

CUDA vs OpenACC : マイクロベンチマークとアプリケーションによるOpenACCコンパイラの評価

丸山直也

理研AICSプログラム構成モデル研究チーム

nmaruyama@riken.jp

2012/12/25 @ 自動チューニングシンポジウム

※本発表内容は星野哲也(東工大)との共同研究です

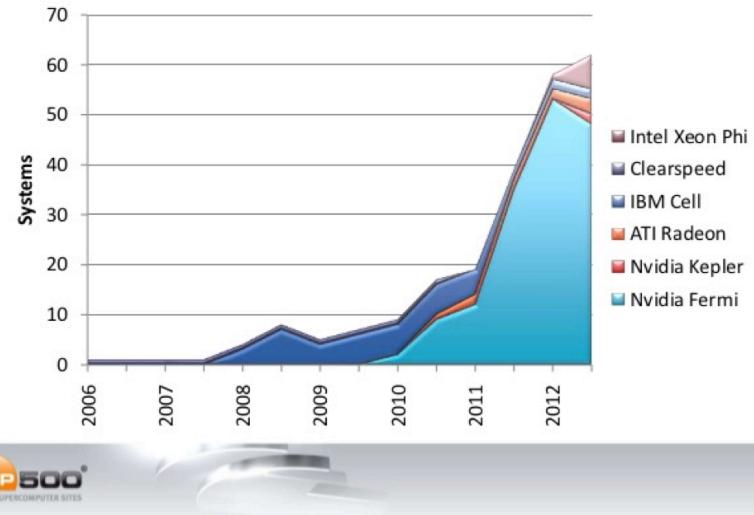
背景

- アクセラレータを搭載したシステムの台頭
 - ex. Titan, Tianhe, TSUBAME2.0
 - CPU向けに作られたレガシーアプリの移植が課題



TITAN 1st in top500 (Nov. 2012)
18688 NVIDIA K20 GPUs

Accelerators/Co-Processors



available at: <http://www.top500.org>

TSUBAME2.0, 4th in top500 (Nov. 2010)
4258 NVIDIA Fermi GPUs



GPUプログラミングインターフェース

- CUDA
 - 現在最も広く使われている
 - 低レベルな記述を必要とする
- OpenACC
 - ディレクティブベース
 - C言語とFortranに対応
 - 多くのアクセラレータに対応(予定)
 - GPU環境への移植の簡素化に期待

```
C
#pragma acc directive [clause]
{
    // C code
}

Fortran
 !$acc directive [clause]
    ! Fortran code
 !$acc end directive
```

OpenACCの指示文シンタックス



OpenACCによるGPU環境への移植

- GPUへの移植において一般に必要とされること

1. GPUへのオフロード箇所の特定
2. GPUコードの記述
3. CPU・GPUの分散メモリ空間の管理

- OpenACCの3つの主要指示文

1. オフロード箇所の指定
 - **parallel, kernels** によりオフロード箇所の指定, GPUでの実行コードを生成
2. データ移動指示
 - **data** ディレクティブによりメモリ空間の半自動的に管理
3. パラメータなどの指示
 - **loop** ディレクティブによりスレッドブロックのサイズなどを指定

```
!$acc data copy(c), copyin(a, b)  
 !$acc kernels
```

```
 !$acc loop gang vector(16)
```

```
 do j = 1, n
```

```
 !$acc loop gang vector(16)
```

```
 do i = 1, n
```

```
 cc = 0
```

```
 do k = 1, n
```

```
 cc = cc + a(i,k) * b(k,j)
```

```
 end do
```

```
 c(i,j) = cc
```

```
 end do
```

```
 end do
```

```
 !$acc end kernels
```

行列積 (PGI コンパイラの場合)

目的と評価方法

- 目的
 - 新しいアクセラレータ向けプログラミングモデルであるOpenACCコンパイラの性能を理解すること
- 評価方法
 - CUDA・OpenACC を用いていくつかのアプリケーションをフェアに移植・最適化し、性能を比較することで評価する
 - カーネルベンチマーク
 - 行列積 (compute intensive kernel)
 - 7点ステンシル (memory intensive kernel)
 - 実アプリケーション
 - JAXAにより研究開発されている流体アプリケーションUPACS

