することができる．

3.2.3　プロセッサ選択

CPU と GPU はそれぞれ独自のメモリ空間を持っており，CPU によるカーネル実行ではメインメモリ上のデータ，GPU によるカーネル実行ではグラフィックカードに搭載されているビデオメモリ上のデータが使われる．このため，カーネルの実行を担当するプロセッサを切り替えるためには CPU-GPU 間のデータ転送が必要となる．この CPU-GPU 間のデータ転送にはある程度の時間を要するため，安易なプロセッサ切替えはアプリケーション全体の性能低下につながる．したがって，転送のオーバヘッドを考慮して適切なタイミングで担当プロセッサを切り替える必要がある．

ここで 2 種類のプロセッサ p と q を搭載するシステムがあり，あるカーネル k_i を実行する場合には p の方が速いことを仮定する．また，現在はプロセッサ q が実行を担当しているものとする．カーネル k_i を 1 回だけ実行するならば，次式が成り立つ場合のみ，担当プロセッサを p に切り替えるべきである．

$$T_{q,p} + T_{p,k_i} < T_{q,k_i} \quad (3)$$

しかし，多くの場合，カーネル実行時間と比較してデータ転送に要する時間は長いため，式 (3) が成立するのは稀である．個々のカーネル単位で考えると，他方のプロセッサ p へデータを転送してからそのカーネルを実行するよりは，現在のプロセッサ q で実行を続ける方が実行時間が短いと判断され，プロセッサの切替えは発生しなくなる．

一方，プロセッサ p で実行する場合と比較すると，プロセッサ q でカーネル k_i を実行するたびに $T_{q,k_i} - T_{p,k_i}$ だけ長い実行時間を要していることになる．この実行時間差の累積（**累積実行時間差**と呼ぶ）がデータ転送に要する時間よりも長くなる場合には，担当プロセッサを p に切り替える方が結果的にアプリケーションの実行時間は短くなる．つまり，次式を満足する N 以上のカーネル実行回数ならば，プロセッサを p に切り替えた方が実行時間を短くすることができる．

$$T_{q,p} < \sum_{i=1}^{N} (T_{q,k_i} - T_{p,k_i}) \quad (4)$$

ただし，式 (4) で必要となる T_{q,k_i} や T_{p,k_i} を式 (2) に基づいて予測するためには D_{k_i} が必要となるが，将来のカーネル呼び出しで引数となるストリームのサイズを求めるのは容易ではなく，不可能である場合もある．

SPRAT ではプログラム中に周期性があることを仮定し，過去の累積実行時間差に基づいてプロセッサ選択を行う．過去の累積実行時間差がデータ転送時間を超過している場合には，将来の実行においてもデータ転送時間以上の累積実行時間差が生じるものと判断し，プロセッサを切り替える．現在プロセッサ q が実行を担当しているものとし，カーネル k_i を実行するプロセッサを決定する手順を以下に示す．

(1)　式 (1) および式 (2) に基づき，$T_{q,p}$，T_{q,k_i} および T_{p,k_i} を予測する．

(2)　プロセッサ p での実行を仮定した場合の累積実行時間差 τ_p を次式に従って更新する[⋆1]．

⋆1 累積実行時間差 τ_p は仮にプロセッサ p で実行していたらどれくらい実行時間を短縮できていたのか，と表す数値と考えることができる．

$$\tau_p \leftarrow \max\{\tau_p + (T_{q,k_i} - T_{p,k_i}), 0\} \quad (5)$$

継続的に $T_{q,k_i} - T_{p,k_i} < 0$ の場合には累積実行時間差が負になり，傾向が変化した場合に迅速に対応できないことから，τ_p はつねに 0 以上の値となるように更新される．

(3)　$\tau_p > T_{q,p}$ の場合，必要なストリームデータをプロセッサ p 側に転送し，カーネル k_i をプロセッサ p で実行する．このとき，プロセッサ q 実行時の累積実行時間差 τ_q は 0 にリセットされる．$\tau_p \leq T_{q,p}$ の場合には，カーネル k_i をプロセッサ q で実行する．

つねに速い方のプロセッサを利用する理想的なプロセッサ選択と比較すると，本手法はプロセッサの切替えの判断までに $T_{q,p}$ の余分な実行時間を要するが，カーネル実行回数が十分に多い場合にはこのオーバヘッドは無視できる．複数種類のカーネル関数の実行を繰り返している場合には，すべての種類のカーネル関数での実行時間差を累積するため，式 (4) の k_i（$i = 0, 1, \ldots$）がそれぞれ異なる種類のカーネル関数であっても本手法を適用可能である．

