OMPCUDA：GPU向けOpenMPの実装

大島聡史[†]　平澤将一[†]　本多弘樹[†]

GPU(Graphics Processing Unit) を用いた汎用演算 GPGPU(General-Purpose computation using GPUs) は高い演算性能が注目されている一方で，プログラム作成の難しさが問題となっている．そこで我々は，既存の並列プログラミング手法を用いた GPGPU プログラミングを提案している．本論文では共有メモリ型並列計算機で広く用いられている OpenMP を用いた GPGPU の可能性を探るため，CUDA 対応 GPU 向けの OpenMP 処理系 OMPCUDA を実装した．また，並列性の高いテストプログラムを用いて評価を行い，並列プログラムが容易に作成できることおよび既存の OpenMP と同様の記述で容易に並列高速化できることを確認した．

OMPCUDA : Implementation of OpenMP for GPU

Satoshi OHSHIMA,[†] Shoichi HIRASAWA[†] and Hiroki HONDA[†]

General-purpose computation using GPU (GPGPU) has been a focus of attention because of its performance, but the difficulty of GPGPU programming is a problem. So we have proposed GPGPU programming style using existing parallel programming style. In this paper, we implemented OMPCUDA OpenMP for CUDA-capable GPU, for explore a possibilities of GPGPU using OpenMP. Then, we evaluated out implimentation using test program. As a result, we confirmed that OMPCUDA make GPGPU parallel programming easy and can get speed-up.

1. はじめに

GPU(Graphics Processing Unit) は CPU(Central Processing Unit) と比べて理論演算性能や電力あたり性能が高いため，数値計算や動画像処理などを高速化するアクセラレータとして注目されている．今日では多くの PC に GPU が搭載されており，GPU を汎用演算に用いる GPGPU(General-Purpose computation using GPUs)[1] を利用可能な環境が普及している．

一方で，GPGPU プログラミングには専用の言語やライブラリを利用する必要がある．これらを利用するためには GPU のハードウェア構成や実行モデル，性能最適化手法などを新たに習得する必要があり，アプリケーションプログラマが GPGPU プログラミングを行う上での手間となっている．

我々はより多くのアプリケーションプログラマが GPGPU を簡便に利用できるようにすることを目指している．GPU は並列処理に適したハードウェアであるため，GPGPU による高速化が特に期待されているのは高い並列性を持つアプリケーションである．一方で CPU 向けの並列化プログラミング手法については既に多くの研究が行われている．そこで，GPGPU プログラミングに既存の並列化プログラミング手法を利用可能とすることによるアプリケーションプログラマにとっての手間の削減を提案している[2]．

本論文では，既存の共有メモリ型並列計算機で広く用いられている OpenMP を用いた GPGPU の可能性を検討する．また，GPGPU 環境の 1 つである CUDA 向けの OpenMP 処理系”OMPCUDA” を実装し，アプリケーションプログラマが GPU の性能を簡便に利用できるかを確認する．

本論文の構成を以下に示す．2 章では現在 GPGPU に用いられているプログラミング手法や言語についてまとめ，GPGPU プログラミングの課題を確認する．3 章では OpenMP を用いた GPGPU プログラミングの可能性を検討する．4 章では OMPCUDA の実装について述べ，5 章では OMPCUDA についてサンプルプログラムを用いて記述の容易性と性能の確認を行う．6 章はまとめの章とする．

† 電気通信大学 大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications
独立行政法人科学技術振興機構，CREST
Japan Science and Technology Agency, CREST

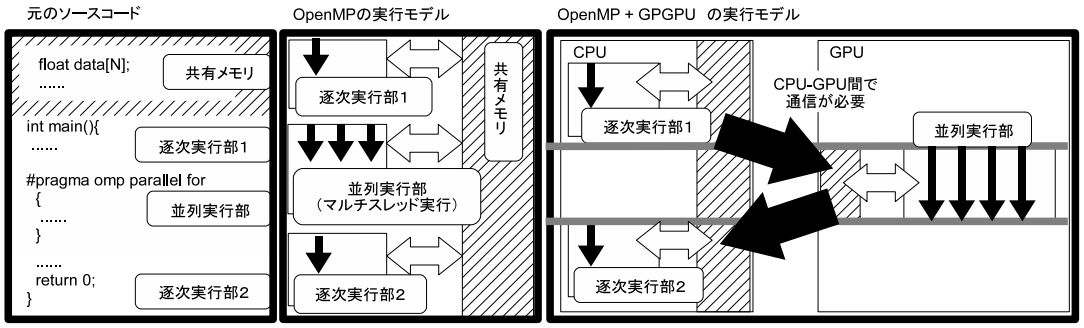

図 1 OpenMP を用いた GPGPU プログラミングの基本的な考え方
Fig. 1 Basic idea of GPGPU programming using OpenMP

