Vol.2010-ARC-192 No.7
Vol.2010-HPC-128 No.7
2010/12/16

ステンシル計算を対象とした 大規模GPUクラスタ向け自動並列化フレームワーク

野村達雄[†1] 丸山直也[†1]
遠藤敏夫[†1] 松岡聡[†1,†2]

近年,GPU を搭載した大規模クラスタが台頭し, 科学技術計算への応用が盛んに行われている. 科学計算の計算カーネルの一つとしてステンシル計算が頻繁に現れる. ステンシル計算はメモリ律速であり, またメモリのアクセスパターンが比較的単純であるため GPU によるアクセラレーションが有効である. しかし, ステンシル計算それ自身は簡潔に記述できるにも関わらず, 並列化のための問題の分割や袖領域の交換などの実装が煩雑になりがちである. そのため,GPU の利用は一部の知識を持った開発者に留まっている. 本研究ではハードウェアから独立した, ステンシル計算本来の簡潔な記述を保ったままのコードから GPU クラスタ向けに並列化されたコードを生成するフレームワークを提案する. ステンシル計算の問題例として三次元拡散方程式を手動で実装したものと, 提案するフレームワークによって自動生成されたもの性能を評価した. その結果フレームワークによって生成されたコードは手動で書かれたコードと比較して 75%〜125%程度の性能が得られた.

A Code Generation Framework for Stencil Computations on Large Scale GPU Clusters

Tatsuo Nomura,[†1] Naoya Maruyama,[†1]
Toshio Endo[†1] and Satoshi Matsuoka[†1,†2]

In the recent years, large scale clusters equipped with GPUs are considered one of the promising architectures in HPC, and as such studies of scientific applications are being conducted on such machines. The kernel of fluid dynamics typicaly belongs to the class of stencil computations. Problems in this class are usualy memory-intensive, and have a relatively simple pattern of memory access, so that it can benefit from using GPU as an accelarator. Although stencil computations themselves can be described concisely, huge amount of code must be manually written for parallelization such as domain decomposition and boundaries exchange. Those difficulties confine the utilization of GPUs to a handful of people who have expertise in it. Our work is to provide a framework which takes concise description of stencil computation as an input and generate parallelized code for GPU clusters. We have used 3d-diffusion-equation as an example problem for evaluation. We have evaluated the performance of two implementations; One has been implemented manually, and another has been generated by the framework. The code generated by the framework has achieved about 75%-125% of performance of the manual implementation.

1. 背　景

近年,HPC 分野で GPU を組み込んだ大規模なクラスタが次々と開発され, 科学技術計算への応用が盛んに行われている.2010 年 11 月, 第 36 回目 Top500[1)] では中国国防技術大学が開発した GPU クラスタ Tianhe-1A が 1 位,NSCS の GPU クラスタ Nebulae が 3 位, 東京工業大学の GPU クラスタが TSUBAME2.0 が 4 位をそれぞれ獲得している. 今後 GPU クラスタを効率的に利用することが益々重要になっている. しかし,GPU を使ったプログラミングは CPU のみを対象にする時より格段に複雑である. 開発者は CPU 向けのコード,GPU 向けのコード, そしてクラスタ上で動かす場合には並列化のための MPI コードを書かなければならない. 開発者は CPU をシリアルにのみ使った場合と比較して, 問題の分割方法, 計算資源の割り当て方法,CPU や GPU 固有の最適化などを更に考慮しなければならない.

科学計算の計算カーネルの一つとしてとしてステンシル計算が頻繁に現れる. ステンシル計算は一般的にメモリ律速であり,GPU を使った場合の性能向上が著しい.[2)] しかし,GPU クラスタを活用するためには, 開発者は CPU 向けに C,Fortran を,GPU 向けに CUDA を, 並列化のために MPI コードを書かなければならない. 特にステンシル計算では問題の領域を分割したときに隣接する領域間でデータ交換をする必要があるため,GPU から CPU,CPU から CPU, 更に CPU から GPU というようなデータ転送アルゴリズムを実装しなければならない. ステンシル計算自体は簡潔に表現できる場合が多いにも関わらず, 実際のプログラムは複雑になりがちである. そのため,GPU クラスタを活用するための敷居が高くなっている.

本研究では簡潔な表現で記述したステンシル計算をソースコードとし,GPU クラスタ向け

†1 東京工業大学
Tokyo Institute of Technology
†2 国立情報学研究所
National Institute of Informatics

Vol.2010-ARC-192 No.7
Vol.2010-HPC-128 No.7
2010/12/16

に並列化されたコードを生成するフレームワークを提案する. フレームワークは入力として C 言語で書かれたステンシル計算のソースコードを受け取り, 最適化された C,CUDA, そして MPI コードを生成する. 開発者が GPU に関する知識を持たなくても GPU クラスタ上で動作するステンシルアプリケーションを簡便に記述できることを目指す.

ステンシル計算の問題例として流体計算の一部である三次元拡散方程式を手動で実装したものと, 提案するフレームワークによって自動生成されたもの性能を評価した. 結果として, フレームワークによって生成されたコードは最適化やノード数によって手動で記述されたコードの 75〜125%程度の性能が得られた.