4. 性能評価と考察

4.1 実験条件

Intel Core 2 Quad Q6600 2.4 GHz（以下 C2Q とする）と DDR2 DRAM 4 GB を搭載する PC と，**表 2** に示す GPU を用いて性能評価のための実験を行う．表 2 で，# SMs は GPU が搭載しているストリーミングマルチプロセッサ数[⋆1]，Mem はビデオメモリ容量，BW は理論最大メモリバンド幅を示している．また，GF88GTX は NVIDIA GeForce 8800 GTX の省略形であり，他の型番に関しても同様の省略形で記載されている．実験用 PC には OS として Fedora Core 7 がインストールされており，GPU のドライバのバージョンは 169.09 である．

表 2　評価に用いる GPU の仕様
Table 2　Specifications of GPUs used for evaluation.

Model	# SMs	Mem[MB]	BW[GB/s]
GF88GTX	16	768	86.4
GF88GT	14	512	57.6
GF86GTS	4	256	32.0
GF85GT	2	256	12.8
GF84GS	2	256	6.4

⋆1 ストリーミングマルチプロセッサは，それぞれ 8 基のストリーミングプロセッサから構成されている．

現在の SPRAT のコンパイラおよび実行時ライブラリの実装は C++言語で行われており，SPRAT 言語で記述されたコードから CPU 用の C++コードと NVIDIA 社製 GPU 用の CUDA driver API で記述されたコードが自動生成される．自動生成されたコードのコンパイル時には，C++コンパイラとして gcc-4.2.1，CUDA コンパイラとして nvcc-release 1.1 V0.2.1221 を用いた．

4.2 予備実験

提案手法では，CPU および GPU におけるストリーム処理実行時間が線形近似によって予測されている．これは，ストリーム処理が多くの場合メモリバンド幅に律速されるためである．しかし，B_{p,k_i} および S_{p,k_i} の値はメモリアクセスパターンや，メモリアラインメント，各ストリームマルチプロセッサへ割り当てるスレッド数など様々な要因で大きく変化する．例として，ビデオメモリへのアクセスが逐次の場合（sequential），ランダムの場合（random），およびアラインメントが GPU の高速メモリアクセスのための要求要件を満たしていない逐次アクセスの場合（non-coalesced）について，ストリームサイズが実行時間に与える影響を評価した結果を**図 3** に示す．本予備実験では GF88GTX を使い，CUDA の用語でグローバルメモリ[2] と呼ばれるメモリ領域へのアクセスに要する時間をアクセスパターンを変化させながら計測した．この結果から分かるとおり，実行時間は様々な要因によって大きく変化する．このため，性能予測に必要な性能パラメータもカーネル関数ごとに計測する必要があることが分かる．

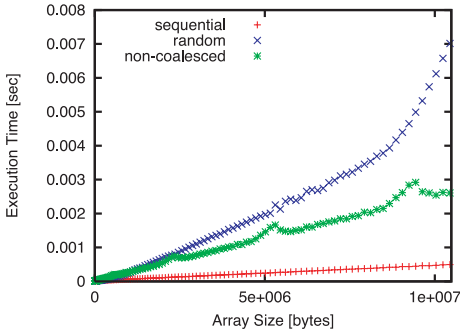

図 3　メモリアクセスパターンと実行時間
Fig. 3　Memory access pattern and the execution time.

4.3 性能評価

本節では，LU 分解および非圧縮流体シミュレーションを用いて提案手法の有効性を評価する．アプリケーション実行開始時には CPU が実行を担当しているものとし，その状態から実行時性能予測に基づいて担当プロセッサを切り替える．

本実験では，事前に 2 種類のアプリケーションをそれぞれ問題サイズ 32，64，128，256，512 で実行して各カーネル関数の実効性能を計測し，算出した性能パラメータを用いる．また，CPU-GPU 間のデータ転送の実効バンド幅と立ち上り時間も事前に計測し，データ転送時間の予測に用いる．カーネル実行時間予測のための性能パラメータは GPU の型番によって大きく変化するのに対して，GPU の性能差は CPU-GPU 間のデータ転送時間にほとんど影響を与えない．事前計測の結果から最小二乗法によって算出した CPU から GPU へのデータ転送の実効バンド幅は約 1.85 [GB/s] であり，立ち上り時間は約 0.06 [msec] であった．同様に GPU から CPU へのデータ転送では実効バンド幅が 0.58 [GB/s]，立ち上り時間は 0.20 [msec] であった．

