Vol.2010-HPC-126 No.9
2010/8/3

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

野村達雄[†1]　丸山直也[†1]
遠藤敏夫[†1]　松岡聡[†1,†2]

流体計算のカーネル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, 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の恩恵を受けやすい．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, 並列化のために問題の分割や袖領域の交換など計算以外のコードを大量に書く必要がある．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．本研究ではステンシル計算本来の簡潔な記述を保ったまま,GPU に関する知識を必要としないコードから自動で 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．その結果手動による実装の 3 分の 1 程度記述で 70%程度の性能を達成できた．今後フレームワークに最適化を実装していくことによって更なる性能向上を目指す予定である．

Towards an Automatic Code Generation Framework for Parallel Stencil Computations on GPU Clusters

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

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, since it can benefit from using GPU as an accelarator. Although stencil computations themselves can be described concisely, we have to write huge amount of code which forms parallelization such as domain decomposition and boundaries exchanging. Those difficulties confine the utilization of GPU to a handful of people who has 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 picked 3d-diffusion-equation as an example problem for evaluation. We evaluated the performance of its two implementations; One is implemented manually, and another is auto-generated by our framework. We have allowed the code size to be reduced to one-third approximately, and achieved about 70% of the performance of hand-coded implementation. We are planning to implement optimizations for more performance gain as the future work.

1. 背　　景

近年 GPU を汎用目的で使う GPGPU がさまざまな分野で活発になっている．HPC の分野では GPU クラスタが台頭し,GPU が計算能力の目覚しい向上の一役を買っている．その一方で,CPU のみを対象にした環境よりもプログラミングが更に複雑化している．アプリケーションの開発者は CPU 向けのコード,GPU 向けのコード, そして並列化のための 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する必要がある．

科学計算の計算カーネル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.[1)] しかし,GPU クラスタを活用するために開発者は 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元に GPU クラスタ向けにコードを自動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カーネルを受け取り,GPU クラスタ向けに C,CUDA, そして MPI コードを生成する．

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

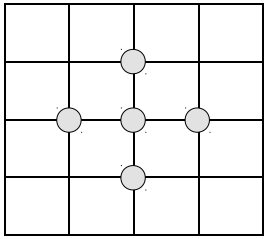
Vol.2010-HPC-126 No.9
2010/8/3

本研究では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. その結果自動生成されたものは手動のものと比べて実行時のノード数により 40%-70%の性能であった. また, 自動生成のためのコードは手動のものと比べて約 3 分の 1 程度の記述量であった.

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. 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. これらの性質は広い帯域幅をもつ GPU の特徴によく符合しているため,GPU を利用した流体計算は適切に実装すれば,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.[1)]

今のところ GPU を利用する場合には開発者が CPU のコードと GPU のコードを書かなければならない. 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 と GPU の通信, 境界条件の処理などを実装しなければならないため, 実際のコーディング量は計算以外の部分が多くを占める. 最適化, チューニングを加えるとコーディングの量と複雑さは更に増すことになる. 単一ノードに対しては,Kamil ら[3)] 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. これについては後ほど関連研究で詳しく述べる. 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[4)] 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.



```
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.

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.

本節では C 言語の文法に従った記述方法について説明し, 次節にその実装方法を述べる.

3.1 設計方針

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

⋆1 希薄流体を扱う場合など, 粒子法[2)] 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

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．

3.2 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．

3.3 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(key, 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．`key` 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 `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, `grid3d_key` 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．

```
grid3d_<key> g;
```

ここで, `<key>`は, `DeclareGrid3D` に渡された識別子で置き換えたものである．`grid3d_<key>`

```
1 #define N (64)
2 #define REAL float
3 DeclareGrid3D(real, REAL);
4 int main(int argc, char *argv[])
5 {
6   // create a new grid object
7   grid3d_real g = grid3d_real_new(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    grid_update(stencil,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）

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.

3.4 基本操作

`DeclareGrid3D(key, type)` により `grid3d_<key>`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.

3.4.1 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, 以下の関数を用いる.

```
grid3d_<key> grid3d_<key>_new(int X, int Y, int Z)
```

ここで`<key>`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 grid3d_<key>_free(grid3d_<key> 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. また, 短縮記法として `grid3d_<key>_free` の代わりにマクロ `grid_free` を用いることも可能である.