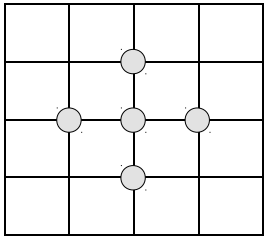
2. ステンシル計算

科学計算の中で, 特に格子法による流体計算の分野では, 一般的に計算のカーネルはステンシル計算になることが多い. 格子法では流体を離散的な粒子の集まりではなく[⋆1],Navier-Stokes に代表されるような偏微分方程式で連続体としてモデル化する. 実際に計算する際は計算対象の空間を格子状に離散化し, 各格子に流速や圧力などの初期値を与え, 離散化した時間経過による値の変化を計算する. 具体的な計算アルゴリズムとしは, 空間の全格子について隣接の格子からの影響を計算し, 値をアップデートすることを繰り返すことによって行われる. (図 1) は単純な二次元上の 5 点ステンシルの一例である. 例は $f^{n+1}_{x,y} = \sum c_i f^n_{x_i,y_i}$ と表せ, 変数はポテンシャル f のみであるが. ステンシルによっては v_x, v_y などの速度変数が加わる場合などがある. また, 速度が変化する場合は f の他に v_x, v_y もアップデートする必要がある.

2.1 ステンシル計算の実装

典型的なステンシル計算は計算部分は比較的少なく, 大量のデータ転送が中心になる場合が多い. また各格子点は, 隣接の格子のデータのみにアクセスするため, メモリへのアクセスパターンが比較的単純である. これらの性質は高スループットなメモリをもつ GPU の特徴によく符合している. そのため GPU を利用して適 b 切に実装すれば,CPU のみのものと比べて数倍から数十倍のパフォーマンス向上が達成できる場合がある.[2)]

現状では,GPU をクラスタを利用してステンシルアプリケーションを開発する場合, 開発者は CUDA や OpenCL などを使って GPU 向けのコードを開発する必要がある.GPU は CPU からドライブしなければならないので,CPU 向けのコードも必要である. 更にノード間で並列化するための MPI コードも必要である. ステンシル計算自体は簡潔にに記述できるにも関わらず, 問題の分割,CPU,GPU の通信, 境界条件の処理などの実装が複雑である. そのため,GPU クラスタを対象にしたステンシルアプリケーションでは計算以外の部分がコードの多くを占めることになる.



```
for (y=0; y<ysize; y++) {
    for (x=0; x<xsize; x++) {
        fnew(x,y) = cc*f(x,y)
           +cw*f(x-1,y)+ce*f(x+1,y)
           +cn*f(x,y-1)+cs*f(x,y+1);
    }
}
```

図 1 5 点ステンシルの例. 二次元空間の各点の値を隣接点を元にアップデートする.

単一 GPU に対しては,Kamil ら[4)] が GPU を含む Chip Multiprocessor(CMP) 向けにステンシル計算のカーネルのコードを自動生成する研究をしている.GPU クラスタをターゲットにした場合, 並列化するために問題空間を分割し各ノードに割り当てなければならない. 分割した空間の境界では袖領域[⋆2]の交換が必要になるため, ノード間でデータの転送しなければならない. ノード間のデータ転送は単純に実装するとボトルネックになりやすく, スケーラビリティを下げる要因になる. 通信の遅延を隠蔽するようなアルゴリズム[5)] が研究されているが, 現状では開発者がそれを手動で実装しなければならない.

本研究ではこれらの複雑さを排除し, 簡便な記述で GPU クラスタを対象にした高性能なステンシルアプリケーションが開発できるようにすることを目指す.

3. フレームワークの設計

ステンシル計算は本質的にはその計算対象のグリッドのデータ表現とそれに対する操作によって実現できる. しかし, 分散メモリ並列環境上で実現するためには格子の分割とそれらの間における適切な同期が必要になる. これは特に GPU クラスタではメモリ階層が複雑になるためにその実現が煩雑になる. 我々は対象ステンシルアプリケーションに本質的に必要

⋆1 希薄流体を扱う場合など, 粒子法[3)] のように流体を離散的な粒子として扱う場合もある.

⋆2 隣接した空間のデータ

Vol.2010-ARC-192 No.7
Vol.2010-HPC-128 No.7
2010/12/16

な部分のみの宣言的な記述を入力とし, それを GPU クラスタを含む各種実行環境にコンパイラにより変換するフレームワークを提案する.

3.1 フレームワークの構成

本フレームワークは主にコンパイラドライバ, トランスレータ, そしてランタイムライブラリから構成されている. 次節で述べられている C 言語を元にした記述を入力ソースコードとし, トランスレータがターゲットプロセッサの種類に応じて C コードや CUDA コードを生成する. 生成されたコードはコンパイラに渡され, 実際のマシンコードが生成される.

3.1.1 トランスレータ

トランスレータ部分ではソースレベルの変換を行う. 入力ソースコードを CPU 向けであれば C コード,GPU 向けであれば CUDA コードなどへ変換する. 変換に際してノード間の並列化に必要な MPI コードなどが挿入される. トランスレータ内部では入力ソースコードを解析する部分と, コード生成する部分にわかれている. 入力されたソースコードはまず構文解析にかけられ,AST を使った中間表現に変換される. 生成された AST からグリッドやステンシルの情報を解析する. 得られた情報を元に最適化を適用し, ターゲット用の並列化されたコードを出力する.