2. GPGPU プログラミングの課題

GPU 本来の目的である画像処理を行うプログラムの作成には，グラフィックス API とシェーダ言語が用いられている．初期の GPGPU ではこれらの言語やライブラリを利用し，演算データの配置をテクスチャに対するデータの設定に割り当て，演算内容をシェーダ言語の記述およびシェーダを用いた画像描画の手続きに置き換え，演算結果の取得にテクスチャ間描画を利用することで GPU による汎用演算を実現した．

しかしこのような画像処理プログラミングの手法を用いて汎用演算を記述するには，対象アプリケーションのアルゴリズムに加えて画像処理プログラミングや GPU アーキテクチャについての知識が必要であり，アプリケーションプログラマにとって大きな負担となってきた．そこで近年では，画像処理プログラミングの知識を必要とせずに GPGPU プログラミングを行える新しい言語やライブラリの研究が進められている．

CUDA[3] や CTM[4] は，GPU ベンダが提供する GPGPU 向けのプログラミング言語やライブラリであり，OpenGL や DirectX よりも低いレベルで GPU にアクセスすることができる．そのため，GPU のアーキテクチャに適した実装を行うことでより高い性能を得られる可能性がある．しかしながら，これらの言語やライブラリは GPU アーキテクチャへの依存度が高いため，利用するには GPU のアーキテクチャを理解する必要がある．そのため，アプリケーションプログラマへの負担が軽減されているとは言い難い．

RapidMind[5] や SPRAT[6] など，GPU による処理を「ストリームと呼ばれる入力データ」に対して「カーネルと呼ばれる演算操作」を作用するものとみなすストリーミング言語の研究も進められている．ストリーミング言語は GPU のアーキテクチャをストリーム処理という概念で抽象化している．そのため GPU に関する知識がないアプリケーションプログラマでもストリーミング言語を使うことで GPU 向けの並列プログラムを作成することが可能となる．しかしながら，ストリーミング処理という新たな概念や新しい言語仕様の習得が新たな手間となっている．

3. OpenMP を用いた GPGPU プログラミング

GPU は並列処理方式によるハードウェアであり，GPGPU において高速化が期待されているのは並列性が高いアプリケーションである．

CPU 向けの並列化については，実行環境や対象アプリケーションにあわせて様々な言語やライブラリが利用されている．そこで，GPGPU においてもこれらの言語やライブラリを利用することができれば，アプリケーションプログラマにとって言語仕様や並列実行モデルを新たに習得する手間を軽減することが可能となり，GPU の持つ高い計算性能をより多くのアプリケーションプログラマが簡便に利用できるようになると考えられる．

そこで我々は，共有メモリ型並列計算機において広く利用されている OpenMP を利用した GPGPU プログラミングの可能性を明らかにしようと考えた．以下本章では OpenMP と GPGPU の対応付けについて検討し，さらに次章では我々が実装を行っている OpenMP 処理系 OMPCUDA の実装について述べる．

CPU 向けの並列化については，SIMD，SMP，マルチコア CPU，PC クラスタ，Grid など様々な環境に向けて数多くの研究が行われている．特にアプリケーションプログラマが明示的に並列性を記述する際には，pthread などのスレッドライブラリ，MPI などの通信ライブラリ，そして OpenMP が広く利用されている．

本研究では，以下の 2 点から OpenMP に着目し，OpenMP を用いることで GPGPU プログラミング容

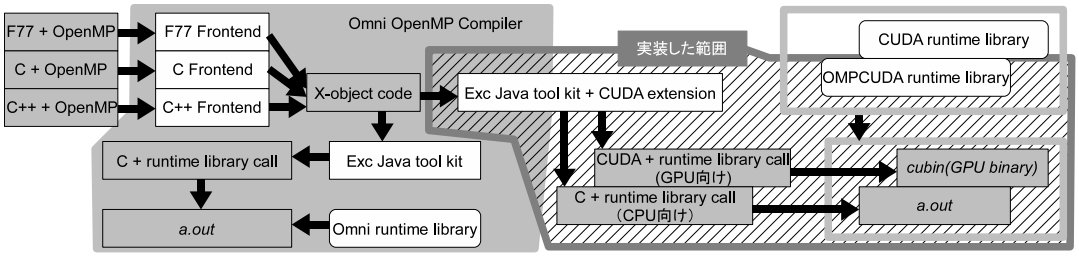

図 2 Omni と OMPCUDA の関係および OMPCUDA の全体像
Fig. 2 Relationship between Omni and OMPCUDA, and whole picture of OMPCUDA

