ステンシルフレームワークによる性能と生産性の両立

丸山直也 （理研AICS/東工大/JST CREST）

ワークショップ　ー地球流体計算の今後ー

神戸大学惑星科学研究センターー

2014年3月11日

流体現象のシミュレーションに至るまで

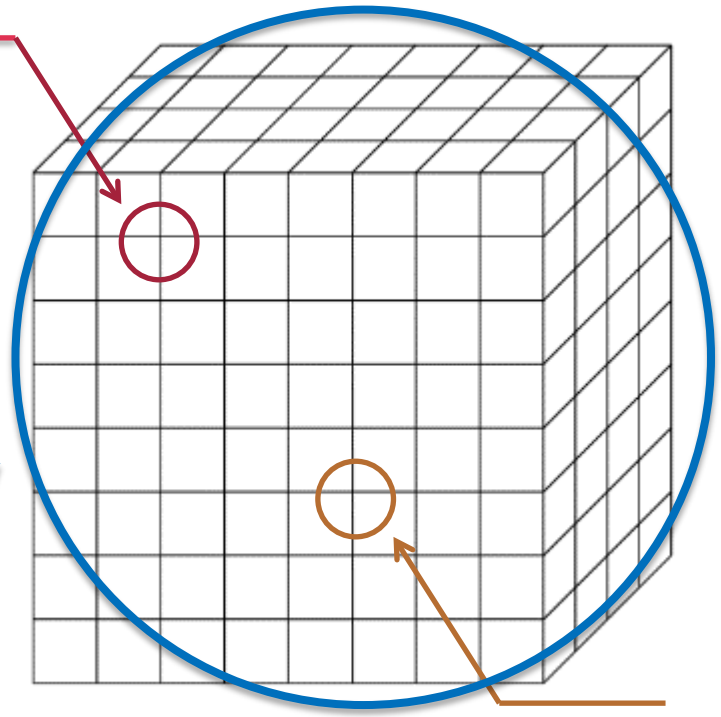
1. 対象現象の方程式による記述

2. 方程式の離散化

3. プログラム記述

ステンシル計算に基づいた流体シミュレーションの構成

2. 各点に物理量を定義（速度、力など）



1. シミュレーション対象空間を定義

3. 各点の更新方法を定義

4. 全体を制御するプログラム（入出力、収束判定、その他）

流体現象のシミュレーションに至るまで

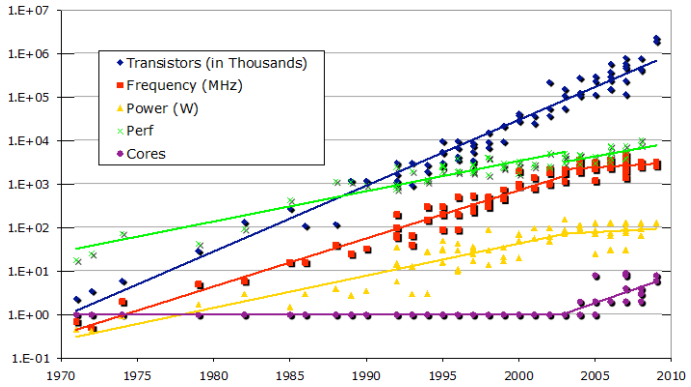
1. 対象現象の方程式による記述

2. 方程式の離散化

3. プログラム記述
    1. 対象空間の定義
    2. 格子点物理量の定義
    3. 格子点更新方法の定義
    4. 全体制御プログラムの記述

流体シミュレーションプログラムの記述

- ステンシル計算自体は単純
  - 各格子点の隣接要素を使った算術演算
  - データ並列性による並列化への適正
    - MPI、OpenMP並列容易

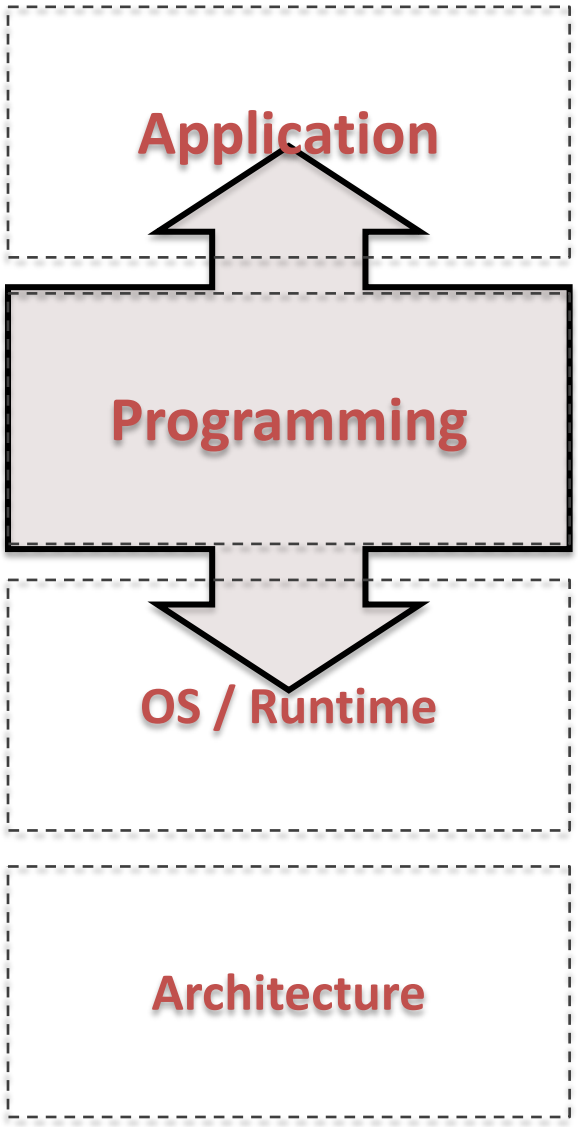


- データ移動コストの増大
  - 計算の並列化 != 性能向上
  - データ移動最小化が重要
- 並列アーキテクチャの多様化
  - GPU等のアクセラレータ
  - アーキテクチャ毎に最適化方針が異なる

現状

- 現在のメインストリーム（？）
  - 各アーキテクチャ毎に人手によるプログラミング
    → 以下のトレードオフ
    - 手間
    - アーキテクチャの将来性
    - 移植による性能向上
  - 指示文ベースのプログラミングモデルにより生産性の改善→性能最適化にはアーキテクチャ固有最適化が必要（[CCGrid'13])
- 研究レベル
  - コンパイラによる自動GPUコード生成＆最適化

ドメイン特化型プログラミング



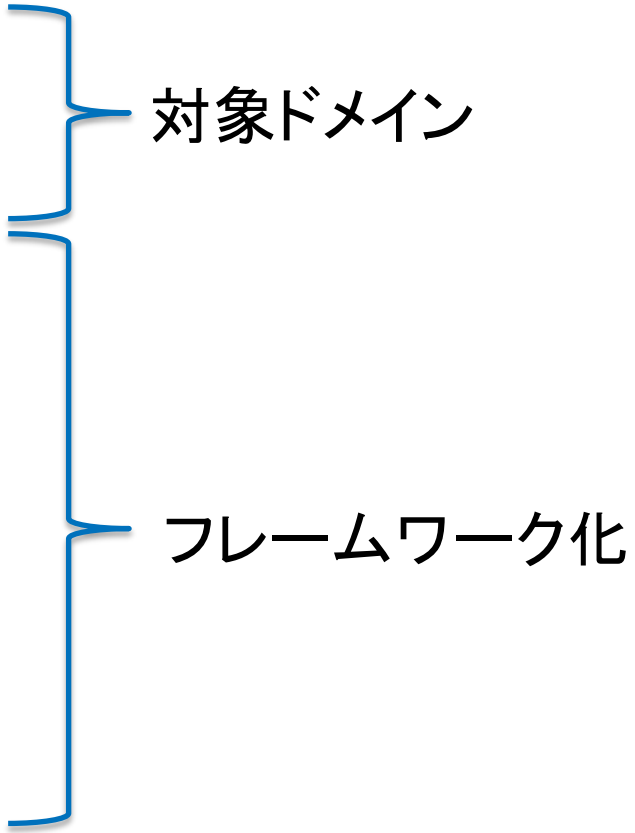
- 特定ドメインに特化することで、アプリケーションドメインのボキャブラリでプログラムを記述可能に
- 汎用性と生産性のトレードオフ
- 例
  - 計算科学： Matlab、R、BLAS、FFTW、OpenFOAM
  - 一般： SQL, Ruby on Rails, LaTeX, Make
  - その他研究レベルでは多くが出現
- 実現方法
  - 言語、フレームワーク、ライブラリ
  - ドメインによって異なる