4.3.1 評価用アプリケーション

本実験で用いる LU 分解は，GPGPU 用のピボットなし LU 分解のコード[16] に基づいて実装されており，`rowop` および `normalize` という 2 種類のカーネルをストリームサイズを減少させながら繰り返し実行する．**図 4** に LU 分解を SPRAT 言語で記述したコードを示す．ストリームの先頭アドレスが変化し，多くの場合，GPU が高いメモリバンド幅を実現するために必要なメモリアラインメントの条件を満たさなくなるため，図 3 中の non-coalesced のように実効バンド幅が著しく低下する．すなわち，意図的に GPU が性能を発揮しにくいアプリケーションとなるように実装されている．このアプリケーションを用いて評価する目的は，GPU にとって不適切な実装が行われた場合に GPU が低速となり，CPU が担当プロセッサとして適切に選択されることを確認することである．

SPRAT コンパイラによって図 4 の SPRAT コードを C++および CUDA のコードに変換した結果の一部を付録に示す．GPU で実行される場合，出力ストリームの要素数だけスレッドが起動し，それぞれが `__global__` 関数を実行することによって CPU のカーネル関数の `for` ループと同一の処理を実現する．

一方，本実験で用いる流体シミュレーションは部分段階法[17] によって二次元流体をシミュレートする．多くのカーネル関数がメモリバンド幅に律速されるため，CPU よりも GPU の方が性能を発揮しやすいアプリケーションとなっている．また，連立一次方程式の解を求めるために反復法（ヤコビ法）を用いており，収束するまで 3 種類のカーネルを繰り返し

```
kernel map rowop(gather stream<float> gath, out stream<float> ostr){
  ostr=gath[0][0]-gath[[-1]][0]*gath[0][[-1]];
}

kernel map normalize(gather stream<float> gath, out stream<float> ostr){
  ostr=gath[0][0]/gath[[-1]][0];
}

int main(int argc, char ** argv){
  stream<float> str(N,N);
  float origMat[N*N];
  int i;

  // -- (snip) --
  streamRead(str,origMat);
  for(i=0;i<N-1;i++){
     stream<float>& s=str[i][i+1](1,N-i-1);
     normalize(s, s); /// kernel invocation
     stream<float>& s=str[i+1][i+1](N-i-1,N-i-1);
     rowop( s, s); /// kernel invocation
  }
  // -- (snip) --
  return 0;
}
```

図 4　SPART 言語による LU 分解
Fig. 4　LU decomposition written in the SPRAT language.

実行する．GPU でこれらのカーネルを実行している場合には，GPU で計算された誤差を CPU 側に転送して収束判定を行っている．

本実験で使用する流体シミュレーションのコードは中間速度の計算，中間速度の発散の計算，圧力場の計算，および次の時間ステップでの速度場の計算のために，以下の 8 種類のカーネル関数から構成されている．ここで，() 内はカーネル関数名である．

(1)　x 方向の中間速度の計算（`calc_u_aux`）
(2)　y 方向の中間速度の計算（`calc_u_aux`）
(3)　中間速度の発散の計算（`calc_div`）
(4)　次の時間ステップの圧力を求める反復法の計算（`calc_p_next`）
(5)　反復法による圧力場の変化量の計算（`calc_p_error`）

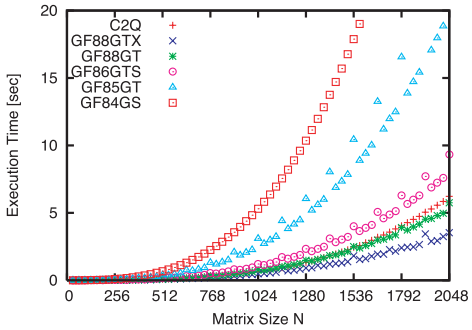

図 5　LU 分解の実行時間
Fig. 5　Execution time of LU decomposition.

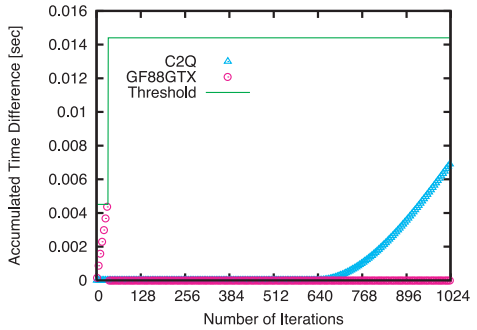

図 6　累積実行時間差の変化
Fig. 6　Changes in the accumulated time difference.