3.1.2 ランタイムライブラリ

ランタイムライブラリはメモリリソースやプロセッサリソースの管理, エラー処理などステンシル計算の種類によらない共通の処理を行う. また, 実行時のオートチューニングや, チェックポイントなどを行う.

3.1.3 コンパイラドライバ

コンパイルドライバは入力ソースコードをトランスレータに渡し, 出力されたコードをターゲットプロセッサのコンパイラに渡す. そしてリンカを呼び出し, 出力されたオブジェクトとランタイムライブラリを結合して実行ファイルを作る.

3.2 言 語 設 計

本フレームワークの設計には以下を基本方針とする.

- ステンシルアプリケーションの簡便かつ宣言的な記述
- 実行環境独立な記述
- 共有メモリプログラミングモデル
- 暗黙的並列プログラミングモデル

本フレームワークでは対象をステンシル計算に限定することでアプリケーションの記述の簡便性を高めることを目標とする. 具体的には, 偏微分方程式の求解などのステンシルアプリケーションでは, 本質的にはその離散化によって得られるグリッドとステンシル計算によって表現可能である. 本フレームワークではアプリケーションプログラマが記述したグリッドデータの定義とステンシル計算の関数定義を与えられ, 各種計算環境上で効率の良い実装コードを自動生成することを目標とする. アプリケーションドメインを限定しない, より適用範囲の広いフレームワークを設計することも可能であるが, 限定されたドメイン専用とすることでより簡便な記述と最適化を実現することを狙う. 我々は将来的にはステンシル計算によって構成される大規模実アプリケーションを本フレームワーク上に簡便に実装できることを目的とする.

フレームワークに従って記述されたアプリケーションの実際の実行環境は逐次 CPU や単一 GPU, 分散メモリ並列複数 CPU, 複数 GPU など様々な場合に対応 できるものとする. アプリケーショ ン自体にはそれら実行環境に依存した記述は許さず, フレームワークおよび実行ランタイムが個々の環境間の差異を吸収する. 特に実環境が分散メモリの場合においても本フレームワークによってアプリケーション側からは共有メモリとして操作可能なデータ表現を実現する.

本フレームワークに従って記述されたアプリケーションでは明示的な並列プログラムを記述するのではなく, ステンシル計算が内包する並列性を仮定したフレームワークによる自動並列化を実現する. これにより, Intel Threading Building Blocks におけるリストの自動並列処理等と同様に, アプリケーションプログラマは MPI や CUDA などの計算環境固有の並列プログラミング言語, ライブラリを用いることなくステンシル計算を自動並列処理による高速化を達成できる.

以上が本フレームワークの設計における基本方針であるが, これらの方針に従いつつ可能な限り効率の良い実装を可能にする設計を目標とする. 例えばグリッドに対する操作や操作対象であるグリッドがコンパイル時に静的に定まるようにフレームワークに制約を加えることで, コンパイル時の最適化や自動チューニングが比較的容易になる. 本フレームワークでは実際のステンシルアプリケーションを記述可能な範囲においてプログラマの自由度を極力減らし, コンパイラによる最適化の適用範囲の向上を狙う.

また, 簡便な記述を実現するために本フレームワーク専用の言語を設計することも可能である. そのようなドメイン固有言語により既存の汎用言語に比べてより簡潔な記述が可能になるが, 本研究では既存言語との親和性を重視し通常の C 言語上にフレームワークを設計する.

Vol.2010-ARC-192 No.7
Vol.2010-HPC-128 No.7
2010/12/16

3.3 プログラム構成

本フレームワークの入力ソースコードは通常の C 言語の文法に従い, 型の定義や大域変数の宣言, 関数定義から構成される. 型の定義には計算対象のグリッドの定義を, 関数定義にはステンシル計算を表す関数の定義を含む. ステンシル関数以外の関数内には, フレームワークが提供する関数やマクロを用いてグリッドの生成, 操作を記述でき, 本フレームワークではドライバコードと呼ぶ. 図 2 にサンプルコードを示す.

ドライバコード以外, すなわちステンシル計算部分以外は実行環境によらず逐次実行と同一の意味を持つものとする. これは例えば MPI 並列のような SPMD プログラムへフレームワークにより変換された場合においても成り立つものとする. 一般に SPMD プログラムではすべての並列プロセスが同一のプログラムを実行するが, 本フレームワークではドライバ部については単一代表プロセスのみが実行した場合と同一の結果を保証する. しかし, 実際に単一プロセスのみによる実行かどうかについては規程せず, 副作用がない限り実際には複数プロセスによる実行を許す.

本フレームワークにより入力ソースコードは上記意味を持つような実装コードに変換される. これは CPU 向けであれば通常の C コード, GPU 向けであれば CUDA コード等, ソースコードレベルへの変換であり, 最終的に通常のコンパイラにより実行コードへコンパイルされる.

3.4 データ表現

本フレームワークではグリッドを基本データ構造として提供し, その次元および要素の型によってグリッドの型が定義される. 本フレームワークを用いてプログラムを記述するユーザはプログラム中に以下のマクロを用いることでデータ構造を宣言する.