流体現象のシミュレーションに至るまで

1. 対象現象の方程式による記述

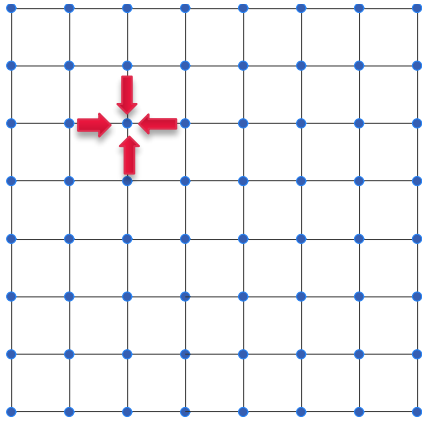
2. 方程式の離散化

3. プログラム記述
    1. 対象空間の定義
    2. 格子点物理量の定義
    3. 格子点更新方法の定義
    4. 全体制御プログラムの記述



Physis ステンシルフレームワーク

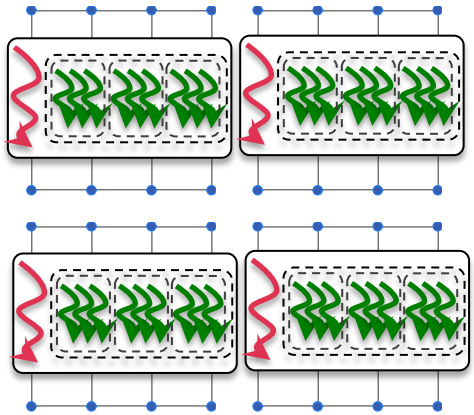
Paper @ SC'11　Code @ http://github.com/naoyam/physis



Stencil DSL

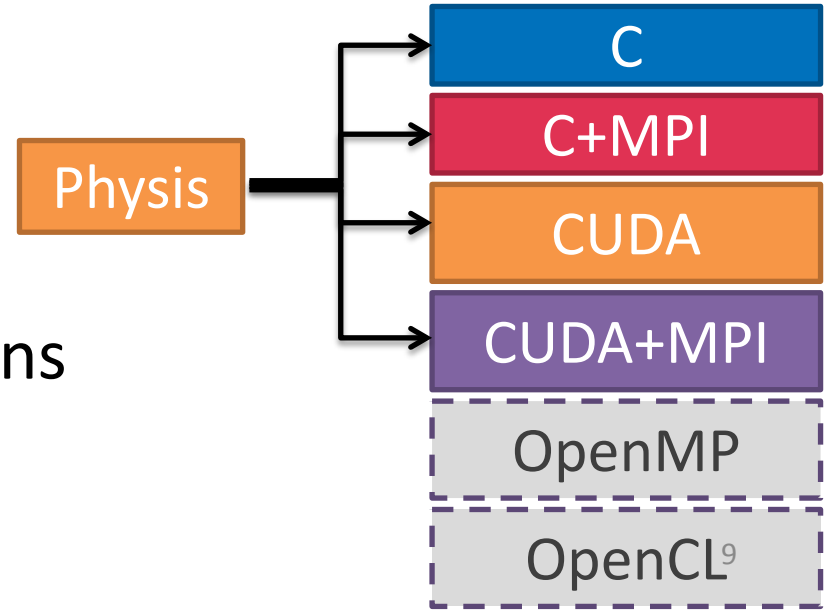
- Declarative
- Portable
- Global-view
- C-based

```
void diffusion(int x, int y, int z,
    PSGrid3DFloat g1, PSGrid3DFloat g2) {
float v = PSGridGet(g1,x,y,z)
 +PSGridGet(g1,x-1,y,z)+PSGridGet(g1,x+1,y,z)
 +PSGridGet(g1,x,y-1,z)+PSGridGet(g1,x,y+1,z)
 +PSGridGet(g1,x,y,z-1)+PSGridGet(g1,x,y,z+1);
PSGridEmit(g2,v/7.0);
}
```



DSL Compiler

- Target-specific code generation and optimizations
- Automatic parallelization



フレームワーク設計指針

- 種々のステンシルに対応
- 計算科学における親和性
- 並列性のプログラマによる明示的な記述
  - Cf. OpenMP vs. コンパイラ自動並列化
- 並列化方法は記述させない
- アーキテクチャ独立

ステンシルをプログラミング可能である必要あり
ライブラリでは不十分

C/Fortranのような記述

宣言的な記述

自動コード生成

Physis DSL

- C言語を限定的に拡張したドメイン特化型言語
- 以下の機能を提供
  1. 対象空間の定義
  2. 格子点物理量の定義
  3. 格子点更新方法の定義

1. 対象空間の定義

- 要件
  - 多次元密直交空間の定義
- Physis
  - 型 PSDomain*N*D により定義
  - 例：N^3の3次元空間

```
PSDomain3D d = PSDomain3DNew(0,N,0,N,0,N);
```

  - 例： N^3空間内の2次元平面

```
PSDomain3D p = PSDomain3DNew(0,N,0,N,0,0);
```

2. 格子点物理量の定義

- 要件
  - シミュレーション問題によって定義する物理変数は異なる→ユーザ定義の必要性
  - 実際の値の持ち方はプログラマから隠蔽
    - Cf. SoA vs. AoS
- Physis
  - C言語構造体として変数を定義
  - 構造体を点に持つ格子の型を宣言
    - DeclareGrid*ND*(構造体の型，格子の型)
  - 定義した格子型のオブジェクトを生成、破棄
    - PSGrid3DTypeNew，PSGridFree
  - 例： 速度vx，vy，vzを持つ3次元格子

```
struct point {double vx, vy, vz};
DeclareGrid3D(struct point, Point);
// Grid3DPoint型が以降利用可能
```

2. 格子点物理量の定義

- 例： 速度vx，vy，vzを持つ3次元格子

```
struct point {
  double vx, vy, vz
};
DeclareGrid3D(struct point, Point);
// PSGrid3DPoint型が以降利用可能

void foo() {
  PSGrid3DPoint g=PSGrid3dPointNew(N,N,N);
  ...
  PSGridFree(g);
}
```

データ構造の性能可搬性

```
PSGrid3DFloat u;
PSGrid3DFloat v;
PSGrid3DFloat w;
PSGrid3DFloat x0, x1, x2, ..., x18;
...
PSGridGet(u, i, j, k);
```

w/o user-defined point type

```
struct Point {
   float u;
   float v;
   float w;
   float x[19];
};
DeclareGrid3D(Point, struct Point);
PSGrid3DPoint g;
PSGridGet(g.u, i, j, k);
PSGridGet(g.x[l], i, j+1, k);
```

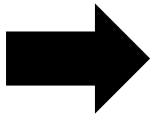
w/ user-defined point type

データ構造の性能可搬性

- SoA vs. AoS
  - GPU → SoA
  - CPU → AoS (?)

```
struct Point {
    float u;
    float v;
    float w;
    float x[19];
};
DeclareGrid3D(Point, struct Point);

PSGrid3DPoint g;
...
PSGridGet(g.u, i, j, k);
PSGridGet(g.x[l], i, j+1, k);
```



Automatic translation to SoA layout

```
struct PointGrid {
    float *u;
    float *v;
    float *w;
    float *x;
};
...
// PSGridGet(g.u, i, j, k);
g->u[i+j*nx+k*nx*ny]
// PSGridGet(g.x[2], i, j+1, k);
g->x[i+j*nx+k*nx*ny+l*nx*ny*nz];
```

3. 格子点更新方法の定義

- 要件
  - ステンシル操作をC言語で記述
  - 明示的なアクセスパターン
    - 最適化に利用
- Physis
  - 各点におけるスカラー計算をC言語の関数として定義
    - N次元ドメインの点に対する操作→N個のダミーインデックス引数
    - Cf. elemental subroutines in Fortran 95
  - `PSGridGet`: 格子点の読み込み
  - `PSGridEmit`: 格子点の書き込み
  - `PSStencilMap`: ドメイン上でステンシル関数を実行
    - 派生として `PSStencilMapRedBlack` など

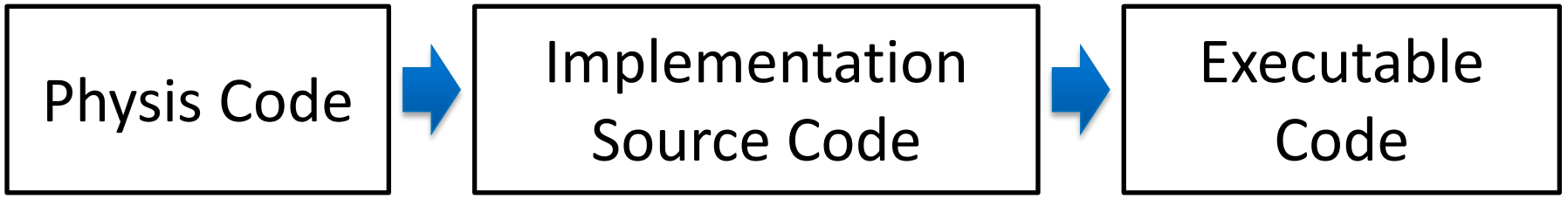
例

```
void kernel(const int x, const int y, const int z,
            PSGrid3DPoint g1, PSGrid3DDouble g2) {
  double v = PSGridGet(g1,x,y,z).vx
    +PSGridGet(g1,x-1,y,z).vx + PSGridGet(g1,x+1,y,z).vx
    +PSGridGet(g1,x,y-1,z).vy + PSGridGet(g1,x,y+1,z).vy
    +PSGridGet(g1,x,y,z-1).vz + PSGridGet(g1,x,y,z+1).vz;
  PSGridEmit(g2,v/7.0);
}
```

```
PSGrid3DPoint g1 = PSGrid3DPointNew(NX, NY, NZ);
PSGrid3DDouble g2 = PSGrid3DDoubleNew(NX, NY, NZ);
PSDomain3D d = PSDomain3DNew(0, NX, 0, NY, 0, NZ);

PSStencilMap(kernel,d,g1,g2);
```