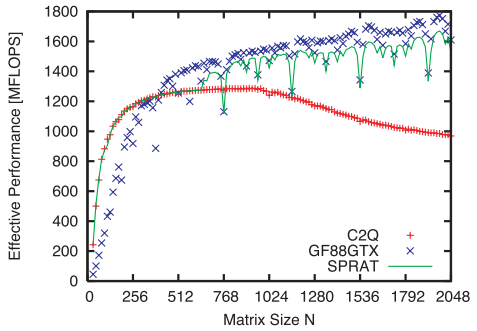

図 7　行列サイズと実効性能の関係
Fig. 7　Relationship between matrix size and effective performance.

(6)　反復法による変化量の最大値を探索（`streamMax`）し，その値が閾値を超えていれば手順 (4) に戻る．
(7)　次の時間ステップにおける x 方向の速度の計算（`calc_u_next`）
(8)　次の時間ステップにおける y 方向の速度の計算（`calc_v_next`）

4.3.2　性能評価結果と考察

行列の 1 辺の要素数（N）とその LU 分解の実行時間を**図 5** に示す．先述のとおり図 4 の LU 分解を CUDA 環境で高速に実行することができないため，GPU による実効性能が非常に低くなっており，その結果として性能の低い GF86GT，GF85GT，および GF84GS はつねに C2Q よりも低速であった．これらの GPU を用いて LU 分解を実行する場合には，SPRAT ではつねに C2Q が用いられる．この結果から，GPU にとって不得手な処理がカーネル関数として記述された場合に，提案手法が CPU を適切に選択できることが示された．

LU 分解の実行時間ではカーネル関数 `rowop` の実行時間が支配的である．図 4 に示されるとおり，`rowop` は `for` ループ中で $(N-1)\times(N-1)$ から 1×1 までストリームサイズを変化させながら合計 $N-1$ 回実行される．C2Q と GF88GTX の組合せの場合，ストリームサイズの大小によって両者の性能が逆転するため，$N-1$ 回の実行途中でプロセッサの切替えが起こる．行列サイズ $1{,}024\times 1{,}024$ の LU 分解における累積実行時間の変化を**図 6** に示す．横軸はループ回数である．アプリケーション開始時に C2Q が実行を担当しているが，GF88GTX の方がカーネル実行時間の予測値が短いため，累積実行時間が実行開始直後から急速に増加し，CPU から GPU へのデータ転送時間の予測値 4.51×10^{-3} [sec] を超えたときに実行担当プロセッサが GF88GTX に切り替わっている．このとき，プロセッサ切替えの閾値も，GPU から CPU へのデータ転送時間の予測値 1.44×10^{-2} [sec] へと変化する．ストリームサイズが小さい場合には C2Q の方が高速であるために，途中から累積実行時間が再度増加し始めるが，GPU から CPU へのデータ転送時間と比較するとその速度差は小さいため C2Q への切替えは起こらない．

それぞれの行列サイズに対して C2Q，GF88GTX，および提案手法で達成した FLOPS 値を**図 7** に示す．この図から分かるとおり，提案手法はプロセッサを適切に使い分けることによって，各サイズに対して速い方のプロセッサを使った場合と同等の性能を達成できている．行列サイズ 32×32 の LU 分解の場合，GF88GTX が C2Q のおよそ 5.4 倍の実行時

間を必要とし，逆に行列サイズ 2,048 × 2,048 の場合には C2Q が GF88GTX のおよそ 1.6 倍の実行時間を必要とする．提案手法が行列サイズに応じて両者を切り替えることで，そのような大幅な速度低下を回避できており，実行時にプロセッサを動的に切り替えることの有効性が明らかになった．ただし，大きい行列に対して GF88GTX のみで LU 分解を実行した場合と比較すると，提案手法の性能は若干低くなっている．これは，データサイズが大きいところでは GF88GTX の方が速いにもかかわらず，図 6 に見られるように実行開始時から累積実行時間差がデータ転送時間の予測値を上回るまで C2Q が実行を担当しているためである．したがって，事前の性能予測あるいはアプリケーション利用者からの指示によって実行開始時のプロセッサを適切に決定することができれば，提案手法のさらなる性能向上が期待できる．

なお，GF88GT の場合には C2Q との性能差が小さいため，GF88GT の方が速いにもかかわらずプロセッサの切替えが起こらなかった．これは性能予測の誤差によって適切な判断ができなかったためである．GPU の実行時間計測時には CPU と GPU との同期のために，通常よりも実行が遅くなる．この誤差により，GF88GT の場合には適切なプロセッサが選択できなかったものと考えられる．一般的に，同期オーバヘッドによって GPU の性能が過小評価されている．一方，キャッシュの影響を考慮できていないため，CPU の性能が過大評価される傾向がある．CPU と GPU の性能差が僅差の場合の適切な判断のためには，キャッシュの影響や同期オーバヘッドの補正などによる性能予測精度の改善が必要不可欠であり，今後の課題である．