`DeclareGrid3D(grid_name, type)`

これは 3 次元グリッドを定義する場合であり, 2 次元の場合も同様の記述によって定義する. `grid_name` は定義する型の名前付けのための識別子であり, `type` はグリッドの各要素データ型である. 図 2 では `grid3d_real` という識別子, 各要素が `float` 型として定義される. これにより, `grid_name` という名前のデータ型が定義され, 以降のプログラムコード中から利用可能になる. 例えば, 通常の C プログラムにおける変数として以下の様に定義された型の変数を宣言できる.

```
<grid_name> g;
```

ここで, `<grid_name>`は, `DeclareGrid3D` に渡された識別子で置き換えたものである. `<grid_name>`型の変数には, 次に説明される操作によって生成されるグリッドオブジェクトへのハンドルを保持する. ハンドルを保持する変数を他の変数に代入する場合, 代入された変数にはハンドルのみコピーされ同一のグリッドデータを参照する.

```
1 #define N (64)
2 #define REAL float
3 DeclareGrid3D(grid3d_real, REAL);
4 int main(int argc, char *argv[])
5 {
6   // create a new grid object
7   grid3d_real g = grid_new(grid3d_real,N,N,N);
8   REAL *buff  = (REAL *)malloc(sizeof(REAL)*N*N*N);
9   REAL  time  = 0.0;
10  int   count = 0;
11  REAL l, dx, dy, dz, kx, ky, kz, kappa, dt;
12  REAL ce, cw, cn, cs, ct, cb, cc;
13  l = 1.0;
14  kappa = 0.1;
15  dx = dy = dz = l/N;
16  kx = ky = kz = 2.0*M_PI;
17  dt = 0.1*dx*dx/kappa;
18  // prepare the input grid data
19  init(buff,N,N,N,kx,ky,kz,dx,dy,dz,kappa,time);
20  ce = cw = kappa*dt/(dx*dx);
21  cn = cs = kappa*dt/(dy*dy);
22  ct = cb = kappa*dt/(dz*dz);
23  cc = 1.0-(ce+cw+cn+cs+ct+cb);
24  // transfer the input data to the grid object
25  grid_copyin(g, buff);
26  do {
27    // update the grid by applying stencil to each point
28    stencil_run(stencil_map(map_kernel,g,ce,cw,cn,cs,ct,cb,cc));
29    time += dt;  count++;
30  } while (time + 0.5*dt < 0.1);
31  grid_copyout(g, buff); // transfer the grid data to buff
32  grid_free(g); // destroy the grid
33  return 0;
34 }
```

図 2 3-D ステンシルプログラムの例（ドライバーコード）

3.5 基 本 操 作

`DeclareGrid3D(grid_name, type)` により`<grid_name>`型が導入されるが, それに対する操作として, グリッドの生成破棄, グリッドデータへのアクセスおよびステンシル計算を本フレームワークではサポートする. ステンシル計算を定義する関数を**ステンシル関数**と呼ぶ.

3.5.1 グリッドの生成および破棄

`DeclareGrid3D` によって定義された型のグリッドを作成するためには, 以下の関数を用

Vol.2010-ARC-192 No.7
Vol.2010-HPC-128 No.7
2010/12/16

いる.

```
<grid_name> grid_new(<grid_name>, int X, int Y, int Z)
```

ここで<grid_name>は型名と同様に DeclareGrid3D に渡された識別子で置き換えたものであり, X, Y, Z はそれぞれ 1 次元目, 2 次元目, 3 次元目のサイズを表す整数値である. これにより指定された型およびサイズのグリッドを表すオブジェクトが生成され, そのハンドルが返される. 図 2 の例では一片のサイズ N の正方グリッドを生成する.

生成されたグリッドオブジェクトは明示的に破棄されない限りその生存期間はプログラムの実行終了時までである. オブジェクトを破棄するためには以下の関数を用いる.

```
void grid_free(<grid_name> g)
```

これにより変数 g に保持されたハンドルのグリッドオブジェクトが破棄される.

3.5.2 データの入出力

grid_new によって生成されたグリッドに対して, そのハンドルを介することでデータ全体に対する一括アクセスが可能である.

```
void grid_copyin(<grid_name> g, const <type> *buf)
void grid_copyout(<grid_name> g, <type> *buf)
```

grid_copyin により buf からハンドル g へ<type>型のデータがコピーされる. buf には 1 次元目から順にデータが並んでいるものとし, g と同サイズのデータがコピーされる.grid_copyout ではその逆にハンドル g で表されるグリッドから通常の配列 buf へコピーされる.

図 2 では初めに通常の float 型配列上に初期入力データとして準備し（init 関数呼び出し）, それを grid_copyin によってグリッドオブジェクトへセットする. また終了前には grid_copyout により逆にグリッドデータを float 型配列へコピーする.

3.5.3 グリッドの更新

グリッドに対してステンシル計算により各要素を更新するためには, ユーザはステンシル計算を定義するステンシル関数を記述する. ステンシル関数はグリッドの各点に対する操作として定義され, 通常の C 言語の関数と同様に記述可能だが, 以下の制約に従う.