3.4.2 データの入出力

`grid3d_<key>_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 grid3d_<key>_copyin(grid3d_<key> g, const <type> *buf)
void grid3d_<key>_copyout(grid3d_<key> g, <type> *buf)
```

`grid3d_<key>_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. `grid3d_<key>_copyin` 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. また, 短縮記法として `grid_copyin` および `grid_copyout` が利用可能である.

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.

3.4.3 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
- 返り値は操作対象グリッドの要素の型と一致する

- 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

具体的には,

```
<type> stencil(int x, int y, int z, grid3d_<key> 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 grid3d_<key>_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> grid3d_<key>_get(grid3d_<key> 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. また, 短縮記法として grid_get もマクロとして提供される. 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マクロである.

グリッドの更新は以下の update 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.

```
void grid3d_<key>_update(stencil_t f, grid3d_<key> 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. また, 簡便性のために grid_update マクロを grid3d_<key>_update の代わりに用いることができる. 図 2 ではループ内で一定回数グリッドの更新を行うために, 上記関数を繰り返し呼び出す. この際にしてされているステンシル関数 stencil は図 3

```
1  REAL stencil(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   return 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

である.

上記関数の変形としてグリッドの更新を指定された回数繰り返す以下の関数も利用可能である.

```
void grid3d_<key>_update_nth(stencil_t f, int n, grid3d_<key> g, ...)
```

これは指定されたステンシル関数を n 回グリッド g に適用するものである.