同様に，二次元グリッドの一辺のグリッド点数（N）と流体シミュレーションの実行時間（100 時間ステップ分）の関係を**図 8** に示す．流体シミュレーションの場合，CPU と GPU の性能の逆転はグリッドサイズが 128 × 128 以下の小さいところで起こっている．それ以上のサイズの場合には，個々のカーネルにおいて多少の性能逆転が見られるものの，アプリケーション全体としては GF84GS を除くすべての GPU が C2Q よりも高速であった．提案手法もほぼその性能逆転のとおりにプロセッサを切り替えることができていることが確認できた．これらの結果より，複数のカーネル関数から構成される流体シミュレーションにおいても，提案手法が適切に実行を担当するプロセッサを選択できていることが明らかになった．

本実験では，ストリームサイズが比較的小さなところで CPU と GPU の性能が逆転している．この原因として，SPRAT で扱う対象が GPU に適したストリーム処理に限定されていること，CPU と GPU 間のデータ転送が比較的少ないこと，および CPU 用に生成される

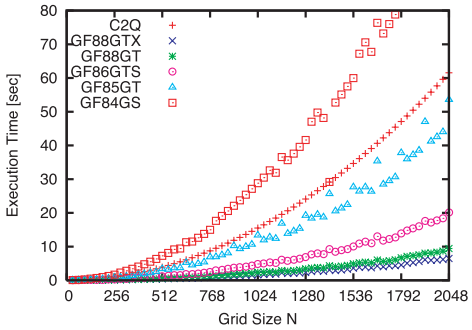

図 8　数値流体シミュレーションの実行時間
Fig. 8　Execution time of CFD simulation.

コードが十分に最適化されていないことなどが考えられる[⋆1]．データサイズによって CPU と GPU の性能が逆転する現象は，多くの文献で報告されている[4),19)]．そのような現象が起こるアプリケーションを SPRAT 言語で記述すれば，プロセッサの切替えによる高速化がさらに顕著に現れるものと考えられる．

4.4　オーバヘッドに関する考察

本節では，SPRAT による抽象化にともなうオーバヘッドについて考察する．まず，SPRAT では CPU と GPU を状況に応じて使い分けてカーネル関数を実行するため，カーネル関数で使われるストリームを `stream` 型データとして扱う．この `stream` 型データは内部的にストリームデータを CPU 側のメインメモリ上と GPU 側のビデオメモリ上に複製して持っている．すなわち `stream` 型データは他の管理情報も含めて配列の 2 倍以上のメモリ容量を消費しており，このことが実行を担当するプロセッサをプログラマに意識させないように抽象化するためのオーバヘッドの 1 つとしてあげられる．

また，カーネル関数の外では `stream` 型データの各要素を参照することはできない．このため，ストリームデータを初期化するためには，まず通常の配列にストリームデータを格納し `streamRead` を使って `stream` 型に複製する必要がある．このメモリコピーに要する時間も抽象化のためのオーバヘッドと考えることができる．一般的に，このメモリコピーのオーバヘッドが無視できないようなアプリケーションに関しては，SPRAT の拡張文法を使わない方が高速である．

⋆1 ストリーム処理を CPU 向けに最適化する方法に関しては文献 18) で議論されている．

SPRAT コンパイラによって自動生成されるコードはストリーム処理を素直に記述したものであり，BrookGPU で見られるような抽象化にともなう顕著な性能低下は生じていない．本実験の LU 分解や流体シミュレーションと同等のストリーム処理を C/C++で直接記述した場合，問題サイズ 2,048 × 2,048 の実行時間はそれぞれ約 5.71 秒および約 57.1 秒であるのに対して，SPRAT で自動生成した CPU 向けコードではそれぞれ約 5.91 秒および約 61.5 秒であった．両者の実行時間の違いは，SPRAT 言語の文法上な制約による冗長な記述，プロセッサ切替えの判定処理に要するオーバヘッド，SPRAT 実行時環境自体のオーバヘッドなどによるものと考えられる．その差は実行時間の 3.5%および 7.7%であり，プロセッサ切替えによって得られる速度向上を考慮すると，オーバヘッドは十分に小さいといえる．