- 引数として操作対象の点のインデックスとグリッドへのハンドル, およびその他引数を持つ
- 関数内コードはすべてグリッドなどの永続データに対しては読み込みのみ許可し, 変更は許されない
- grid_emit(<grid_name>, <type>) を呼び出すことで点の更新を行う

具体的には,

```
<type> stencil(int x, int y, int z, <grid_name> g, ...)
```

として定義される. ここでステンシル関数の名前は任意であり, 先頭の 3 引数はグリッドの点のインデックスを表し, 第 4 引数が操作対象グリッドである. さらに任意個数のスカラー値の引数を持つことが許される. ステンシル関数内における永続データに対する書き込みを許さないことでステンシル関数内に依存関係が生じることを防ぎ, 並列化のための解析を単純化する.

ステンシル関数内からは関数 grid_get によりグリッド g の 1 要素へのアクセスが可能である. ステンシル計算におけるデータアクセスは隣接データに限られるため, 本関数では隣接アクセスのみを仮定したオフセットによる位置指定を行う. またオフセットはコンパイル時定数と限定する.

```
<type> grid_get(<grid_name> g, int xo, int yo, int zo)
```

これはハンドル g で表されるグリッドオブジェクトの要素 (x+xo, y+yo, z+zo) の値を返す. アクセス要素の位置をステンシル関数の適用対象点からの定数オフセットと表す制約を加えることで, ステンシル関数のデータアクセスパターンをコンパイル時に判定できる. これは特に分散メモリ上にグリッドを分割して持ち, 境界領域の適切かつ効率的な交換を実現するために必要な特性である.

図 3 に拡散方程式を離散化したステンシルを示す. ここでは 3 次元グリッドの各点に対して, その上下前後左右の隣接点の値を用いて更新する. 関数内で用いられている grid_dimx, grid_dimy, grid_dimz はそれぞれグリッドの各次元のサイズを得る関数である.

グリッドのに対するステンシルの適用は以下の stencil_map プリミティブをステンシル関数を指定して呼び出すことで実現する.

```
stencil_object stencil_map(map_kernet f, <grid_name> g, ...)
```

ここで stencil_t は上記ステンシル関数の型であり, 第 1 引数にはステンシル関数を指定する. 第 2 引数には操作対象のグリッドのハンドルを渡し, さらにそれ以降の任意引数としてスカラー値が任意個数指定可能である. ただし, 任意引数はステンシル関数の引数とその個数, 型が一致する必要がある. この際にしてされているステンシル関数 map_kernel は図 3 である.

stencil_map が呼ばれると, ステンシルが実行される代わりに返り値として stencil_object が作られる. このオブジェクトを引数として stencil_run を呼出たときに初めてステンシルが実行される.

Vol.2010-ARC-192 No.7
Vol.2010-HPC-128 No.7
2010/12/16

```
1 REAL map_kernel(int x, int y, int z, grid3d_real g,
2                 REAL ce, REAL cw, REAL cn, REAL cs,
3                 REAL ct, REAL cb, REAL cc)
4 {
5   int nx, ny, nz;
6   nx = grid_dimx(g);
7   ny = grid_dimy(g);
8   nz = grid_dimz(g);
9
10  REAL c, w, e, n, s, b, t;
11  c = grid_get(g, 0, 0, 0);
12  w = (x == 0)    ? c : grid_get(g,-1, 0, 0);
13  e = (x == nx-1) ? c : grid_get(g, 1, 0, 0);
14  n = (y == 0)    ? c : grid_get(g, 0,-1, 0);
15  s = (y == ny-1) ? c : grid_get(g, 0, 1, 0);
16  b = (z == 0)    ? c : grid_get(g, 0, 0,-1);
17  t = (z == nz-1) ? c : grid_get(g, 0, 0, 1);
18  grid_emit(g, cc*c + cw*w + ce*e + cs*s + cn*n + cb*b + ct*t);
19 }
```

図 3 3 次元拡散方程式を計算するステンシル関数

```
void stencil_run(int n, stencil_run_object stencil1,
                 stencil_run_object stencil2, ...)
```

第 1 引数の n はステンシルを実行する回数であり, その後に呼び出すステンシルが順ガンに続く.stencil_run は呼び出されると, 渡されたステンシルオブジェクトを順番に n 回実行する. 例えば次のような関数呼び出し

```
stencil_run(100, stencil_map(kernel1, g1, g2),
                 stencil_map(kernel2, g2, g3),
                 stencil_map(kernel3, g3, g1));
```

があったとき, これは一連のステンシルが kernel1,kernel2,kernel3 の順番で 100 回実行することを意味する.

3.5.4 グリッドのリダクション

グリッドの全要素をリダクションする操作として, 以下の stencil_reduce 関数を提供する.

```
<type> stencil_reduce(<grid_name> g, reducer_t f)
```

ここで第一引数が操作対象グリッドを表し, 第 2 引数がリダクション関数を指定する. リダクション関数では<type>型の引数を 2 つ入力として持ち, 1 つの<type>型の値を返す.

表 1 評価用マシンの構成.