実装



DSLトランスレータ　　コンパイラ

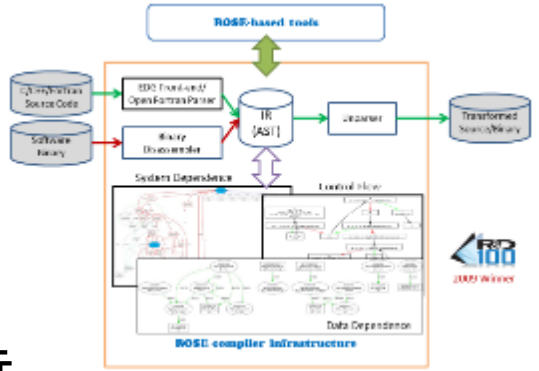
- 使い方
  - physisc-ref user-program.c

  →リファレンスターゲット（逐次C）コードを生成
  - physisc-cuda user-program.c

  → CUDAコードを生成

DSLトランスレータ

- ROSEコンパイラフレームワーク
  - http://rosecompiler.org
  - LLNLにて開発
  - C/C++のソースレベル変換
    - ソースコード上のハイレベルな情報を保持
    - LLVM/Clangも検討したが（一部実装もした）、バイトコードレベルに落とす段階で情報が欠落
  - Steep learning curveだが、まあまあ安定して動作
    - C/C++フロントエンドは商用コンパイラでも使わているEdg
  - Fortranフロントエンドもあるが、あまりサポートされてない（？）
  - けっこう綺麗なコードを生成（わりと重要）
  - CUDA, OpenCLをASTとしてサポート
    - ASTの操作としてプログラミング可能



コード生成例

Physisユーザコード

```
static void kernel_physis(const int x, const int y, const int z,
                          PSGrid3DReal  g1, PSGrid3DReal  g2,
                          REAL ce, REAL cw, REAL cn, REAL cs,
                          REAL ct, REAL cb, REAL cc) {
  int nx, ny, nz;
  nx = PSGridDim(g1, 0);
  ny = PSGridDim(g1, 1);
  nz = PSGridDim(g1, 2);

  REAL c, w, e, n, s, b, t;
  c = PSGridGet(g1, x, y, z);
  if (x == 0)    w = PSGridGet(g1, x, y, z); else w = PSGridGet(g1, x-1, y, z);
  if (x == nx-1) e = PSGridGet(g1, x, y, z); else e = PSGridGet(g1, x+1, y, z);
  if (y == 0)    n = PSGridGet(g1, x, y, z); else n = PSGridGet(g1, x, y-1, z);
  if (y == ny-1) s = PSGridGet(g1, x, y, z); else s = PSGridGet(g1, x, y+1, z);
  if (z == 0)    b = PSGridGet(g1, x, y, z); else b = PSGridGet(g1, x, y, z-1);
  if (z == nz-1) t = PSGridGet(g1, x, y, z); else t = PSGridGet(g1, x, y, z+1);
PSGridEmit(g2, cc*c + cw*w + ce*e + cs*s
             + cn*n + cb*b + ct*t);
  return;
}
```

コード生成例

$ physisc-cuda physis/examples/diffusion-benchmark/diffusion3d_physis.c

```
__device__ static void kernel_physis(const int x,const int y,const int z,__PSGrid3DFloat_dev *g1,__PS
Grid3DFloat_dev *g2,float ce,float cw,float cn,float cs,float ct,float cb,float cc)
{
  int nx;
  int ny;
  int nz;
  nx = PSGridDim(g1,0);
  ny = PSGridDim(g1,1);
  nz = PSGridDim(g1,2);
  float c;
  float w;
  float e;
  float n;
  float s;
  float b;
  float t;
  c = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,y,z)];
  if (x == 0) w = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,y,z)]; else  w = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,(x - 1),
  if (x == (nx - 1)) e = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,y,z)]; else e = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,(x
  if (y == 0) n = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,y,z)]; else n = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,(y - 1)
  if (y == (ny - 1)) s = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,y,z)]; else s = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,
  if (z == 0) b = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,y,z)]; else b = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,y,(z -
  if (z == (nz - 1)) t = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,y,z)]; else t = ((float *)(g1 -> p0))[__PSGridGetOffset3DDev(g1,x,
  ((float *)(g2 -> p0))[__PSGridGetOffset3DDev(g2,x,y,z)] = (((((((cc * c) + (cw * w)) + (ce * e)) + (cs * s)) + (cn * n)) + (cb * b)) + (ct
}
```

- コードの大まかな構成、変数名を維持
- 変換後のコードの人手による編集もわりと容易

トランスレータの動作詳細

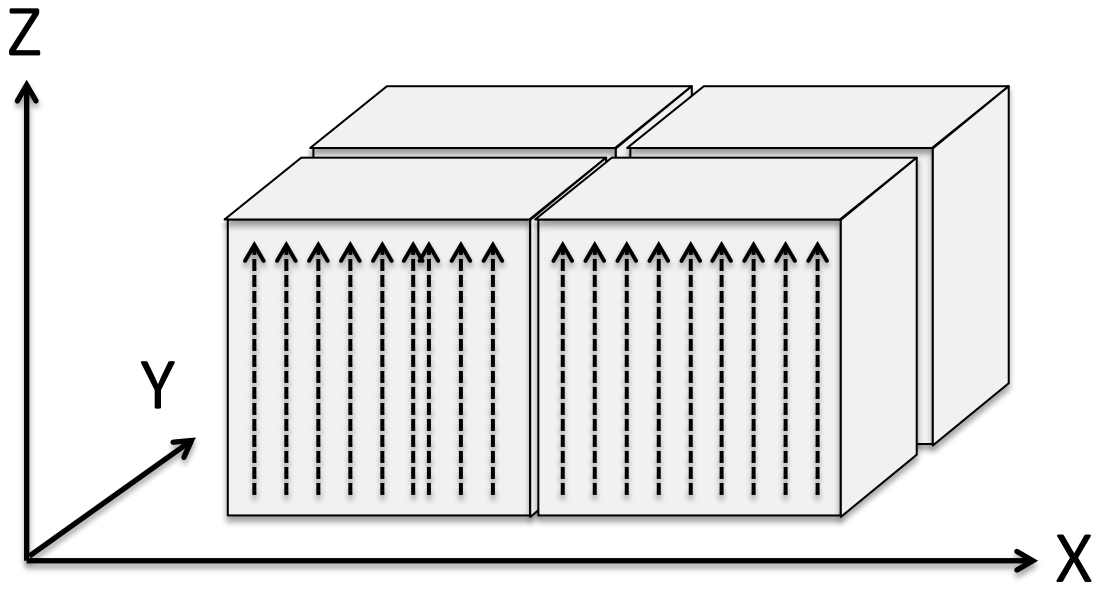
1. ユーザプログラムの解析
   1. 格子型宣言の検出
   2. ステンシル関数の検出
   3. ステンシル関数内の格子へのアクセス解析
2. コード生成
   1. 格子型の実際の定義を生成
   2. 格子オブジェクトの生成、破棄の呼び出しを適切な関数呼び出しへ変換
   3. ステンシル関数定義をターゲットアーキテクチャ向けに変換
   4. ステンシル関数呼び出しコードの生成
3. 最適化

2. コード生成

- ターゲット毎のコード生成をC++のクラスとして実装
  - ReferenceTranslator
    - コード変換の検証用リファレンスコード生成
  - CUDATranslater
    - ReferenceTranslatorを継承し、CUDA固有部分を実装
  - MPITranslator
  - MPICUDATranslator
- ユーザー定義格子型
  - CPU向け→AoSとして定義
  - GPU向け→SoAとして定義
  - ユーザコードのPSGridGetを適宜変換
- ステンシル関数呼び出し
  - CUDA、MPIターゲットの場合は領域分割により並列化

CUDA Thread Blocking

- Each thread sweeps points in the Z dimension
- X and Y dimensions are blocked with AxB thread blocks, where A and B are user-configurable parameters (64x4 by default)



Example: 7-point Stencil GPU Code

```
__device__ void kernel(const int x,const int y,const int z,__PSGrid3DFloatDev *g,
                                                  __PSGrid3DFloatDev *g2)
{
  float v = ((((( *__PSGridGetAddrFloat3D(g,x,y,z) +
              *__PSGridGetAddrFloat3D(g,(x + 1),y,z)) +
              *__PSGridGetAddrFloat3D(g,(x - 1),y,z)) +
              *__PSGridGetAddrFloat3D(g,x,(y + 1),z)) +
             *__PSGridGetAddrFloat3D(g,x,(y - 1),z)) +
              *__PSGridGetAddrFloat3D(g,x,y,(z - 1))) +
               *__PSGridGetAddrFloat3D(g,x,y,(z + 1)));
   *__PSGridEmitAddrFloat3D(g2,x,y,z) = v;
}

__global__ void __PSStencilRun_kernel(int offset0,int offset1,__PSDomain dom,
                                      __PSGrid3DFloatDev g,__PSGrid3DFloatDev g2)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x + offset0;
  int y = blockIdx.y * blockDim.y + threadIdx.y + offset1;
  if (x < dom.local_min[0] || x >= dom.local_max[0] ||
       (y < dom.local_min[1] || y >= dom.local_max[1]))
    return ;
  int z;
  for (z = dom.local_min[2]; z < dom.local_max[2]; ++z) {
    kernel(x,y,z,&g,&g2);
  }
}
```