同様に，SPRAT コンパイラによって自動生成される GPU 向けのコードでも，冗長な処理は少ないため，本論文で実装したアプリケーションに関して極端な性能低下は見られなかった．SPRAT を介さずに CUDA で図 4 を同様に記述した場合[⋆1]，行列サイズ 2,048 × 2,048 の LU 分解に GF88GTX で約 3.55 秒を要する．図 5 における GF88GTX の結果は約 3.56 秒であり，両者の実行時間はほぼ等しい．この結果から，GPU 向けのコードにおいても抽象化にともなうオーバヘッドは小さいといえる．

また，絶対位置指定されている `gather` ストリーム要素は複数のスレッドから参照されるため，SPRAT ではそのようなデータがあらかじめ共有メモリ[2] に複製されている（付録の GPU 向けコード参照）．図 4 の処理を共有メモリを使わないように CUDA で直接記述した場合，行列サイズ 2,048 × 2,048 の LU 分解に GF88GTX で約 8 秒を要する．したがって，GPU による高速処理のために重要な共有メモリを CUDA に関する知識なしに一部利用できており，限定的ではあるが SPRAT コンパイラは最適化作業を補助できている．この観点からも SPRAT コンパイラの有用性は明らかである．

なお，文献 16) と比較して図 5 の GPU の性能は著しく低い．これは文献 16) では OpenGL による実装を用いているのに対して，SPRAT では CUDA のコードに変換しているためであり，図 4 のコードでは CUDA のメモリアラインメントに関する制約を満たせないことから図 3 中の non-coalesched のようにメモリアクセス性能が極端に低下しているためである．このようにあるプロセッサが不得手としている処理の場合，その実効性能が低いことを検知し，処理の担当を他のプロセッサに変更できることが SPRAT の最大の特徴である．また，そのように性能が出ないように書かれたコードを性能が出るようなコードに変換することも，今後の重要な課題の 1 つである．

⋆1 SPRAT が生成するコードと同様に共有メモリを利用し，スレッド数などの条件も同一とした．

5. ま と め

本論文では，CPU と GPU を搭載する PC の性能を複合型計算システムとして最大限に活用することを目的とし，各処理に対して適切な担当プロセッサを自動選択する機能を有するストリーム処理記述用言語 SPRAT を提案し，その有効性を実証実験により評価した．SPRAT では，同一のプログラム記述から CPU および GPU で実行するためのコードを自動生成し，実行時に各ストリーム処理を担当するプロセッサを選択可能とする．実験の結果より，システムに搭載されている CPU と GPU の組合せや，処理されるデータのサイズに依存して CPU と GPU の演算性能が逆転するという特性を，本提案手法では自動的に利用できることが示された．

今後の課題としては，実行時間だけではなく他の評価関数，特に消費電力などを考慮してプロセッサを適切に切り替えることがあげられる．ストリーム処理の内容によって GPU の実効性能は大きく異なるが，消費電力はその実効性能には必ずしも比例しない．このことから，高い実効性能が期待できる処理のみを GPU で選択的に実行することにより，処理終了までに消費されるエネルギーを削減できると考えている．また，実行時性能予測の精度改善のため，オンライン自動チューニング技術[20] の利用などが考えられる．さらに，本論文中の LU 分解のように，GPU の性能が低くなる SPRAT コードを最適化する手法の開発も重要な課題である．

謝辞　多くの貴重なコメントをいただいた査読者の方々に深く感謝いたします．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) NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Architecture programming guide version 1.1 (2007).
3) Peercy, M., et al.: A Performance-Oriented Data Parallel Virtual Machine for GPUs, *ACM SIGGRAPH 2006 Sketches* (2006).
4) McCool, M.D., et al.: Performance Evaluation of GPUs Using the RapidMind Development Platform, *Poster reception at the ACM/IEEE SC06* (2006).