CPU	Model	Intel Core i7 920
	Clock	2.67GHz
	Cores	4 physical cores (8 logical cores)
GPU	Model	Tesla C2050
	Clock	1.15GHz
	Device Memory	3GB
	Compute Capability	2.0
	CUDA Runtime Version	3.10
	CUDA Driver Version	3.10
Host Memory	DDR3 12GB	
Network	Infiniband DDR 20Gb/s	
OS	CentOS 5.3	
ノード数	10	

4. 性能評価

提案フレームワークの性能を評価するために, ステンシル計算の一種, 流体計算の一部である三次元拡散方程式 (1) を対象に性能を評価した. この式は空間のある一点の時間に関するラプラシアンをとったものであり, その点の拡散を表している. 式 (2) はそれを二次精度で離散化したものである.

$$\frac{\partial f}{\partial t} = \kappa \nabla^2 f = \kappa \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \right) \tag{1}$$

$$f^{n+1}_{i,j,k} = f^n_{i,j,k} + \kappa \Delta t \left(\frac{f^n_{i+1,j,k} - 2f^n_{i,j,k} + f^n_{i-1,j,k}}{\Delta x^2} + \frac{f^n_{i,j+1,k} - 2f^n_{i,j,k} + f^n_{i,j-1,k}}{\Delta y^2} + \frac{f^n_{i,j,k+1} - 2f^n_{i,j,k} + f^n_{i,j,k-1}}{\Delta z^2} \right) \tag{2}$$

この問題に対して手動で実装したものと, フレームワークによって自動生成されたものの性能をそれぞれ評価した. 評価環境は**表 1** の通りである. 性能評価に使用した問題は, グリッドのサイズが 256x256x256,256x256x384,256x256x512 の三種類についての Strong Scaling と, グリッドのサイズが 1 ノードが 256x256x256 の Weak Scaling である.

4.1 ベンチマークコード

評価では GPU クラスタ向けに CUDA と OpenMPI を用いて手動で実装したものと, 自動生成されたものの性能をそれぞれ測定した. ベンチマークコードでは Shared Memory は

Vol.2010-ARC-192 No.7
Vol.2010-HPC-128 No.7
2010/12/16

使わず Global Memory だけを使っている. これは Fermi 世代の GPU から導入されたキャッシュが有効に働くため,Global Memory だけで十分な性能が得られるからである.[6].GPU のブロック分割は x,y それぞれ 64,4 で分割している. 手動による実装 (*manual*) では各 CUDA スレッドが z 方向にイテレーションする実装である. フレームワーク生成されたコードでは各 CUDA スレッドで z 方向に z のサイズ分だけイテレーションする実装 (*normal_zloop*) と, 各 CUDA スレッドに対して z 方向を 16 回のイテレーションで固定したもの (*fixed_zloop*) の 2 種類のコードがある. ノード間の並列化については MPI を用いて, 問題領域を z 軸方向に一次元で等間隔で分割する. イテレーション時の袖領域の交換は単純に隣接するノードと行い, 遅延の隠蔽は実装していない.

4.2 最 適 化

手動で実装したコードには次のような最適化が適用されている.

(1) グリッドサイズを定数値で定義し, 計算カーネルでのアドレス計算などに利用
(2) ロードするアドレスを事前に計算

2 の最適化 (図 4) は GPU 固有の最適化である. ロード命令で分岐が発生するとスレッドダイバージェンスが発生し,GPU メモリへのコアレスアクセスが崩れてしまう. アドレス計算を先に分岐で計算しておくことにより, ロード命令で分岐が発生しなくなる. フレームワークではカーネル関数からグリッドの点にアクセスするためには定数値で点の相対位置距離を指定する. そのため, フレームワークの入力としてのこの最適化を適用することはできず, 図 5 のような記述になる.

フレームワークで生成されたコードでは以下の最適化を適用している.

(1) グリッドのサイズが定数の場合はカーネル関数に伝播
(2) 必要なアドレス計算をカーネルの先頭で行う
(3) 各 CUDA Thread が計算する領域の z 方向を固定する

フレームワークでは, ロード命令の分岐を避けるような最適化を自動で行うことは困難である. フレームワークではその変わりに 2 の最適化で, 必要なアドレス計算をカーネルの先頭で行うことで, アドレス計算自体の分岐をなくす最適化を適用している. この最適化は, 図 5 のような分岐ブロックが小さい場合に有効である.3 の最適化は,*fixed_zloop* に適用されている最適化である. この最適化により, 各スレッドブロックの実行時間小さくなり, 常に多数のスレッドブロックが同時に実行されるようになる.