易に行えるのではないかと考えた．これらの考えに基づく OpenMP を用いた GPGPU プログラミングの基本的な考え方を図 1 に示す．

(1) OpenMP は実行環境として，同時に実行可能な複数のスレッドと，スレッド間で共有されたメモリを持つ環境を想定している．
一方現在の GPU は，同時に実行可能な多数の演算器と，GPU 全体で共有されたメモリを搭載している．
そのため GPU 上の多数の演算器を OpenMP におけるスレッドに，GPU 上のメモリを OpenMP におけるスレッド間で共有されたメモリに割り当てることで，OpenMP を GPU に対応づけられると考えられる．

(2) GPU は並列処理に特化したハードウェアであるため，GPGPU においては並列性の高い部分のみを GPU 上で並列実行し，その他の部分は CPU で逐次実行するのが一般的である．
一方 OpenMP を用いた並列化では，プログラマが指示子を用いて明示的に指定した部分のみを複数のスレッドを共有メモリを用いて並列実行し，その他の部分は逐次実行する．そのため逐次実行部と並列実行部の境界が明確である．プログラム全体の中から並列化による高速化を行いやすい部分のみを選択的に並列実行するという実行モデルは GPGPU と OpenMP で共通している．

既存の OpenMP では逐次実行部と並列実行部とで同じメモリ空間を利用する．これに対して CPU と GPU は互いに独立したメモリを持ちメモリ空間も異なるため，CPU-GPU 間で必要なデータを送受信する必要がある．OpenMP の実行モデルでは逐次実行部と並列実行部が同時に実行されることはないため，逐次処理部と並列処理部の境界で必要なデータを全て送受信すれば正しい実行結果が得られると考えられる．

より高速な実行のためには，並列実行部を CPU と GPU で分割実行することが有効である可能性がある．しかし，CPU-GPU 間での適切なデータの送受信が必要になるため，実装が難しくなることや逆に性能が低下してしまうことが予想される．さらに，並列実行部をより高速に実行するには，CPU のみで実行するか，GPU のみで並列実行するか，(可能であれば)CPU と GPU で分割並列実行するかを選択する必要が生じる．この選択は並列実行部の計算量や CPU-GPU 間の通信時間に左右されるため，アプリケーションプログラマが選択するための新たな OpenMP 指示子やの導入や，最も高速な方法の選択を支援するチューニング機構などが必要となる可能性も考えられる．

4. OMPCUDA の提案と実装

OpenMP を用いて記述されたプログラムの並列実行部を GPU 上で実行するには，アプリケーションプログラマが記述したプログラムを GPU 上で実行可能な形式，例えば CUDA やシェーダ言語等，既存の GPGPU プログラミングで用いられている形式に変換する必要がある．

本研究ではプログラムを変換する機構が作りやすいことを重視して実行環境を CUDA 対応 GPU に絞り，OpenMP を用いて記述されたプログラムの並列実行部を CUDA 対応 GPU 上で実行するための処理系 OMPCUDA の実装を行った．

以下本章では OMPCUDA の具体的な実装について述べる．OMPCUDA は，OpenMP を用いた典型的な並列プログラムである**#pragma omp parallel for** 節を用いたループ並列プログラムに対して，GPU の特徴である高い並列性を有効に活用し効果的に並列化・高速実行することを主眼に置いて実装を行った．

4.1 全体の構成

OMPCUDA は，既存の OpenMP 処理系 Omni OpenMP Compiler version 1.6[7] (以下 Omni) をベースに実装した．Omni と OMPCUDA の関係および OMPCUDA の全体像を図 2 に示す．