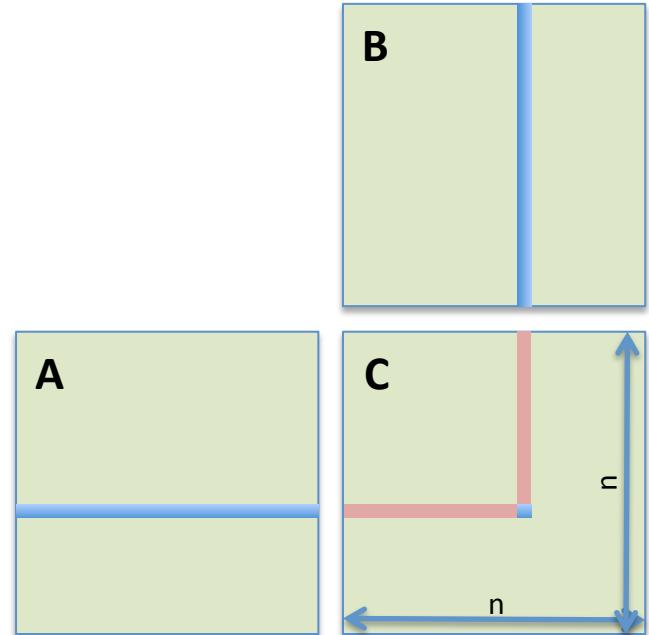
もうCUDA/OpenCLは必要ない？

- 指示文に基づいたプログラミング → CUDAなどよりよっぽど移植が簡単
- 性能？

カーネルベンチマーク

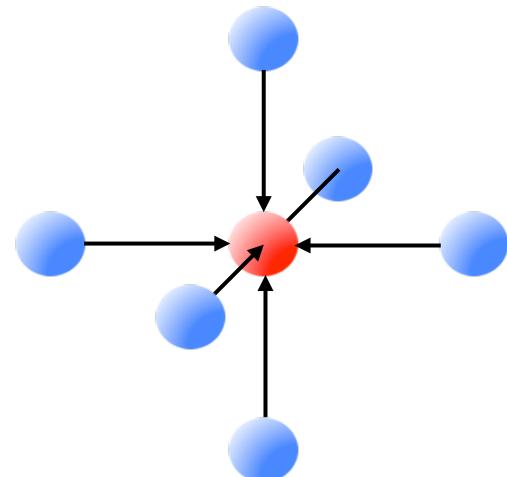
- 行列積

- $C = A \times B$
- A, B, C は $n \times n$ 行列, 倍精度
- コンピュートバウンド



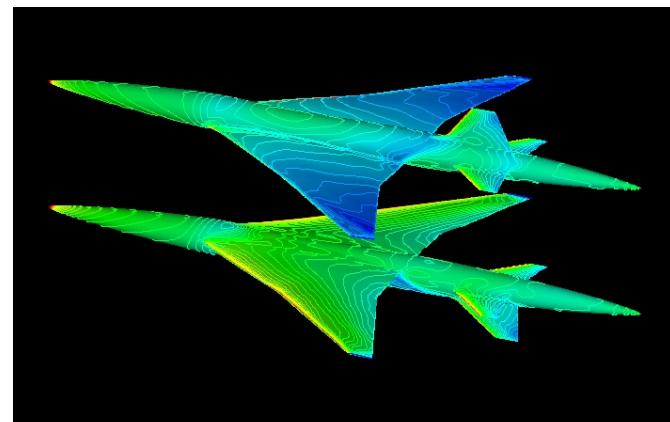
- 7点ステンシル

- 流体計算の差分計算などに用いられる
- 7点の隣接セルからセルの値を更新する
- 3-D配列, 単精度
- メモリバウンド



実アプリベンチマーク： UPACS

- UPACS
 - 宇宙航空研究開発機構により研究・開発
 - 航空宇宙分野の様々な流体現象の解析を目的とし、多くのCFDソルバを提供している
 - 本研究ではNavier-Stokes 方程式を解くソルバを選択
- プログラム概要
 - マルチブロック構造格子法による並列化
 - MPI によるフラット並列、コンパイラの自動並列化によるノード内並列
 - およそ10万行のFortran 90 からなる
 - 3つの主要フェーズからなる
 - Convection, Viscosity, Time Integration
 - 実行時間のおよそ90%を占める