4.3 評 価

図 6 は 256x256x256 の三次元拡散方程式の性能を GPU クラスタ上で Strong Scaling で

```
1 if (condition) {
2   addr = x + y*nx + z*nx*ny;
3 } else {
4   addr = (x - 1) + y*nx + z*nx*ny;
5 }
6 x = f[addr];
```

図 4 ロード命令の分岐をなくす最適化

```
1 if (condition) {
2   x = grid_get(g, 0, 0, 0);
3 } else {
4   x = grid_get(g, -1, 0, 0);
5 }
```

図 5 フレームワークを用いた記述

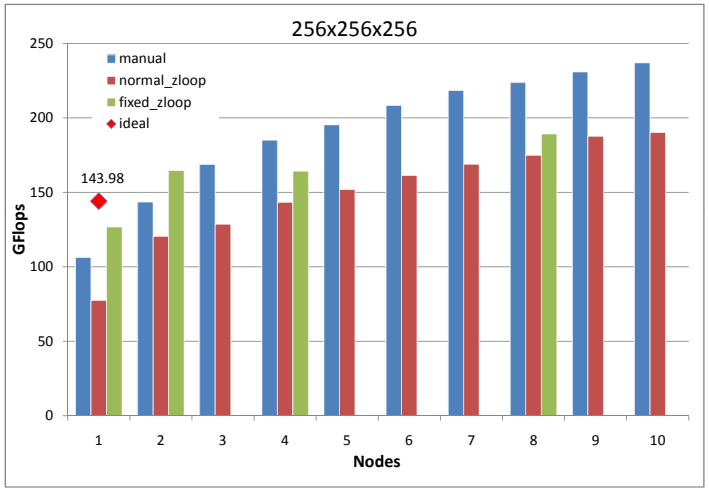


図 6 サイズ 256x256x256 の三次元拡散方程式の性能

計測した結果を表すグラフである. 横軸はノード数であり,1 ノードに対して 1MPI プロセスで実行している. グラフ上の 1 ノード時のひし形は GPU のバンド幅を元にして計算した理論値である. C2050 のメモリバンド幅は実測値で約 84.4GB/s であったので,1 ノードでの理論性能は約 144GFlops となる.*manual* は 1 ノードで 106GFlops の性能であった, これは理論値の約 74%である. 一方で自動生成したものはそれぞれ *normal_zloop* が 77GFlops,*fixed_zloop* が 127GFlops であった. *normal_zloop* は *manual* の約 73%程度の性能であったが,*fixed_zloop* は *manual* の約 125%であった.1 ノードで実行する場合には z 方向のサイズが大きいため,z 方向のイテレーションを各 CUDA スレッド 16 回ずつに分割したする最適化が有効であったと考えられる.

Vol.2010-ARC-192 No.7
Vol.2010-HPC-128 No.7
2010/12/16

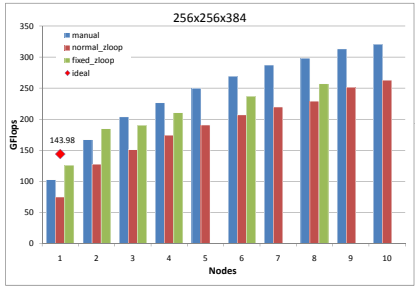


図 7 サイズ 256x256x384

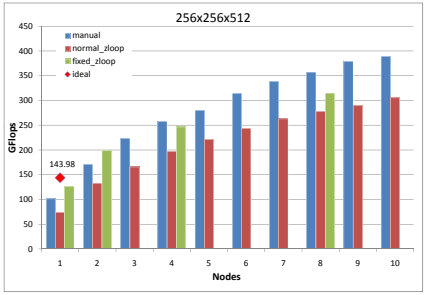


図 8 サイズ 256x256x512

ノード数を増やした場合,*normal_zloop* は常に *manual* の約 73% の性能を維持していた.*fixed_zloop* についてはノード数が増えていくと,*normal_zloop* よりも性能が悪くなった. これは問題の分割を z 方向について行なっているためで, ノード数が増えていくと z 次元が次第に小さくなるため, 最適化の効果が低減したためであると考えられる. なお *fixed_zloop* は z 方向のサイズを固定しているため, ノード数によっては余りが出てしまうため, いくつかのノード数において計測されていない.

グリッドサイズが 256x256x384(図 7) と 256x256x512(図 8) の Strong Scaling についても同様の傾向がみられる. 袖領域交換時の通信がボトルネックとなっているため,1 ノードの計算量が増えることによって, 全体としての見かけの性能は向上している.

図 9 は 1 ノードで計算するグリッドが 256x256x256 である Weak Scaling を計測した結果である.1 ノードで計測した結果と多ノード時の結果が同様の傾向を示している.*fixed_zloop* はほぼ常に *manual* の 120-130%程度の性能が出ている. また, 当然ではあるが Strong Scaling の時と比べよくスケーリングしていることがわかる.

5. 関 連 研 究

High Performance Fortran (HPF) は Fortran 90 を基にデータ並列実行のための言語拡張である.[7] 分散メモリ並列計算機向けに配列の分散をプログラム中に明示的に指定し, それによりコンパイラによるループの自動並列化を実現している我々のフレームワークの設計目標である共有メモリプログラミング, 暗黙的並列プログラミングモデル, 実行環境独立な記述は HPF において広く研究開発されたものであるしかしながら我々の知る限り HPF は今日において広く使われてはいない. その理由は参考文献[7] に詳しい. 我々のフレームワークでは HPF などのデータ並列言語と設計目標を多く共有するが, アプリケーションを限定することで HPF などのより汎用なデータ並列言語に比べて, 本フレームワークにおいて解決する必要がある技術的課題はより単純かつ容易なものに限定可能である.