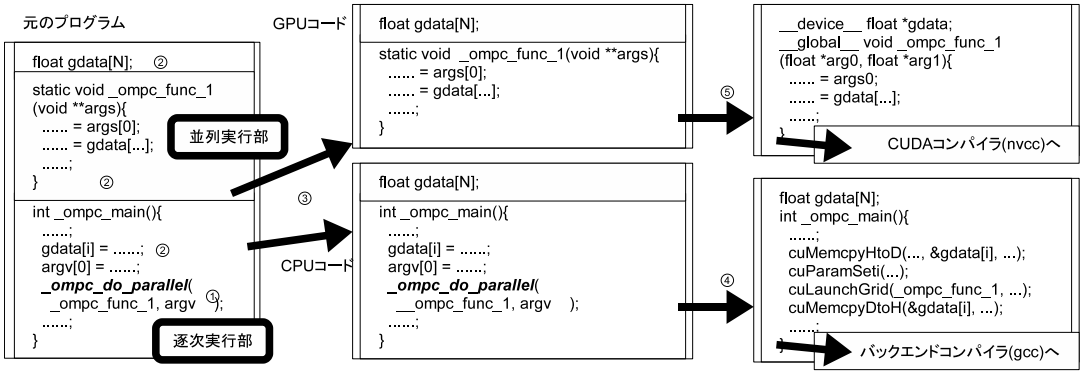

図 3 プログラム変換機構の処理手順
Fig. 3 Sequence of program converter

OpenMP における並列実行部の指定はブロック単位で行われるのに対して，Omni の中間コードにおける並列実行部は関数として分離させられている．分離させられた関数は，実行時ライブラリの並列実行開始関数によってスレッドに割り当てられ，並列実行される．

一方で，CUDA における GPU 上での並列処理も関数単位であり，CPU からの指示に従い GPU 上の多数の演算器で同時に同じプログラムを実行する．ただし，演算の前に CPU から GPU へ必要なデータを送信し，また演算の後には GPU から CPU へ結果を送信する必要がある．

そこで OMPCUDA では，処理の対象を OpenMP プラグマの挿入されたプログラム (ソースコード) ではなく，Omni によってプラグマの解釈と実行時ライブラリの呼び出しに変換された中間表現とした．

プログラム変換機構は中間コードを調査し，“Omni の並列実行関数呼び出し” を “CPU から GPU へのデータ送信，GPU への演算実行開始指示，GPU から CPU へのデータ書き戻し” の一連の処理に置き換えて CUDA 向けのプログラムに変換する機構である．CPU-GPU 間で送受信する必要があるデータを探し出す機能も持ち合わせている．

実行時ライブラリは，Omni の実行時ライブラリが持つ機能を CUDA 向けに再実装したものである．

4.2 プログラム変換機構

OMPCUDA のプログラム変換機構が行う，Omni の中間表現を CUDA に対応したプログラムへと変換する手順を以下に示す．この手順は図 3 と対応している．図には C 言語風のソースコードを示しているが，実際には必要に応じて C 言語と Omni の中間表現とを相互変換して処理を行っている．

(1) 中間表現で書かれたプログラムを調査し，実行時ライブラリの並列実行開始関数が呼び出されている部分を探し出す (図 3-①).

(2) 並列実行部の内部で利用されている変数を確認し，CPU-GPU 間で送受信を行う必要がある変数を特定する (図 3-②).

(3) 並列実行部を別ファイルに書き出す (図 3-③). 並列実行部に関数呼び出しが存在する場合は，その関数内で利用している変数についても同様に確認し，まとめて書き出す．ただし，並列実行部で呼ばれている関数は逐次実行部から呼び出される可能性もあるため，元のソースコードにも残したままとする．
以下，別ファイルに書き出したソースコードを GPU コード，残されたソースコードを CPU コードと呼ぶことにする．

(4) CPU コードについては，Omni の並列実行開始関数呼び出しを CPU-GPU 間のデータ送受信および GPU への演算実行開始指示に書き換える．これをバックエンドコンパイラ (gcc など) でコンパイルし，CUDA の実行時ライブラリとリンクして CPU 用の実行ファイルを生成する (図 3-④).

(5) GPU コードには，CUDA の記法に従い関数や変数に指示子を付加する．さらに関数の引数を調整し，CPU-GPU 間で変数の送受信が行えるようにする．最後に CUDA コンパイラ (nvcc) でコンパイルを行い GPU 用の実行ファイルを生成する (図 3-⑤).