5) Papakipos, M.: SC06 GPGPU Course: PeakStream Platform, *the ACM/IEEE SC06 tutorial* (2006).
6) 片桐孝洋：ソフトウェア自動チューニング，慧文社 (2004).
7) Buck, I., et al.: Brook for GPUs: Stream Computing on Graphics Hardware, *ACM Trans. Graph.*, Vol.23, No.3, pp.777–786 (2004).
8) Houston, M.: SC06 GPGPU Course: High-Level Languages, *The ACM/IEEE SC06 tutorial* (2006).
9) 大島聡史ほか：CPU と GPU を用いた並列 GEMM 演算の提案と実装，情報処理学会論文誌：コンピューティングシステム，Vol.47, No.SIG 12(ACS 15), pp.317–328 (2005).
10) Jiang, C. and Snir, M.: Automatic Tuning Matrix Multiplication Performance on Graphics Hardware, *14th International Conference on Parallel Architectures and Compilation Techniques* (*PACT'05*), pp.185–194 (2005).
11) Buck, I., et al.: GPUBench: Evaluating GPU Performance for Numerical and Scientic Applications, *2004 ACM Workshop on General Purpose Computing on Graphics Processors*, pp.C–20 (2004).
12) Trancoso, P. and Charalambous, M.: Exploring Graphics Processor Performance for General Purpose Applications, *The Euromicro Symposium on Digital System Design, Architectures, Methods and Tools* (2005).
13) 伊藤信悟，伊野文彦，萩原兼一：GPGPU アプリケーションの開発を支援するための性能モデル，先進的計算基盤システムシンポジウム（SACSIS2007），pp.27–34 (2007).
14) He, B., et al.: Efficient Gather and Scatter Operations on Graphics Processors, *The ACM/IEEE SC07* (2007).
15) Buck, I. and Purcell, T.: A Toolkit for Compunation on GPUs, *GPU Gems*, Addison-Wesley (2004).
16) Galoppo, N., et al.: LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, *The ACM/IEEE SC05* (2005).
17) 梶島岳夫：乱流の数値シミュレーション，養賢堂 (1999).
18) Gummaraju, J. and Rosenblum, M.: Stream Programming on General-Purpose Processors, *MICRO 38: Proc. 38th annual IEEE/ACM International Symposium on Microarchitecture*, pp.343–354 (2005).
19) Christen, M., et al.: General-Purpose Sparse Matrix Building Blocks using the NVIDIA CUDA Technology Platform, *Workshop on General Purpose Processing on Graphics Processing Units*, pp.1–8 (2007).
20) 須田礼仁：オンライン自動チューニングのための Bayes 統計に基づく逐次実験計画法，ハイパフォーマンスコンピューティングと計算科学シンポジウム，pp.73–80 (2008).