3.4.4 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 reduce 関数を提供する.

```
<type> grid3d_<key>_reduce(grid3d_<key> 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.

4. フレームワークの実装

提案フレームワークはユーザプログラムを各種実行環境向けにソースコードレベルで変換する. 本論文では GPU クラスタを対象とした変換手法について述べる. 対象とする GPU クラスタは複数の計算ノードに同性能の GPU が複数接続され, それらの計算ノード間は MPI により通信が可能とする. 本実装では接続されている GPU はすべて同一の性能を持つと仮定し, 異種 GPU を使った場合の負荷分散については考慮しない. また, 本フレーム

ワークでは本質的には GPU に限らず CPU も計算に参加させられるが, 本実装では GPU のみを対象とする.

本フレームワークでは GPU クラスタ上の複数 GPU を用いるために 1 枚の GPU 毎にローカル CPU 上に 1 プロセスを排他的に割り当て, それらの CPU プロセスを MPI により並列プログラムとして動作させる. ノード内に複数 GPU を搭載する場合においても本実装では GPU の枚数分 MPI プロセスを用いる.

4.1 データ表現

ユーザプログラム中に宣言されたグリッドデータ型を GPU クラスタ向け実装として実現する. GPU クラスタ上の複数の GPU デバイス間ではそのメモリ空間が共有されない. 従って, 利用可能 GPU に対して均等に計算対象グリッドを分割し, それらの間で PCI バス, ホストメモリ, インターコネクトを介して適宜データの交換, 同期を行う. グリッドの分割方法としては本実装では均質な性能を持つ GPU を仮定するため, 1 次元の均等分割を行う. 具体的には, GPU 単体が持つ部分グリッドを C 言語の構造体として表現し, ホスト CPU 上の MPI プロセスから GPU メモリへの参照を管理する. また, ステンシル関数の適用を並列に実行するため, 部分グリッド領域をダブルバッファとして確保し計算に用いる.

一般には多次元分割の方が通信コストを削減できるが, プログラミングが煩雑になる. 本フレームワークではそのような煩雑なプログラミングをフレームワーク側で自動処理することを目的としているが, 現状の実装では 1 次元分割のみに対応している. 多次元分割および性能不均質 GPU への対応は今後の課題である.

4.2 基 本 操 作

3.4 節で述べた本フレームワークにおける各種操作の GPU クラスタにおける実装について説明する.

4.2.1 グリッドの生成および破棄

グリッドの生成用ルーチン `grid_new` は, 各 GPU が担当する部分グリッド用の領域を GPU メモリ上に確保する. これは全 MPI プロセスにおいて, `cudaMalloc` を呼び出すことで実現する. ステンシル計算では離散化した各点の計算に隣接した点の値を用いるため, 分割したグリッドの境界に位置する点の計算には隣接部分グリッドの点, すなわち袖領域の値も必要である. GPU メモリ空間は個々のデバイス間で分離されているため, ホストを介して隣接部分グリッドを担当する他の GPU から該当データを転送する必要がある. 本実装では複数 GPU を用いたステンシル計算の既存実装と同様に[4),5)], ステンシル計算を始める前に部分グリッドに必要な袖領域を GPU メモリに転送し, そのために部分グリッドの領域を袖領域を含めたサイズとして確保する.

GPU メモリ上に袖領域を含めた部分グリッドのメモリ領域を確保するために, ステンシル関数を解析し袖領域のサイズをフレームワークによるコンパイル時に特定する. 3 節で述べたように, 本フレームワークではステンシル関数によるグリッドへのアクセスは各点からコンスタントオフセットによるものと制約を課したため, コンパイル時に袖領域のサイズを特定可能である. 具体的にはまず入力ソースプログラム中における各グリッドオブジェクトについてそのステンシル関数を特定する. これは `grid_new` によって生成されたグリッドオブジェクトに対して `grid_update` の呼び出しを検出し, それに引数として指定されている関数を特定すればよい. 特定したステンシル関数について, そのプログラムコードを解析し, `grid_get` に対する引数の各次元毎の最小値, 最大値を求める. これにより各次元毎の袖領域のサイズを特定する.

4.2.2 データの入出力

グリッドへの一括データ入出力を提供する `grid_copyin` および `grid_copyout` の実現について述べる. `grid_copyin` ではドライバコード内に宣言されたバッファからグリッドへデータを転送する. GPU クラスタではグリッドを表す領域はクラスタ上の複数 GPU に分割して構成されるため, ドライバコードより適切な MPI プロセスが管理する部分グリッドへ転送する. `grid_copyout` では逆に複数ノード上に分散した部分グリッドをドライバコード内に宣言されたバッファに転送する. 各 MPI プロセスから担当する部分グリッドをルートプロセスに転送することで実現する.

4.2.3 グリッドの更新

グリッドの更新のためのプリミティブである `grid_update` および `grid_update_nth` を実装するために, 本フレームワークでは以下の処理を行う.

(1) ステンシル関数の変換
(2) ステンシル関数を呼び出す CUDA グローバル関数の生成
(3) グローバル関数を呼び出す CPU 用関数の生成

ステンシル関数は上述した解析により特定した関数であり, これを CUDA の GPU 用関数に変換する. 具体的には CUDA では GPU 関数には`__device__`というキーワードを指定する必要があるため, これを関数の先頭に付与する. また, 関数内では `grid_get` によりグリッド要素のアクセスが発生するが, これを部分グリッドバッファへの参照へと変換する.

次に, 各 CUDA スレッドのエントリ関数となる CUDA グローバル関数を生成する. 同関数は上記ステンシル関数を用いて部分グリッド内の担当領域の更新を行う. CUDA におけ

るグリッドのアップデートの実装手法としては, 各 CUDA スレッドがグリッドの 1 要素を担当する手法や, 複数要素を担当する手法が考えられる. 現在の実装では簡便化のために 3 次元グリッドの場合は第 1 次元, 第 2 次元については個々にスレッドを割り当て, 第 3 次元については全要素を 1 スレッドが担当する構成となっている.