3. 最適化

- ステンシル適用ループに対するピープホール最適化
- トランスレータコマンドのオプションとして最適化有効･無効設定ファイルを指示可

```
OPT_OFFSET_COMP = true | false
OPT_LOOP_PEELING = true | false
OPT_REGISTER_BLOCKING = true | false
OPT_LOOP_OPT = true | false
OPT_UNCONDITIONAL_GET = true | false
```

最適化： レジスタブロッキング

- スレッドローカルな局所性の活用
  - ローカル変数の使い回しによるグローバルメモリアクセス削減

適用前

```
for (int k = 1; k < n-1; ++k) {
  g[i][j][k] = a*(f[i][j][k]+f[i][j][k-
1]+f[i][j][k+1]);
}
```

適用後

```
double kc = f[i][j][0];
doubke kn = f[i][j][1];
for (int k = 1; k < n-1; ++k) {
  double kp = kc;
  kc = kn;
  kn = f[i][j][k+1];
  g[i][j][k] = a*(kc+kp+kn);
}
```

最適化： インデックス計算最適化

- インデックス計算の共通部分式括りだし
  - イテレーション内の共通項
  - イテレーション間の共通項

適用前

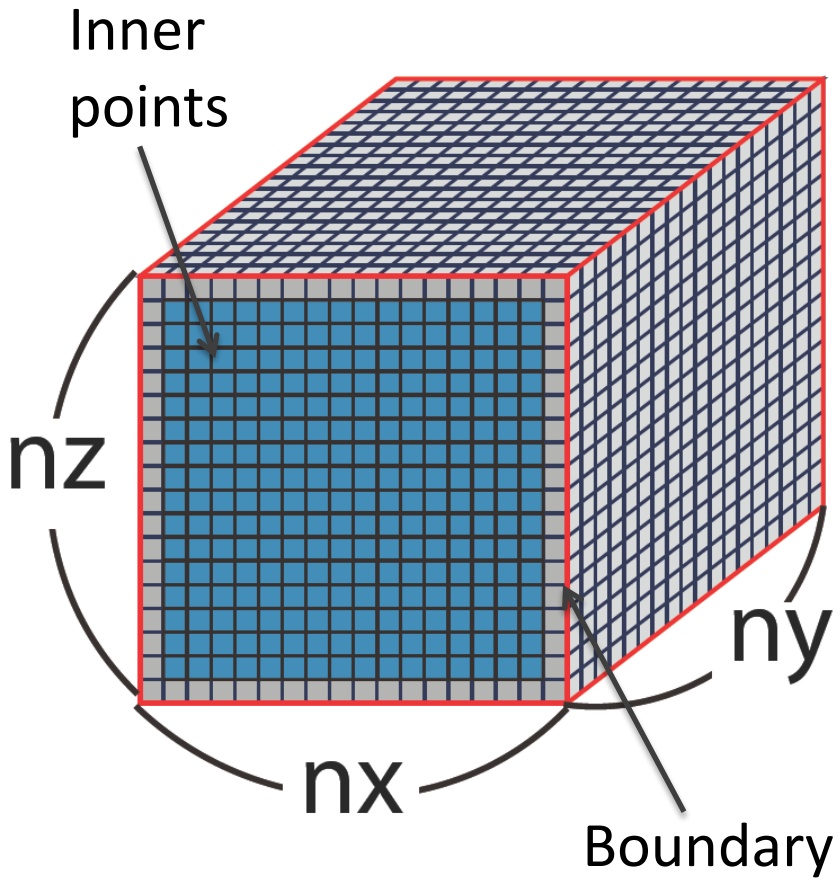
```
for (int k = 1; k < n-1; ++k) {
  g[i][j][k] = a*(f[i+j*n+k*n*n]+
      f[i+j*n+(k-1)*n*n]+f[i+j*n+(k+1)*n*n]);
}
```

適用後

```
int idx = i+j*n+n*n;
for (int k = 1; k < n-1; ++k) {
 g[i][j][k] = a*(f[idx]+f[idx-n*n]+f[idx+n*n]);
 idx += n*n;
}
```

その他ループピーリング等のループ最適化器

Optimization: Overlapped Computation and Communication



1. Copy boundaries from GPU to CPU for non-unit stride cases

2. Computes interior points

3. Boundary exchanges with neighbors

4. Computes boundaries

Time

Optimization Example: 7-Point Stencil CPU Code

```
for (i = 0; i < ...; i++) {
  __PSStencilRun_kernel_interior<<<s0_grid_dim,block_dim,0, stream_interior>>>
      (__PSGetLocalOffset(0),__PSGetLocalOffset(1),__PSDomainShrink(&s0 -> dom,1),
      *((__PSGrid3DFloatDev *)(__PSGridGetDev(s0 -> g))), *((__PSGrid3DFloatDev *)(__PSGrid
  int fw_width[3] = {1L, 1L, 1L};
  int bw_width[3] = {1L, 1L, 1L};
  __PSLoadNeighbor(s0 -> g,fw_width,bw_width,0,i > 0,1);
  __PSStencilRun_kernel_boundary_1_bw<<<1,(dim3(1,128,4)),0,
      stream_boundary_kernel[0]>>>(__PSDomainGetBoundary(&s0 -> dom,0,0,1,5,0),
          *((__PSGrid3DFloatDev *)(__PSGridGetDev(s0 -> g))), *((__PSGrid3DFloatDev *)(__PS
  __PSStencilRun_kernel_boundary_1_bw<<<1,(dim3(1,128,4)),0,
      stream_boundary_kernel[1]>>>(__PSDomainGetBoundary(&s0 -> dom,0,0,1,5,1),
          *((__PSGrid3DFloatDev *)(__PSGridGetDev(s0 -> g))), *((__PSGrid3DFloatDev *)(__PS
  ...
  __PSStencilRun_kernel_boundary_2_fw<<<1,(dim3(128,1,4)),0,
      stream_boundary_kernel[11]>>>(__PSDomainGetBoundary(&s0 -> dom,1,1,1,1,0),
          *((__PSGrid3DFloatDev *)(__PSGridGetDev(s0 -> g))), *((__PSGrid3DFloatDev *)(__
  cudaThreadSynchronize();
}
cudaThreadSynchronize();
}
```

Computing Interior Points

Boundary Exchange

Computing Boundary Planes Concurrently

評価

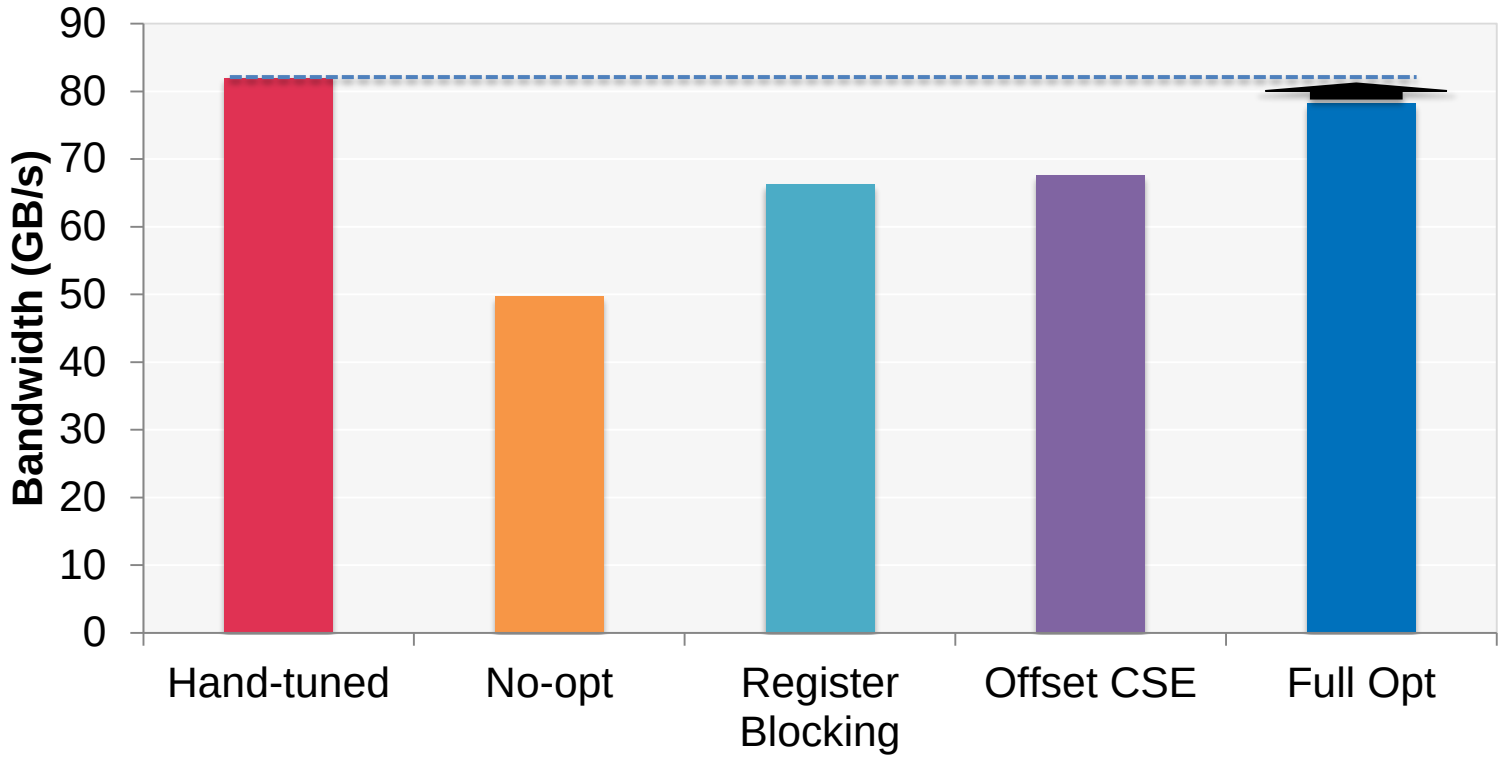
- 小規模ベンチマークステンシルによる基礎評価
- 実アプリケーションによる応用評価