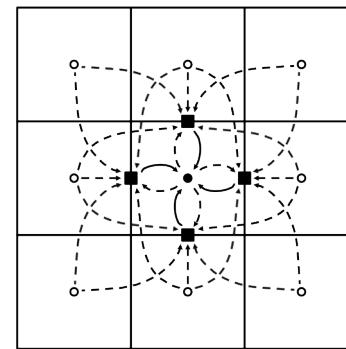
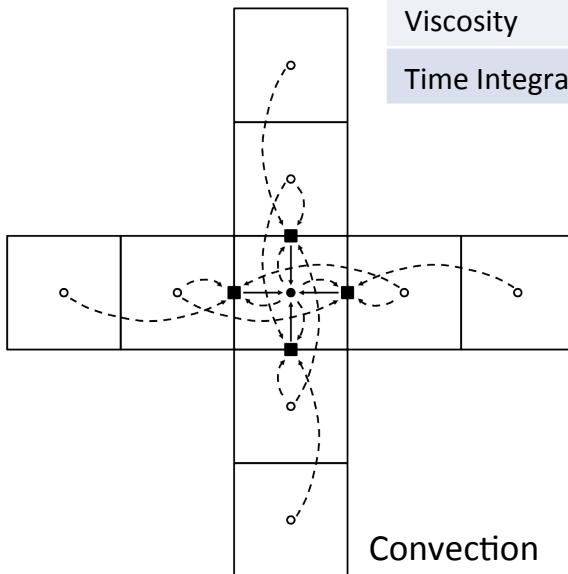


実アプリケーション

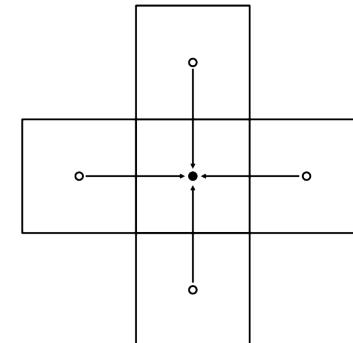
- 主要3フェーズ

- 各フェーズは3次元のステンシル計算を行う
- Convection, Viscosity ではループ伝搬依存はない(陽解法)
- Time Integration は各セルの更新に依存があり、ウェーブフロントの並列処理が必要(陰解法)

Phase	LoC	# of subroutines	# of loop nests	# of 3-D data	Read	Upload	Temporary	% of Total time (1CPU core)
Convection	682	10	7	29	5	15		25.0%
Viscosity	599	5	5	33	5	25		37.7%
Time Integration	622	5	5	20	5	5		28.5%



Viscosity



Time Integration

最適化 : 行列積

- Baseline
 - kernels ディレクティブと、2つのloop ディレクティブを用いて実装
 - data ディレクティブを用いて必要な配列はGPUのメモリ上に転送済み
 - CUDAでは各スレッドがそれぞれ内積を計算。スレッドブロックは(16, 16)
- Thread Mapping
- Cache blocking
- Loop unrolling
- Shared Memory Blocking

```
!$acc kernels present(a, b, c)
!$acc loop
do j = 1, n
!$acc loop
do i = 1, n
cc = 0
do k = 1, n
cc = cc + a(i,k) * b(k,j)
end do
c(i,j) = cc
end do
end do
!$acc end kernels
```

OpenACC 行列積

最適化 : 行列積

- Baseline
- Thread Mapping
 - CUDAのスレッドブロックサイズにあたる, gang, worker, vector を明示的に指定する
 - 指定する各パラメータは、いくつか実行した中で最速のものを選ぶ
- Cache blocking
- Loop unrolling
- Shared Memory Blocking

```
!$acc kernels present(a, b, c)
!$acc loop gang vector(THREAD_Y)
    do j = 1, n
!$acc loop gang vector(THREAD_X)
    do i = 1, n
        cc = 0
        do k = 1, n
            cc = cc + a(i,k) * b(k,j)
        end do
        c(i,j) = cc
    end do
end do
!$acc end kernels
```

OpenACC 行列積 (PGI バージョン)