最後にグローバル関数を用いてグリッドをアップデートする CPU 用関数を生成し, 入力プログラム中の `grid_update` への呼び出しを同関数への呼び出しに変換し, GPU メモリ上のダブルバッファを交換する. `grid_update_nth` の場合は `n` 回呼び出しを繰り返す.

4.2.4 グリッドのリダクション

グリッドの要素のリダクションを計算する `grid_reduce` では, `grid_update` の場合と同様にまずリダクション関数を検出し, それを GPU 関数へと変換する. 次に同関数を用いて部分グリッドのリダクションを実装する CUDA グローバル関数を生成する. 最後に, CPU 用関数として, グローバル関数を呼び出し部分グリッドのリダクションを行い, さらに MPI を用いて全 MPI プロセスのリダクションを行う関数を生成し, 入力プログラム中の `grid_reduce` への呼び出しを同関数へと変換する.

5. 性能評価

本研究の提案しているフレームワークの性能を評価するために, 流体計算の一部である三次元拡散方程式 (1) を取り出して評価した. 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. 本研究で実装したフレームワークはまだプロトタイプの段階であり, 生成したコードは最適化されない. したがって, 性能は手動で記述したものと比べると遅くなってしまう. 現段階では主にステンシル計算の記述方法に注目しており, 簡潔な記述で GPU クラスタ上で動くコードが生成されることを確認するための評価である.

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
Host Memory	12GB	
Network	Infiniband DDR 20Gb/s	

5.1 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. GPU 向けのコードは Shared Memory を使っておらず,Global Memory だけを使っている. その理由については次の節で詳しく述べる. GPU のブロック分割は x,y それぞれ 64,4 で分割し, 各 CUDA Thread が z 方向にイテレーションする実装である. 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. ノード数で割り切れなかった分については MPI の Rank が一番低いプロセスが担当する. イテレーション時の袖領域の交換は単純に隣接のプロセスと行い, 遅延の隠蔽は実装していない. コードの記述量は自動生成のものは約 110 行であり, 手動で記述したものは約 320 行であった. 単純な比較ではあるが, 自動生成のタメのコードは手動実装の約 3 分の 1 程度の記述量であった.

5.2 評　　価

図 4 は 1 ノードの GTX285 と C2050(Fermi) で手動で実装したもののそれぞれの性能を評価したグラフである. 理論値は GPU のメモリバンド幅の実測値を元に計算している. GTX285 のメモリバンド幅は実測値で約 124GB/s であるのに対して,C2050 は約 79GB/s である. GTX285 では Shared Memory を使ったものは Global Memory を使ったものの約 2.6 倍の性能が出ており, 理論値の約 46%の性能である. 一方で C2050 では Shared Memory を使ったものは Global Memory を使ったものよりも遅くなっており, 約 90%の性能である. これは NVidia の GPU は Fermi 世代より L1,L2 キャッシュが実装されたため,Global Memory を使った時でも性能が出るようになったためである. Shared Memory を使った場合にはブロックの境界をチェックするための分岐がオーバーヘッドとなり,Global Memory を使ったものより遅くなっていると考えられる.

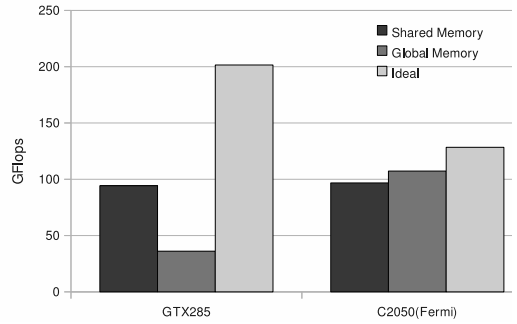


図 4 Global Memory と Shared Memory を使った場合のそれぞれの性能.

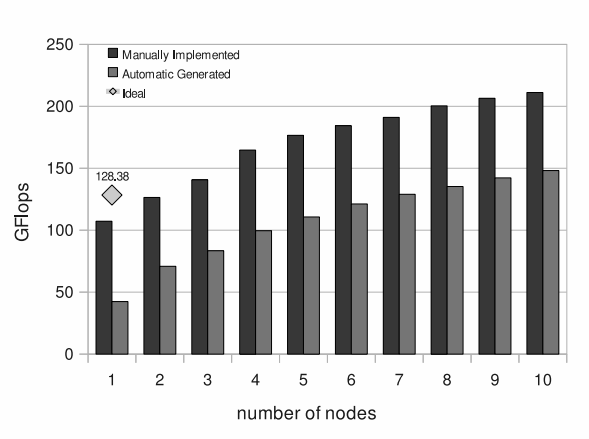


図 5 サイズ 256x256x256 の三次元拡散方程式の性能.

図 5 は 256x256x256 の三次元拡散方程式の性能ををを GPU クラスタ上で測定したグラフである. 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. 手動で実装したコードは Shared Memory を使っていないが, 理論値の約 84%の性能が出ている. 1 ノードで実行した場合の性能は手動で実装したものが約 107GFlops であるのに対して, 自動生成したものは約 42GFlops と約 40%程度の性能である. これは自動生成したコードでは実装上アドレス計算が点にアクセスするごとに発生していることが主なオーバーヘッドになっていると考えられ, 今後の改善していく予定である. しかし,10 ノードで実行した場合は, 手動のものが約 211GFlops であるのに対して, 自動生成したものは約 148GFlops と約 70%程度の性能である. ノード数を増やした時のスケーラビリティがあまり高くないのは袖領域の交換時に発生する通信の遅延が原因である. ステンシル計算ではイテレーション毎に袖領域のデータを交換する必要があるため, ネットワークのバンド幅がボトルネックになってしまう. グラフを見ると双方のスケールの仕方がほぼ同じであるため, 袖領域の交換によるオーバーヘッドは手動で実装したものと自動生成したもので同程度であると言える. したがって, 性能差は主に GPU のコードによるものと考えられる.

6. 今後の拡張