付　　録

SPRAT コンパイラによって図 4 の各カーネル関数を変換した結果の一部を以下に示す．なお，SPRAT コンパイラでは名前の競合を避けるために複雑な変数名を出力するが，読みやすさのために以下ではできるだけ短い変数名に置換してある．

```
// --- CPU kernels ---
void _krnl_normalize_column__spratc_cpu(void *func_params)
{
  __FuncCPUArg *func_cpu_arg = (__FuncCPUArg *)func_params;
  void         *arg          = func_cpu_arg->arg;
  int k_index[2] = {0,0};
  int k_beg[2] = { func_cpu_arg->start_x, func_cpu_arg->start_y};
  int k_end[2] = { func_cpu_arg->end_x, func_cpu_arg->end_y};
  int k_size[2] = { func_cpu_arg->end_x-func_cpu_arg->start_x,
                    func_cpu_arg->end_y-func_cpu_arg->start_y };
  float *dst=*(float**)((char*)arg);
  int pitch_dst=*(int*)((char*)arg+izeof(float*));
  float *src=*(float**)((char*)arg+sizeof(float*)+sizeof(int));
  int pitch_src=*(int*)((char*)arg+sizeof(float*)+sizeof(int)
                        +sizeof(float*));

  for (k_index[1]=k_beg[1];k_index[1]<k_end[1];k_index[1]++) {
    for (k_index[0]=k_beg[0];k_index[0]<k_end[0];k_index[0]++) {
       dst[k_index[0]*1+k_index[1]*pitch_dst+0*0]
          /=src[pitch_src*(-1)+1*(k_index[0]+(0))];
    }
  }
  return;
}

void _krnl_row_operation__spratc_cpu(void *func_params)
{
  __FuncCPUArg *func_cpu_arg = (__FuncCPUArg *)func_params;
  void         *arg          = func_cpu_arg->arg;
  int k_index[2]={0,0};
  int k_beg[2]={func_cpu_arg->start_x,func_cpu_arg->start_y };
  int k_end[2]={func_cpu_arg->end_x,func_cpu_arg->end_y};
  int k_size[2]={func_cpu_arg->end_x-func_cpu_arg->start_x,
                 func_cpu_arg->end_y-func_cpu_arg->start_y};
  float *dst= *(float**)((char*)arg);
  int pitch_dst=*(int*)((char*)arg+sizeof(float*));
  float *src=*(float**)((char*)arg+sizeof(float*)+sizeof(int));
  int pitch_src=*(int*)((char*)arg+sizeof(float*)+sizeof(int)
                        +sizeof(float*));

  for (k_index[1]=k_beg[1];k_index[1]<k_end[1];k_index[1]++) {
    for (k_index[0]=k_beg[0];k_index[0]<k_end[0];k_index[0]++) {
       dst[k_index[0]*1+k_index[1]*pitch_dst+0*0]
          -=src[pitch_src*(-1)+1*(k_index[0]+(0))]
```

```
          *src[pitch_src*(k_index[1]+(0))+1*(-1)];
    }
  }
  return;
}

// --- GPU kernels ---
__global__ void _krnl_normalize_column__spratc_cuda(
    float *dst, int pitch_dst,
    float *src, int pitch_src,
    int odd_x, int odd_y )
{
  int k_index[2]
  = { blockDim.x*blockIdx.x+threadIdx.x ,
      blockDim.y*blockIdx.y+threadIdx.y };
  int k_size [2]
  = { blockDim.x*gridDim.x-odd_x,
      blockDim.y*gridDim.y-odd_y };
  int flag = (k_index[0]<k_size[0])
           &&(k_index[1]<k_size[1]);
  int index_dst=k_index[1]*pitch_dst+k_index[0];
  int index_src=k_index[1]*pitch_src+k_index[0];
  __shared__ float sbuf[BLOCK_SIZE_X];

  if((threadIdx.y==0)&&(k_index[0]<k_size[0])){
    sbuf[threadIdx.x+0]=src[(-1)*pitch_src+k_index[0]*1];
  }
  __syncthreads();
  if(!flag) return;
  dst[index_dst]/=sbuf[threadIdx.x];
  return;
}

__global__ void _krnl_row_operation__spratc_cuda(
    float *dst, int pitch_dst,
    float *src, int pitch_src,
    int odd_x, int odd_y)
{
  int k_index[2]
  = { blockDim.x*blockIdx.x+threadIdx.x ,
      blockDim.y*blockIdx.y+threadIdx.y};
  int k_size [2]
  = { blockDim.x*gridDim.x-odd_x ,
      blockDim.y*gridDim.y-odd_y};
  int flag = (k_index[0]<k_size[0])
           &&(k_index[1]<k_size[1]);
  int index_dst=k_index[1]*pitch_dst+k_index[0];
  int index_src=k_index[1]*pitch_src+k_index[0];

  __shared__ float sbuf1[BLOCK_SIZE_X];
  __shared__ float sbuf2[BLOCK_SIZE_Y];

  if((threadIdx.y==0)&&(k_index[0]<k_size[0])){
    sbuf1[threadIdx.x+0]
      =src[(-1)*pitch_src+k_index[0]*1];
  }
  else if((threadIdx.y==1)
     &&(blockDim.y*blockIdx.y+threadIdx.x<k_size[1])){
    sbuf2[threadIdx.x+0]
     =src[(blockDim.y*blockIdx.y+threadIdx.x)
           *pitch_src+(-1)*1];
  }
  __syncthreads();
  if(!flag) return;
  dst[index_dst]-=sbuf1[threadIdx.x]*sbuf2[threadIdx.y];
  return;
}
```

(平成 20 年 1 月 29 日受付)
(平成 20 年 5 月 5 日採録)

滝沢　寛之（正会員）

平成 11 年東北大学大学院情報科学研究科博士後期課程修了．博士（情報科学）．同年新潟大学助手，平成 15 年東北大学助手，平成 16 年同大学講師．現在，高性能計算システム，コンピュータアーキテクチャとそれらの応用に関する研究に従事．平成 18 年船井情報科学奨励賞，平成 20 年野口研究奨励賞受賞．電子情報通信学会，IEEE 各会員．

白取　寛貴

平成 18 年東北大学工学部機械知能工学科卒業．平成 20 年同大学大学院情報科学研究科博士前期課程修了．現在，新日鉄ソリューションズ株式会社勤務．在学中は，GPU による高性能計算の性能予測に関する研究に従事．

佐藤　功人（学生会員）

平成 19 年東北大学工学部機械知能工学科卒業．現在，同大学大学院情報科学研究科博士前期課程に在学し，異種プロセッサを協調して効率的に利用するための言語に関する研究に従事．

小林　広明（正会員）

昭和 63 年東北大学大学院博士課程修了．同年東北大学助手．平成 3 年東北大学講師．平成 5 年東北大学助教授．平成 13 年東北大学教授（平成 20 年度よりサイバーサイエンスセンターセンター長兼任）．平成 18 年より NII 客員教授併任．コンピュータアーキテクチャ，並列処理システムとその応用に関する研究に従事．工学博士．IEEE Senior Member，ACM，電子情報通信学会各会員．