基礎評価

- Performance and productivity
- Sample code
  - 7点diffusion
  - 姫野ベンチマーク
- Platform
  - Tsubame 2.0
    - Node: Westmere-EP 2.9GHz x 2 + M2050 x 3
    - Dual Infiniband QDR with full bisection BW fat tree

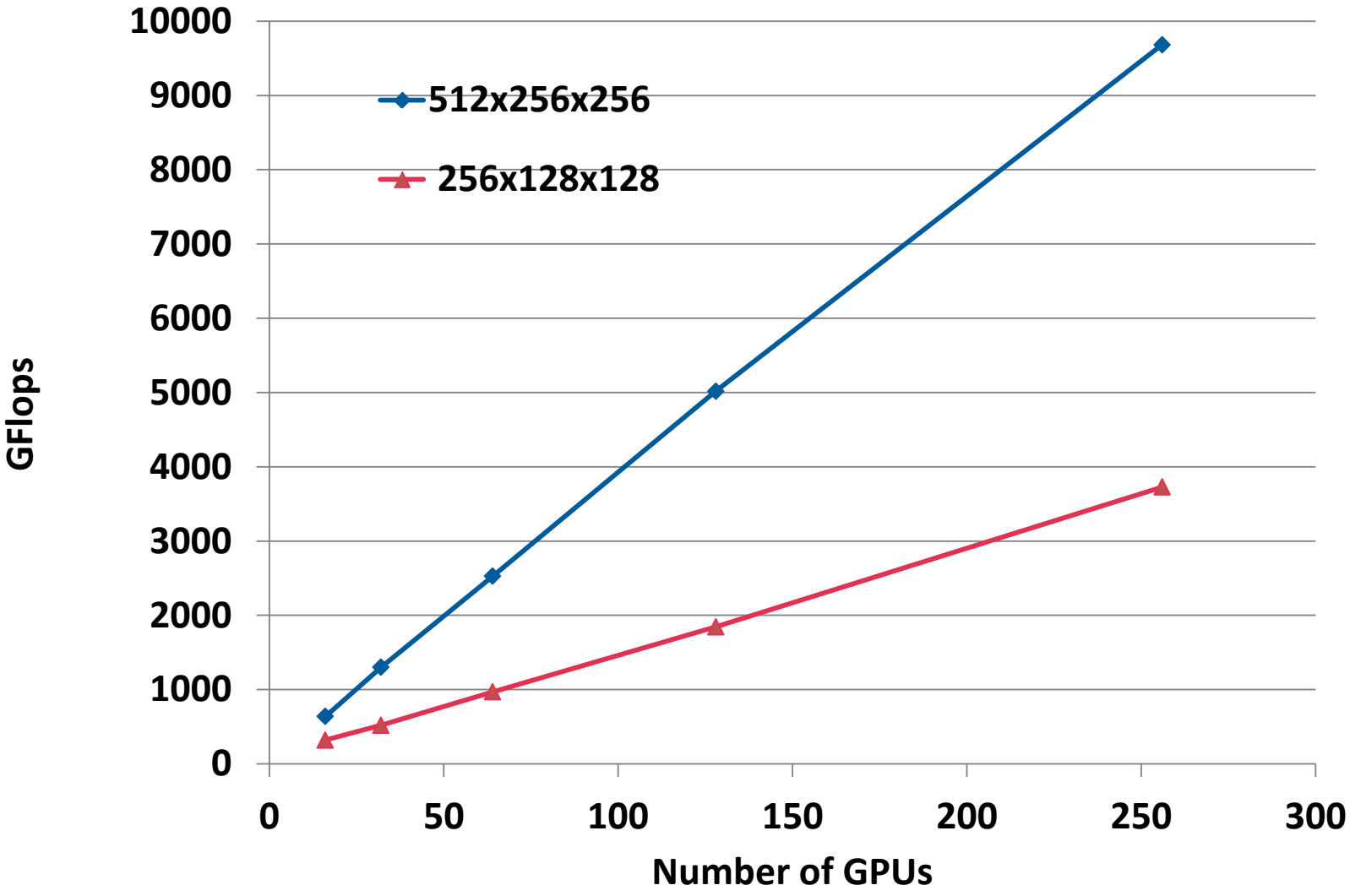
7点diffusion

1 GPU (Tesla M2050), CUDA 4.1



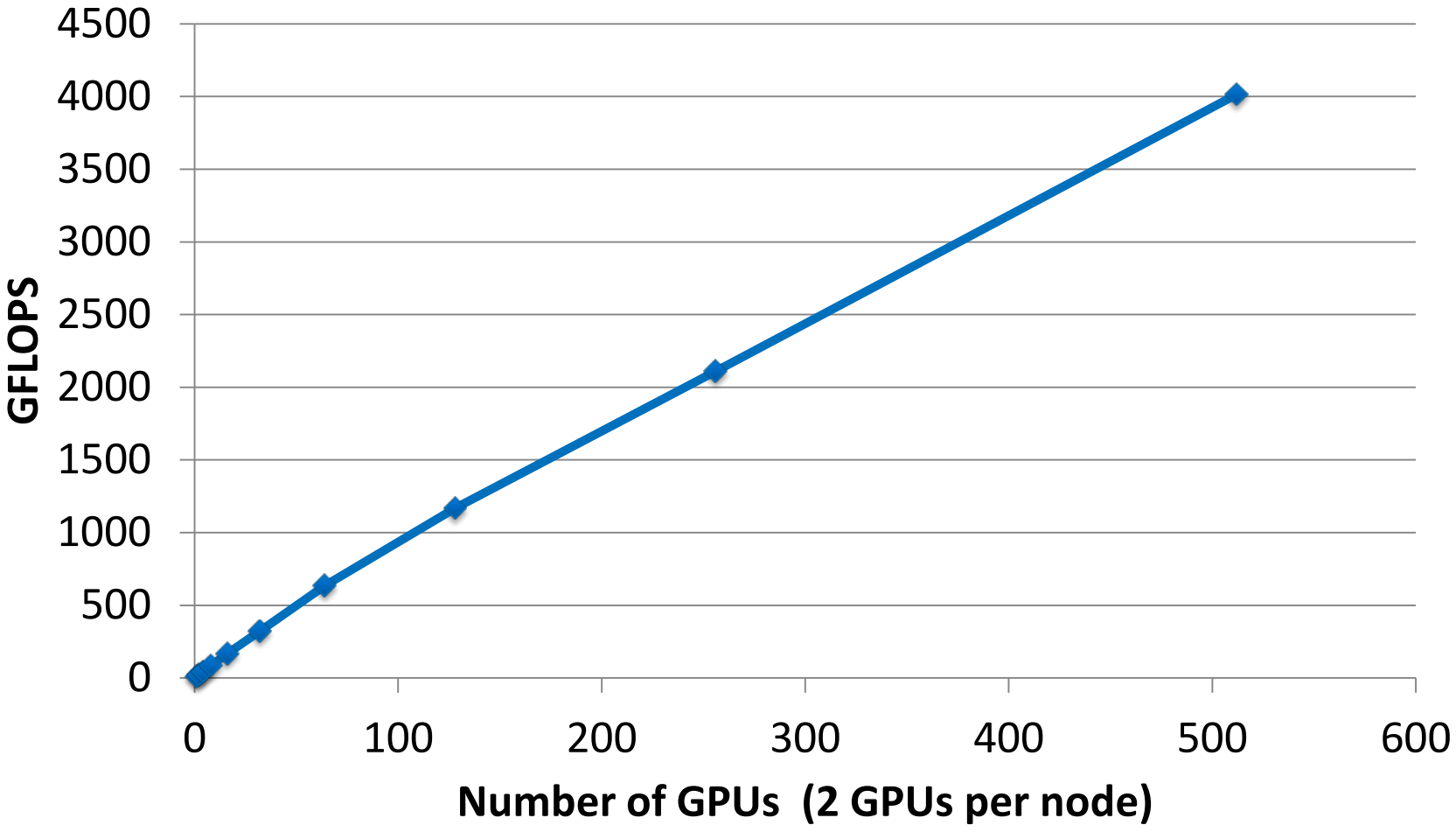
コード自動生成により、人手による最適化コードの95%の性能を達成

Diffusion Weak Scaling