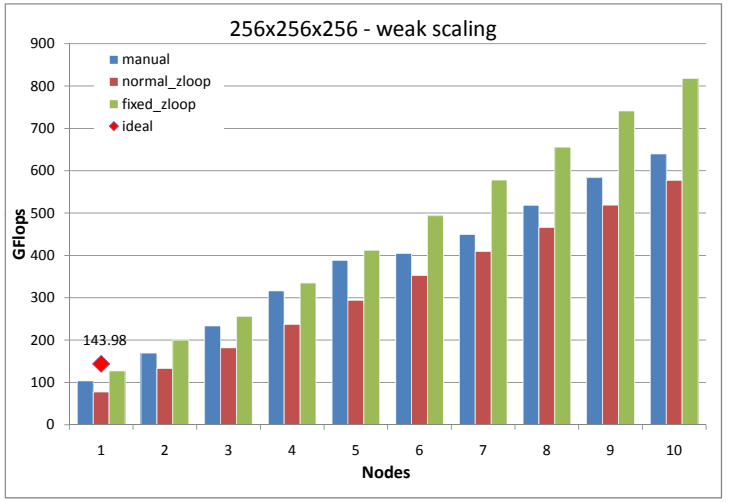


図 9 1 ノード 256x256x256 の Weak Scaling での性能

Kamil らは Chip Multiprocessor(CMP) をターゲットにしたステンシル計算の自動チューニングフレームワークを提案している.[4] 彼らのフレームワークでは Fortran 95 のサブセットといくつかのアノーテーションでステンシルを記述し, それを自動並列化した上でコンパイルするさらに Intel Nehalem, AMD Barcelona, Sun Victoria Falls および Nvidia GTX280 などの異なるターゲットデバイス毎への自動チューニングを実現しているステンシル計算の自動並列化では問題の分割をどのように行い, 袖領域のデータ交換をどう処理するのか重要な鍵であるが, 彼ら研究では共有メモリ型の CMP が対象であるためデータの交換は必要ない. 我々の研究では大規模 GPU クラスタを対象にしているが, 単一ノードでの性能向上も重要な鍵の一つである.

Chapel, X10, Fortress は DARPA の Hight Productivity Computing Systems プログラムの一部であり,HPC 向けの新しい言語を開発するプロジェクトである.[8),9)] 既存のプログラミング言語は大規模クラスタのような並列性を仮定してい設計されていないため, それらを利用するためのプログラミングが困難になっている. これらプロジェクトでは HPC 向

Vol.2010-ARC-192 No.7
Vol.2010-HPC-128 No.7
2010/12/16

けに全く新しい抽象度の高い言語の設計を試みている. 我々の研究では既存の広く使われている言語の上にアプリケーションドメインを限定した並列化フレームワークを提供するという異なったアプローチをしている.

6. 結論と今後の課題

本研究では簡潔な記述でステンシル計算を記述し, 自動で GPU クラスタ向けにコードを生成するフレームワークを提案した. 提案したフレームワークを実装し, 流体計算の一部である三次元拡散方程式フレームワークによって生成されたコードと手動によるコードを GPU クラスタ上で性能評価を行なった. フレームワークによって生成されたコードは手動で実装したものと比べて 1 ノード時, また Weak Scaling については 125%程度の性能であった.Strong Scaling についてはノード数と最適化の種類によって 75%-125%程度の性能であった.

今後フレームワークに更なる最適化やオートチューニングなどを実装していく予定である. また, ベンチマークとして実アプリケーションに近い複雑なものをフレームワークを使って実装し, 性能評価を行う予定である.

謝辞 本研究の一部は科学技術振興機構戦略的創造研究推進事業「Ultra-Low-Powr HPC: 次世代テクノロジのモデル化・最適化による超低消費電力ハイパフォーマンスコンピューティング」, および NVIDIA CUDA Center of Excellence による。

参 考 文 献

1) : http://www.top500.org.
2) Shimokawabe, T., Aoki, T., Muroi, C., Ishida, J., Kawano, K., Endo, T., Nukada, A., Maruyama, N. and Matsuoka, S.: An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code, *SC10* (2010).
3) 越塚誠一：計算力学レクチャーシリーズ 5 粒子法 (2007).
4) Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, Samuel Williams: An Auto-Tuning Framework for Parallel Multicore Stencil Computations, *IEEE International Parallel & Distributed Processing Symposium - IPDPS 2010* (2010).
5) 小川 慧，青木尊之，山中晃徳：マルチ GPU によるフェーズフィールド相転移計算のスケーラビリティ，情報処理学会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 (HPCS2010)，pp.117–124 (2010).
6) 野村 達雄, 丸山 直也, 遠藤 敏夫, 松岡聡：GPU クラスタを対象にした並列ステンシル計算の自動生成フレームワーク，*SWoPP 2010*.
7) Kennedy, K., Koelbel, C. and Zima, H.: The rise and fall of High Performance Fortran: an historical object lesson, *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ACM, pp.7–1–7–22 (2007).
8) Weiland, M.: Chapel, Fortress and X10: novel languages for HPC, Technical report, EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, UK (2007).
9) F.Barrett, P. C.Roth, S. W.P.: Finite difference stencils implemented using chapel, Technical report, ORNL Technical Report (2007).