最適化 : 行列積

- Baseline
- Thread Mapping
- Cache blocking
 - 各スレッドが行列Cの 4×4 の部分行列を計算
 - それぞれのSMが16倍大きなエリアを計算することになるため、ローカリティが高まる
- Loop unrolling
- Shared Memory Blocking

```
!$acc kernels present(a, b, c)
!$acc loop gang vector(THREAD_Y)
do j = 1, n, 4
!$acc loop gang vector(THREAD_X)
do i = 1, n, 4
    c00 = 0; c01 = 0; c02 = 0; c03 = 0
    c10 = 0; c11 = 0; c12 = 0; c13 = 0
    c20 = 0; c21 = 0; c22 = 0; c23 = 0
    c30 = 0; c31 = 0; c32 = 0; c33 = 0
    do k = 1, n
        c00 = c00 + a(i ,k) * b(k,j )
        c10 = c10 + a(i+1,k) * b(k,j )
        c20 = c20 + a(i+2,k) * b(k,j )
        c30 = c30 + a(i+3,k) * b(k,j )
        c01 = c01 + a(i ,k) * b(k,j+1)
        ...
        c33 = c33 + a(i+3,k) * b(k,j+3)
    end do
    c(i ,j ) = c00
    c(i+1,j ) = c10
    ...
    c(i+2,j+3) = c23
    c(i+3,j+3) = c33
end do
end do
!$acc end kernels
```

最適化 : 行列積

- Baseline
- Thread Mapping
- Cache blocking
- Loop unrolling
 - 最内のkループに手動で16段のループアンローリングを行う
- Shared Memory Blocking

```
!$acc kernels present(a, b, c)
!$acc loop gang vector(THREAD_Y)
do j = 1, n, 4
!$acc loop gang vector(THREAD_X)
do i = 1, n, 4
    c00 = 0; c01 = 0; c02 = 0; c03 = 0
    c10 = 0; c11 = 0; c12 = 0; c13 = 0
    c20 = 0; c21 = 0; c22 = 0; c23 = 0
    c30 = 0; c31 = 0; c32 = 0; c33 = 0
do k = 1, n, 16
    c00 = c00 + a(i ,k) * b(k,j )
    ...
    c33 = c33 + a(i+3,k) * b(k,j+3)
    c00 = c00 + a(i ,k+1) * b(k+1,j )
    ...
    c33 = c33 + a(i+3,k+1) * b(k+1,j+3)
    ...
    c00 = c00 + a(i ,k+15) * b(k+15,j )
    ...
    c33 = c33 + a(i+3,k+15) * b(k+15,j+3)
end do
c(i ,j ) = c00
c(i+1,j ) = c10
...
c(i+2,j+3) = c23
c(i+3,j+3) = c33
end do
end do
!$acc end kernels
```

最適化 : 行列積

- Baseline
- Thread Mapping
- Cache blocking
- Loop unrolling
- Shared Memory Blocking
 - 同一スレッドブロック内のスレッドが、シェアードメモリを用いてデータを共有し内積を計算
 - OpenACCでは、仕様上CUDAの**syncthreads**にあたる同期の指示文がないため、CUDA版のみに適用

```
real(8), shared :: Asub(17,16), Bsub(17,64)
real(8) :: Cij1, Cij2, Cij3, Cij4

tx = threadIdx%x
ty = threadIdx%y
i = (blockIdx%x-1) * 16 + threadIdx%x
j = (blockIdx%y-1) * 16 * 4 + threadIdx%y

Cij1 = 0.0; Cij2 = 0.0; Cij3 = 0.0; Cij4 = 0.0
do kb = 0, M-1, 16
    Asub(tx,ty ) = A(i,kb+ty )
    Bsub(tx,ty ) = B(kb+tx,j )
    Bsub(tx,ty+16) = B(kb+tx,j+16)
    Bsub(tx,ty+32) = B(kb+tx,j+32)
    Bsub(tx,ty+48) = B(kb+tx,j+48)
    call syncthreads()
    do k = 1,16
        Cij1 = Cij1 + Asub(tx,k) * Bsub(k,ty )
        Cij2 = Cij2 + Asub(tx,k) * Bsub(k,ty+16)
        Cij3 = Cij3 + Asub(tx,k) * Bsub(k,ty+32)
        Cij4 = Cij4 + Asub(tx,k) * Bsub(k,ty+48)
    enddo
    call syncthreads()
enddo
C(i,j ) = Cij1
C(i,j+16) = Cij2
C(i,j+32) = Cij3
C(i,j+48) = Cij4
```