4.3 実行時ライブラリ

Omni の実行時ライブラリは，スレッドの生成および割り当てやループのスケジューリング，スレッド間の同期処理などの機能を提供している．

OMPCUDA の実行時ライブラリは，Omni において逐次実行部で行っている処理は CPU で行い，並列

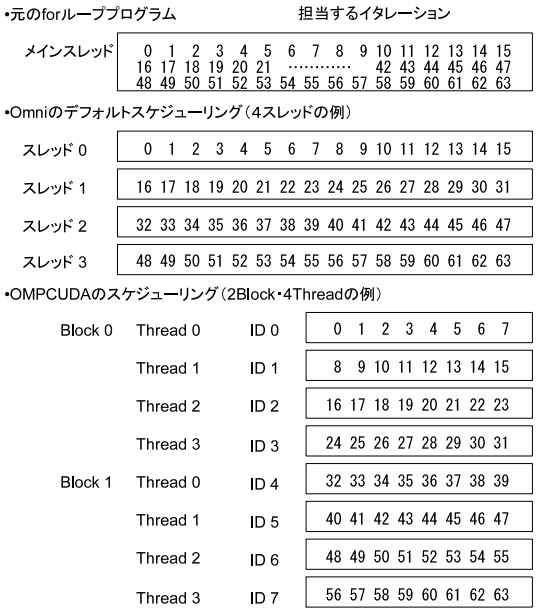

図 4 for ループのスケジューリング例
Fig. 4 Exapmle of for-loop scheduling

実行部で行っている処理は GPU で行う，という方針で実装している．CPU から GPU または GPU から CPU へ移動した方が性能が向上する処理もあると考えられるが，今後の課題とし本論文では扱わない．

OMPCUDA では単純な for ループの並列化に最低限必要な関数として，GPU 上で実行される for ループのスケジューリングを行う関数を実装した．Omni では CPU 上で実行されるスレッド割り当ておよび並列実行開始関数も提供しているが，これについてはプログラム変換機構による CUDA 実行時ライブラリ関数の呼び出しへの書き換えが役割を果たしている．

for ループのスケジューリング

Omni における for ループの並列化は，実行時ライブラリが提供する，各スレッドが自らのスレッド ID を元に新たな部分ループを作成するための関数を用いて行われている．CUDA の実行時ライブラリにもスレッド ID に対応する一意の ID を得る機構が備わっているため，OMPCUDA はこれを用いて Omni と同様に部分ループを作成するための関数を提供することとした．現在の OMPCUDA ではループのイタレーションを等分割するスタティックスケジューリングのみを提供している．実装したスケジューリング機構による for ループのスケジューリング例を図 4 に示す．

CUDA では GPU 上に多数の演算器が搭載されており，各演算器は 8 個単位で演算器群を構成している．CUDA では演算器ごとの実行単位を Thread，演算器群ごとの実行単位を Block と呼んでおり，CPU から GPU へ並列実行指示を行う際には Block 数と Thread 数を指定する．同一 Block 内の Thread は同一演算器群内の演算器に割り当てられて実行される．各 Block を構成する Thread 数は Block ごとに異なる値を指定することはできず，また一回の並列実行中に使用する Thread 数や Block 数を変更することはできない．同一演算器群内の各演算器は同時に異なる種類の演算を行うことができないものの，Block ごと・Thread ごとに個別の ID が振られるため，実行時にはこの ID を用いることで異なるデータに対して同じ演算を行うことができる．指定する Block 数および Thread 数が演算器群数および演算器数を超える場合には時分割実行され，最も良い性能が得られる演算器群あたりの演算器数は 256 程度であるとされている．

CUDA における並列処理については，1Block 内の複数 Thread を用いた並列処理，複数 Block を用いるが各 Block 内では 1Thread のみを利用する並列処理，複数 Block・複数 Thread を用いた並列処理，以上の 3 種類の並列処理が考えられる．複数 Block・複数 Thread を用いた並列処理が最も多くの演算を同時実行できるものの，同一 Block 内の各 Thread が同時に異なる種類の演算を行えないことから，対象問題によっては複数 Thread を用いた並列処理の効果が得られない可能性がある．

これに対して，単純な for ループの並列化では各スレッドが同時に同じ演算を行うことが多い．そのため CPU 向けの OpenMP におけるスレッド並列処理は，上記 3 種類の並列処理のいずれにも対応づけることができると考えられる．ただし，対象問題によっては複数 Thread を用いた並列処理の効果が得られない可能性があるうえに，最適な Block 使用数・Thread 使用数は対象問題と対象 GPU により左右される．そのため現在の OMPCUDA ではデフォルトの Thread 数として 256 を，Block 数としてループ回数を 256 で割ったものを用いている．

5. 評　　価

本章では OMPCUDA について，並列プログラムが容易に作成できるか，並列実行の効果が容易に得られるかの二点について評価を行う．

評価環境は表 1 に示す通りである．

評価対象プログラムとしては，理想的な並列性を持つプログラムが実際に並列高速化できるかを確認するため，行列積を用いた．行列積は高い並列性を持つ問題であり，GPU を用いて高速実行できる問題として