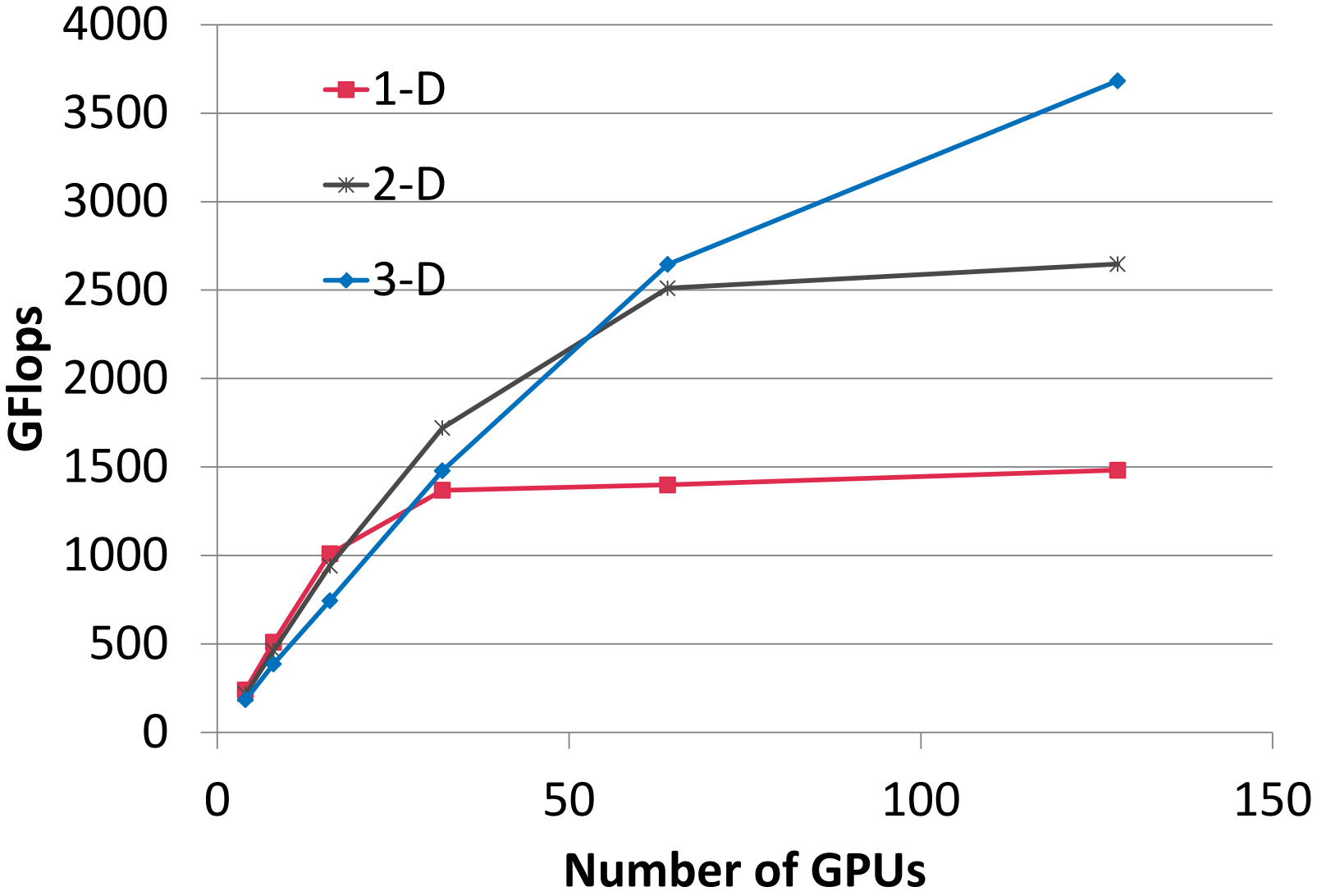
Seismic Weak Scaling

Problem size: 256x256x256 per GPU



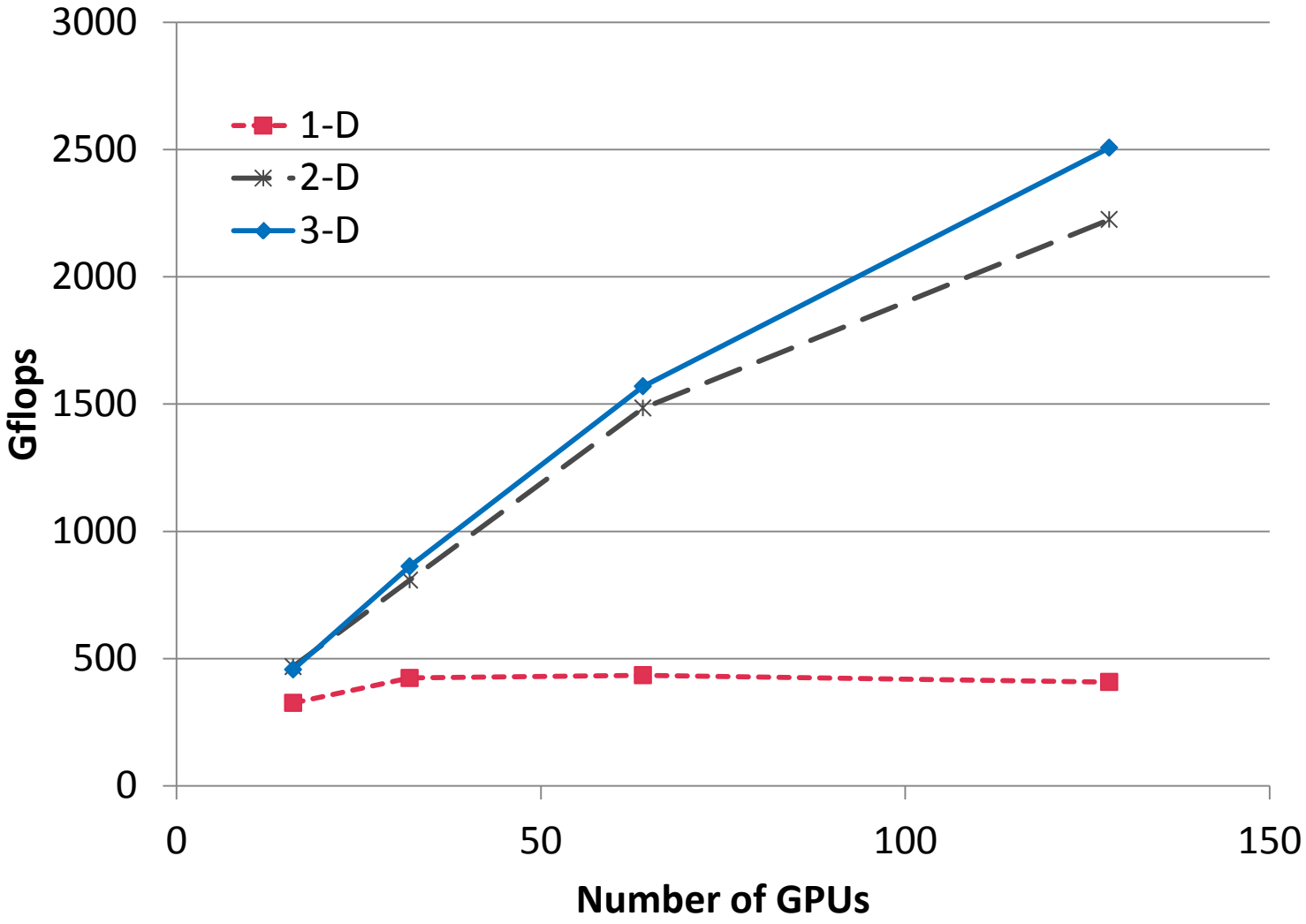
Diffusion Strong Scaling

Problem size: 512x512x4096



Himeno Strong Scaling

Problem size XL (1024x1024x512)



応用評価

- 格子ボルツマン法
- 気象シミュレーション
- 熱流体ソルバー

格子ボルツマン法による大規模都市気流計算

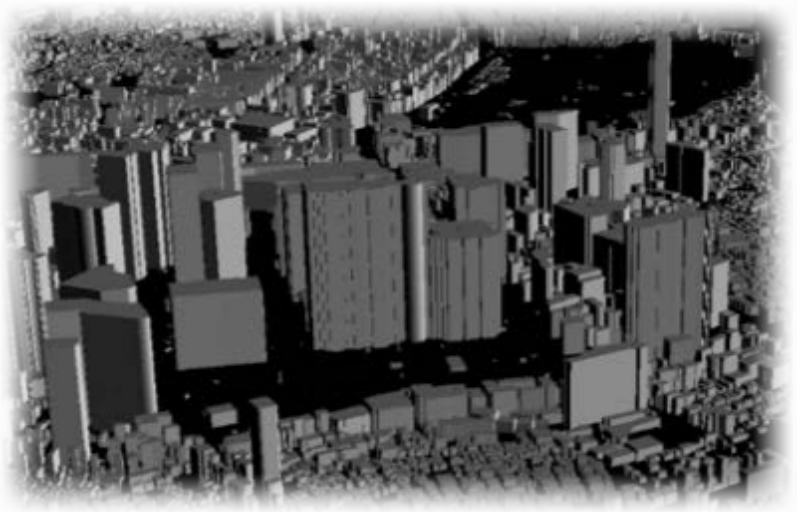
小野寺、青木（東工大）ら [HPCS'13]