本研究で提案するフレームワークの実装は現段階ではまだプロトタイプであるため最適化は実装されていない. そのため手動で実装したものと比べて性能が悪くなっている. 今後は本フレームワークに最適化やオートチューニングを実装することで性能向上を目指す予定である. 最適化は単一 GPU の固有部分と, 並列化した時の全体の計算環境に対する最適化の二つの部分に大きく分けることができる. 単一 GPU に対する最適化としては分岐の削減, ブロックサイズの自動探索, インデックス計算の簡略化などが考えられる. 本フレームワークでは計算グリッドのサイズを定数で定義するため, コンパイル時に静的にインデックス計算をある程度定数に置き換えることができる. 将来的には実行時にコード生成をすることにより, 問題サイズを変数にしても定数時と同様な利点を得られるようにする予定である. これによって開発者が手動で実装することが困難な最適化を行えるものと考えている. また GPU ではスレッドブロックのサイズは性能を大きく左右するため, 通常は開発者が手動で最適なブロックサイズを試行錯誤して決めている. 本フレームワークでは将来的に自動で最適なブロックサイズを探索できるようにしたいと考えている. 全体の最適化としては袖領域交換時の遅延の隠蔽が主要な最適化となる. 前節で見たように, 単一ノードでは GPU のメモリバンド幅による理論値に近い性能を達成することが可能であるが, 複数ノードで実行した場合はネットワークがボトルネックとなってしまう. 袖領域の交換による遅延を隠蔽出来れば複数ノードで実行した時に高いスケーラビリティが期待できる. また, その他に CPU

を計算に参加させることも視野に入れている.

7. 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.[6] 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 Kennedy らによると,[6] 多くの先進的な研究成果や当時は先進的であった Fortra 90 に依存した設計が 90 年代のコンパイラ技術では十分な性能および並列化を達成できなかったことが挙げられている. また同文献ではそれにも関わらず規則的なグリッドにより構成される問題だけでなく, 疎な問題を含む多くの様々なアプリケーションの自動並列化をサポートすることを目的とし, より多くの問題分割方法のサポートなど言語機能のさらなる拡張が必要であったことが挙げられている. さらに実際問題としてはそのような高機能並列言語をサポートするコンパイラの開発が困難であったことが HPF の普及を妨げた一つの要因であったとされている. 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, GPU クラスタを対象としてそれらのための自動並列化および分散共有メモリを実現可能な範囲に対象アプリケーションドメインを限定する. 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. 本フレームワークの設計および実装はまだ初期段階にあるが, 今後ステンシル計算を基にした実際のアプリケーション[1] の本フレームワーク上の実装などを通して, 設計目標の実現可能性と維持しつつフレームワークの汎用性を高めるための設計の拡張を行っていく予定である.

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.[3] 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. Ravi ら[7] はステンシル計算とは違うクラスの問題の自動並列化を研究している. 彼らの研究では Generalized Reduction Computations のクラスの問題に対して CPU と GPU を協調させたコンパイラとランタイムシステムを提案している. ユーザは基本的には Reduction 関数を実装するだけでよく, 問題の分割やタスクのスケジューリングなどはすべてランタイムシステムが行う. 問題の例として k-means と主成分解析 (Principal Component Analysis) を実装し, 評価を行っているが, この種の問題は通常単純に問題を分割できるため, 比較的並列化が容易である. 彼らの研究では分割した問題を配分し,CPU と GPU 協調動作させることでパフォーマンスを向上させている. 我々の研究ではステンシル計算をターゲットに設定し, 実行環境を GPU を含むクラスタに広げている. ステンシル計算は通常, 問題を分割したとき, 分割された領域間で必ず袖領域の交換が必要になる. そこで CPU と GPU をうまく協調させることによってデータ転送の遅延を隠蔽し, 高いパフォーマンスを得ることができる.

8. 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. フレームワークのプロトタイプを実装し, 三次元拡散方程式を問題の例として GPU クラスタ上で性能評価を行った. 自動生成のためのコードは手動で実装を行ったものの 3 分の 1 程度の記述量であり,10 ノードを用いた場合に手動で実装したものの約 70%の性能であった.

現時点本フレームワークは自動並列化に留まっているが, 今後は最適化やオートチューニングを実装していく予定である. GPU 単体向けの最適化としてブロックサイズの自動探索や, インデックス計算の簡略化, 分岐の最適化などを行っていく. また, 計算環境全体に対する最適化として, 問題の分割方法の探索や袖領域の交換による遅延の隠蔽なども実装して行く予定である.

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) 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). To appear.
2) 越塚誠一：計算力学レクチャーシリーズ 5 粒子法 (2007).
3) 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).
4) 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).
5) Phillips, E. and Fatica, M.: Implementing the Himeno Benchmark with CUDA on GPU Clusters, *IEEE International Parallel & Distributed Processing Symposium*, pp.1–10 (2010).
6) 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).
7) Vignesh T. Ravi, Wenjing Ma, David Chiu, Gagan Agrawal: Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations, *International Conference on Supercomputing - ICS 2010* (2010).