表 1 評価環境
Table 1 Evaluation environment

CPU	Xeon E5345 2.33GHz (QuadCore)
メインメモリ容量	4.0GB (32bitOS につき 3.2GB 使用可)
GPU	GeForce GTX 280 (コア Clock 604MHz, SP クロック 1296MHz)
ビデオメモリ容量	1.0GB
GPU 接続バス	PCI-Express Gen2 x16
OS	CentOS 5.0 (kernel 2.6.18)
コンパイラ	GCC 3.4.6, nvcc 2.0 v0.2.1221 Omni OpenMP Compiler version 1.6

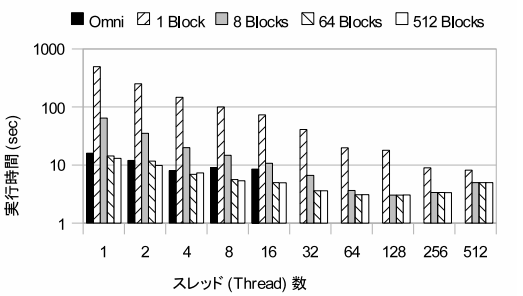

図 6 行列積の実行時間
Fig. 6 Execution time of matrix multiplication

```
#define N 1024
float a[N*N], b[N*N], c[N*N];
#pragma omp parallel for private(j,k)
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
      for(k=0; k<N; k++){
        c[i*N+j] += a[i*N+k] * b[k*N+j];
  } } }
```

図 5 行列積のソースコード
Fig. 5 Source code of matrix multiplication

も知られている．描画機能やシェーダを用いた実装，GPU の性能モデル化，CPU と GPU を用いた並列計算など様々なテーマで取り上げられている[8]．

本論文では最大性能を得ることよりも単純な実装で容易に性能を得られることを重視しているため，図 5 に示すような単純な三重ループによる実装を用いて性能を比較した．

実行結果を図 6 に示す．この図は Omni の使用しているスレッド数に対する実行時間と，GPU の Thread 数・Block 数に対する実行時間を示している．GPU のコアは CPU と比べるとクロック周波数が低く単純な構造のため，Thread 数・Block 数が少ないときは実行時間が長くなってしまっている．しかし使用する Thread 数・Block 数が増えると並列化の効果があらわれ，CPU を上回る性能を得られている．

この実験ではソースコードに修正を加えることなくプログラムを実行することができている．また実行時間についても，同じソースコードを用いながらも並列度を上げることで CPU より高い性能が得られている．これらのことから，OMPCUDA を用いることでアプリケーションプログラマは OpenMP の知識を用いて容易に高速な GPGPU プログラムを作成できることが確認できた．

6. おわりに

本論文では既存の並列化手法を用いて GPU を簡便に利用するための 1 つの実装として，CUDA 対応 GPU 向けの OpenMP 処理系 OMPCUDA を実装した．OMPCUDA は，アプリケーションプログラマにとって大きな手間となる GPU アーキテクチャに関する知識の習得や CPU-GPU 間の通信の記述等を隠蔽できること，並列化による性能向上を得られることを確認した．

現在は様々な OpenMP プログラムに対応すべく，for ループとともによく用いられるリダクション演算への対応などの実装を行っている．

参 考 文 献

1) Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. and Purcell, T. J.: A Survey of General-Purpose Computation on Graphics Hardware, *Eurographics 2005, State of the Art Reports*, pp. 21–51 (2005).
2) 大島聡史, 平澤将一, 本多弘樹: 既存の並列化手法を用いた GPGPU プログラミングの提案, 情報処理学会研究報告 (ARC-175), pp. 7–10 (2007).
3) NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Archtecture Programming Guide version 2.0 (2008).
4) Advanced Micro Devices: ATI CTM Guide Version 1.01 (2006).
5) RapidMind Inc.: RapidMind, `http://www.rapidmind.net/`.
6) 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ティングシステム, Vol. 1, No. 2, pp. 207–220 (2008).
7) M.Sato, S.Satoh, K.Kusano and Y.Tanaka: Design of OpenMP Compiler for an SMP Cluster, *EWOMP '99*, pp. 32–39 (1999).
8) 大島聡史, 吉瀬謙二, 片桐孝洋, 弓場敏嗣: 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. SIG12(ACS 15), pp. 317–328 (2006).