格子ボルツマン法の利点

- 単純なアルゴリズム
- 圧力Poisson方程式のような行列の反復計算が不要なため、大規模計算でも高効率に計算が可能
- 複雑物体も容易に取り扱う事ができる

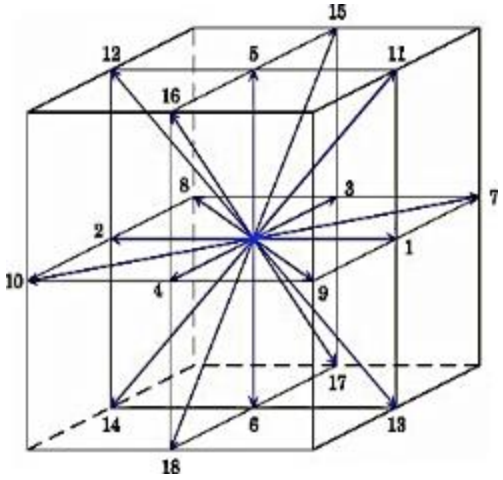
コヒーレント構造Smagorinskyモデルの導入

都市の建物データ



新宿区・渋谷区・目黒区・千代田区・中央区・港区・江東区を含む10km×10km四方のエリア

D3Q19 discrete velocity



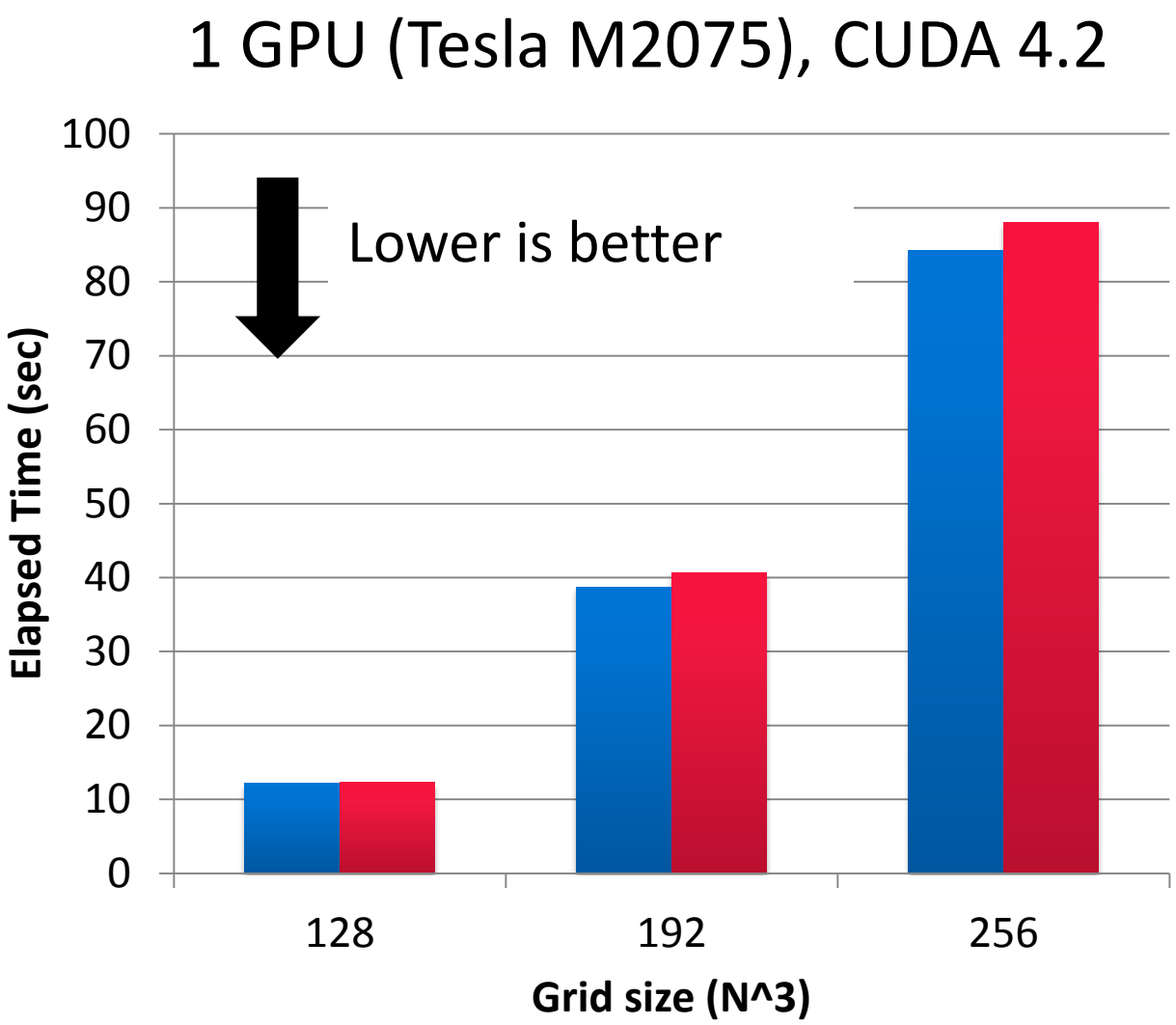
Physisによる記述

- 格子点に定義する物理量の定義
- オリジナルデータ構造を踏襲
- 各点に対する操作をステンシル関数として記述
  - オリジナルのCUDAカーネルコードから移植

```
struct VariablesP {
  FLOAT f_n[NUM_DIRECTION]; // NUM_DIRECTION=19
};
DeclareGrid3D(Variables, struct VariablesP);
struct BasisVariablesP {
  FLOAT r_n; FLOAT u_n; FLOAT v_n; FLOAT w_n;
};
DeclareGrid3D(BasisVariables, struct BasisVariablesP);
struct FluidPropertyP {
  FLOAT vis;
};
DeclareGrid3D(FluidProperty, struct FluidPropertyP);
struct StressP {
  FLOAT vis_sgs; FLOAT Fcs_sgs;  FLOAT Div;
  FLOAT SS; FLOAT WW;
};
DeclareGrid3D(Stress, struct StressP);

PSGrid3DVariables cq_p;
PSGrid3DVariables cqn_p;
PSGrid3DBasisVariables cbq_p;
PSGrid3DFluidProperty cfp_p;
PSGrid3DStress str_p;
```

性能比較



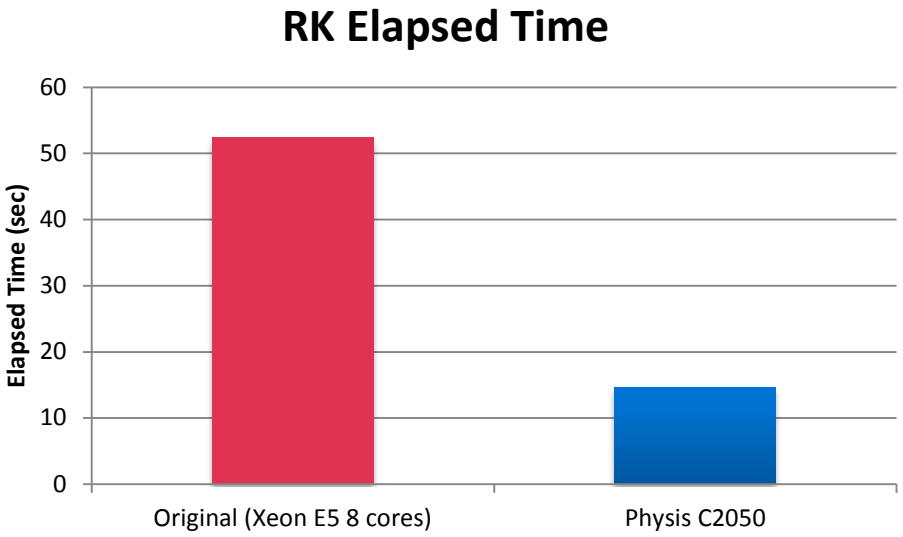
- ほぼ人手によるコードと同様の性能
- Physis版はGPU向けコードだけでなくCPU向けコードも生成可能→可搬性

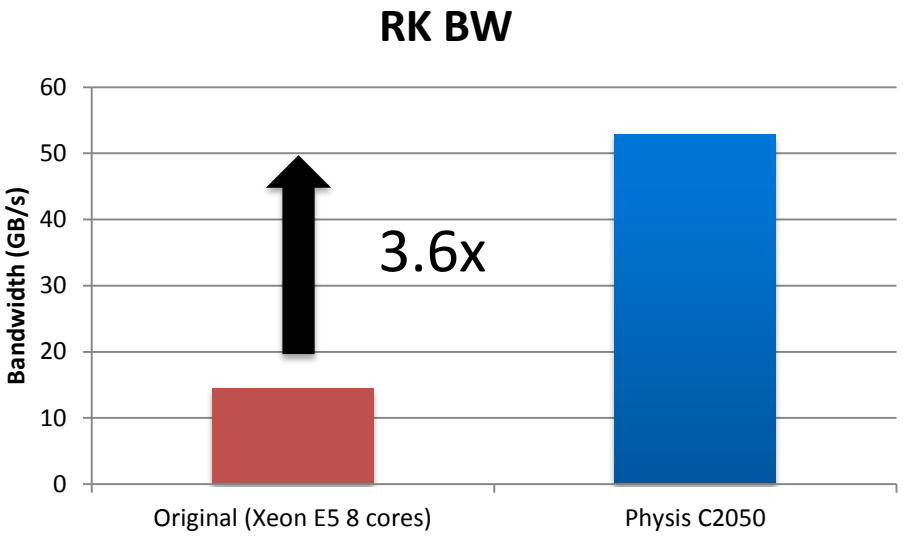
気象シミュレーション

- SCALE-LES： 3次元直交格子有限体積法による気候シミュレーションコード
- 手始めに力学過程の主要サブルーチンのPhysis版を実装
  - ルンゲクッタ法
  - 多数（25）の小規模な3重ループから構成
- コードサイズ比較
  - オリジナル行数（F90） 520行
  - Physis行数 244行

性能比較

- CPU: Xeon E5 8 cores
  - BW: ~30 GB/s
- GPU: Tesla C2050
  - BW: ~100 GB/s
- バンド幅比率（およそ
  倍）に近い性能差
  - アプリB/F: 2.1





SCALE-LES今後の計画

- Physisへの完全移植
  - 力学過程はルーチン残り2つ
  - 物理過程
    - スカラー計算なためメモリアクセスは単純
    - 分岐が多いためSIMD・ベクトル化が難しい
- SCALE−LESの最適化
  - 小規模ステンシルカーネル25本
  - カーネル融合、分割
  - 単純な総当り自動チューニングではケース数の爆発的な増加
  - 性能モデルによる絞り込み＋自動チューニング
- 複数ノード・複数GPUでの評価
  1. ノード並列: オリジナルMPI + ノード内並列 : Physis
  2. ノード間・ノード内並列 : Physis
- PhysisのCPUバックエンドを使った他アーキテクチャとの性能可搬性の実証
  - 京、Intelマシンなど

最適化

- 格子点アクセスの局所性
  - ステンシルカーネル間で共通に読み込む格子
  - カーネル間でread-after-writeな依存関係を持つ格子
- ステンシルカーネル（ループ）の融合によりDRAMアクセス回数削減可能
- 通常のCUDAプログラミンでは人手によりカーネル関数の融合が必要
- Physis DSLトランスレータによる自動化

熱流体ソルバー

- FFVCミニアプリ
  - 東大生産技術研究所で開発中の三次元非定常非圧縮熱流体ソルバーFFVCの簡易版
  - 直交等間隔格子、有限体積法、Fractional Step法
- ひとまず・・・主要カーネルのポアソンソルバーステンシルをPhysisで記述
  - 全体の一部に過ぎないためデータコピーオーバーヘッド有り
  - 効率上はアプリケーション全体のデータモデルをフレームワーク上に移植するべき

オリジナルFortranループ

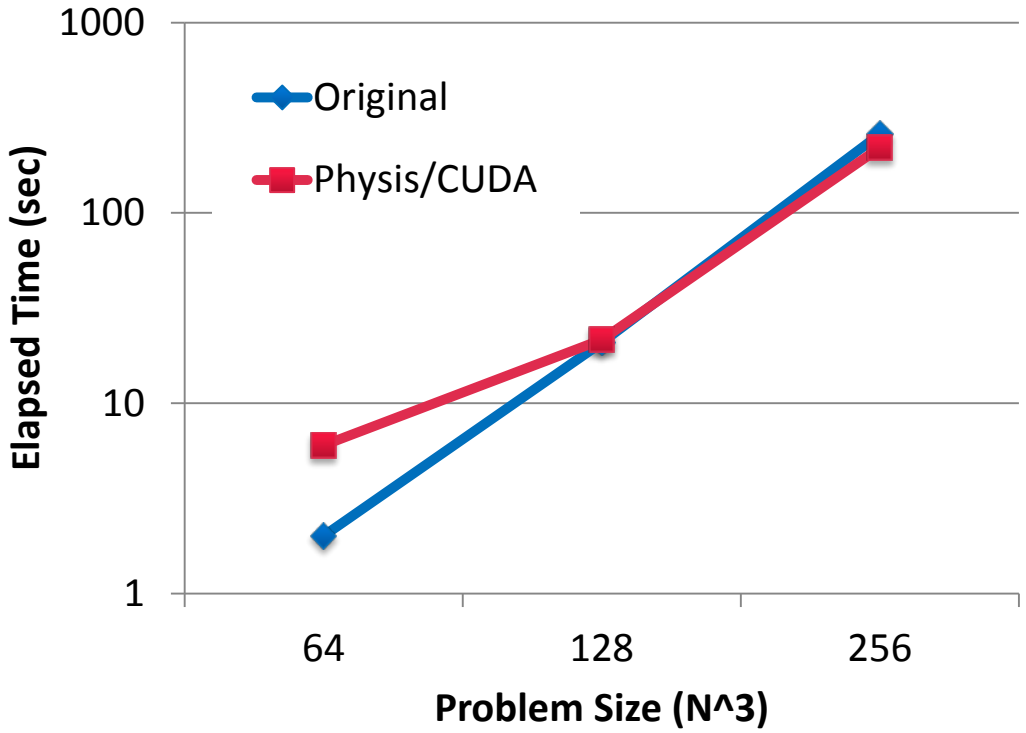
```
do k=1,kx
  do j=1,jx
  do i=1+mod(k+j+color+ip,2), ix, 2
    idx = bp(i,j,k)
    ndag_e = real(ibits(idx, bc_ndag_E, 1))  ! e, non-diagonal
    ndag_w = real(ibits(idx, bc_ndag_W, 1))  ! w
    ndag_n = real(ibits(idx, bc_ndag_N, 1))  ! n
    ndag_s = real(ibits(idx, bc_ndag_S, 1))  ! s
    ndag_t = real(ibits(idx, bc_ndag_T, 1))  ! t
    ndag_b = real(ibits(idx, bc_ndag_B, 1))  ! b
    dd = 1.0 / real(ibits(idx, bc_diag, 3))  ! diagonal
    pp = p(i,j,k)
    ss = ndag_e * p(i+1,j  ,k  )  + ndag_w * p(i-1,j  ,k  ) &
      + ndag_n * p(i  ,j+1,k  ) + ndag_s * p(i  ,j-1,k  ) &
      + ndag_t * p(i  ,j  ,k+1) + ndag_b * p(i  ,j  ,k-1)
    dp = ( dd*ss + b(i,j,k) - pp ) * omg
    p(i,j,k) = pp + dp
    res = res + dble(dp*dp) * dble(ibits(idx, Active, 1))
  end do
  end do
end do
```

Physisステンシル関数

```
static void sor2sma_kernel(const int i, const int j, const int k,
                           PSGrid3DReal p, REAL_TYPE omg, PSGrid3DDouble res,
                           PSGrid3DReal b, PSGrid3DInt bp) {
  int idx = PSGridGet(bp, i, j, k);
  REAL_TYPE ndag_e = real(ibits(idx, BC_NDAG_E, 1)); //  ! e, non-diagonal
  REAL_TYPE ndag_w = real(ibits(idx, BC_NDAG_W, 1)); //  ! w
  REAL_TYPE ndag_n = real(ibits(idx, BC_NDAG_N, 1)); //  ! n
  REAL_TYPE ndag_s = real(ibits(idx, BC_NDAG_S, 1)); //  ! s
  REAL_TYPE ndag_t = real(ibits(idx, BC_NDAG_T, 1)); //  ! t
  REAL_TYPE ndag_b = real(ibits(idx, BC_NDAG_B, 1)); //  ! b
  REAL_TYPE dd = 1.0 / real(ibits(idx, BC_DIAG, 3)); //  ! diagonal
  REAL_TYPE pp = PSGridGet(p, i, j, k);
  REAL_TYPE ss = ndag_e * PSGridGet(p, i+1, j  ,k  ) + ndag_w * PSGridGet(p, i-1,j  ,k  )
     + ndag_n * PSGridGet(p, i  ,j+1,k  ) + ndag_s * PSGridGet(p, i  ,j-1,k  )
     + ndag_t * PSGridGet(p, i  ,j  ,k+1) + ndag_b * PSGridGet(p, i  ,j  ,k-1);
  REAL_TYPE dp = ( dd*ss + PSGridGet(b, i, j, k) - pp ) * omg;
  PSGridEmit(p, pp + dp);
  PSGridEmit(res, dble(dp*dp) * dble(ibits(idx, ACTIVE_BIT, 1)));
}
```

性能結果

- Tesla M2075 vs. Intel SNB Xeon E5-2670
  - M2075 BW: 90 GB/s > Xeon: 35 GB/s
- ポアソンソルバーカーネルのみ



今後の計画

- 密な計算＋疎な集合 への拡張
  - Structured Adaptive Mesh Refinement
  - Fast Multipole Method
  - など
- 課題
  - 生産性
    - 分散メモリ環境におけるAMR等の操作
  - 性能
    - 負荷分散、通信コスト
- 方針：高性能密計算コード＋動的負荷分散ランタイム
  - 密な計算：　DSLによりコード生成
    - ノード内並列→アーキテクチャ・アプリケーション依存
  - 疎な集合：　スケーラブルなランタイム/ライブラリによる並列化
    - ノード間並列→ある程度の種類のパターン＆標準API（MPI）

おわりに

- アプリケーションフレームワーク
  - アプリケーションドメインのボキャブラリによってプログラムを記述 → 生産性の向上
  - ドメインの知識を用いた最適化 → 高性能の実現
- 例： Physis: ステンシル計算向けフレームワーク
    - http://github.com/naoyam/physis

- 謝辞
  - JST CREST「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」