実験環境

実験環境 (TSUBAME2.0 Thin ノード)

	CPU	GPU
Type	Intel Xeon Westmere-EP 2.9GHz	NVIDIA Fermi M2050
Cores	6 core	14SM, 448 CUDA core

コンパイラ・オプション

	compiler version	option
PGI	pgfortran 12.10-0	OpenACC: -O3 -acc -ta=nvidia,cc20,keepgpu CUDA Fortran: -O3 -Mcuda=cc20
Cray	ftn 8.1.0.165	-O3 -h accel=nvidia_20 -h system_alloc
CAPS	hmpp 3.3.0 host compiler ifort 11.1	--nvcc-options -arch,sm_20 --nvcc-options -O3 ifort -O3
CUDA C	nvcc 4.1	-O3 -arch=sm_20
Fortran	ifort 11.1	-O3 -static -xP -openmp

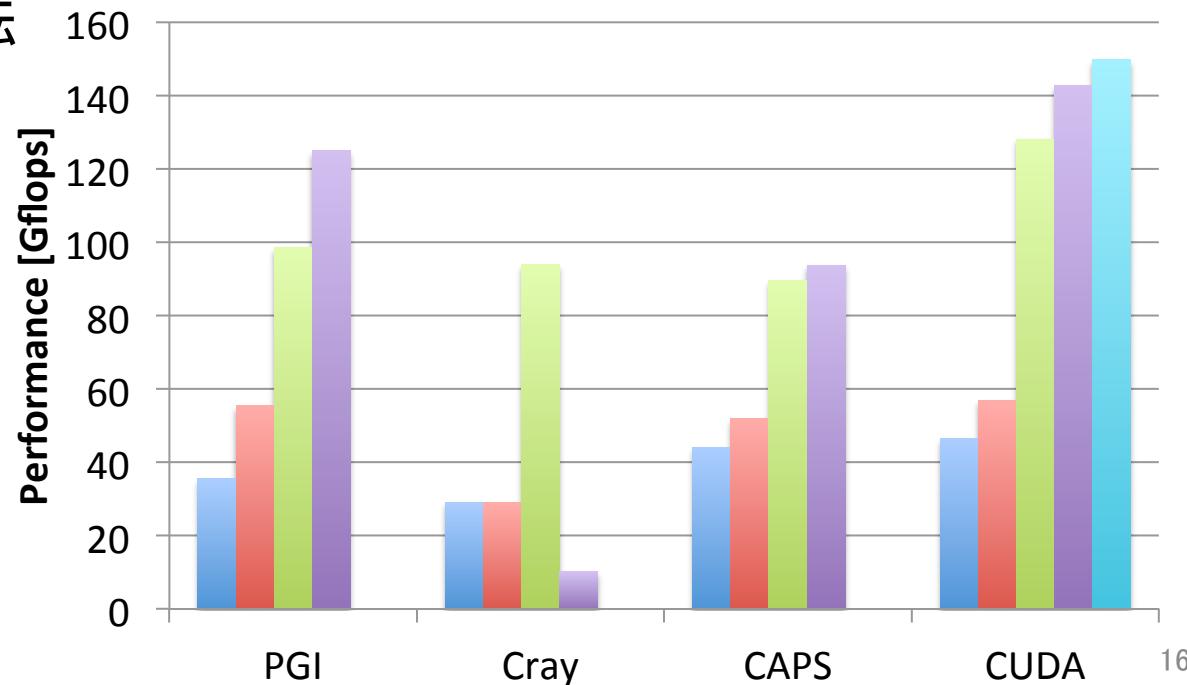
性能評価: 行列積

- 表は逐次のCPUコードからの変更・追加行数
- グラフは各最適化を適用した時の性能[GFlops]
- 行列サイズは 2048^2
- CPU-GPU間のデータ転送時間は含まない

オリジナルからの変更・追加行数

	Baseline	Thread mapping	Cache blocking	loop unrolling	Shared blocking
OpenACC	9	11	62	302	
CUDA	26	26	77	317	45

■ baseline
■ +thread mapping
■ +cache blocking
■ +loop unrolling
■ shared blocking



性能評価: 行列積

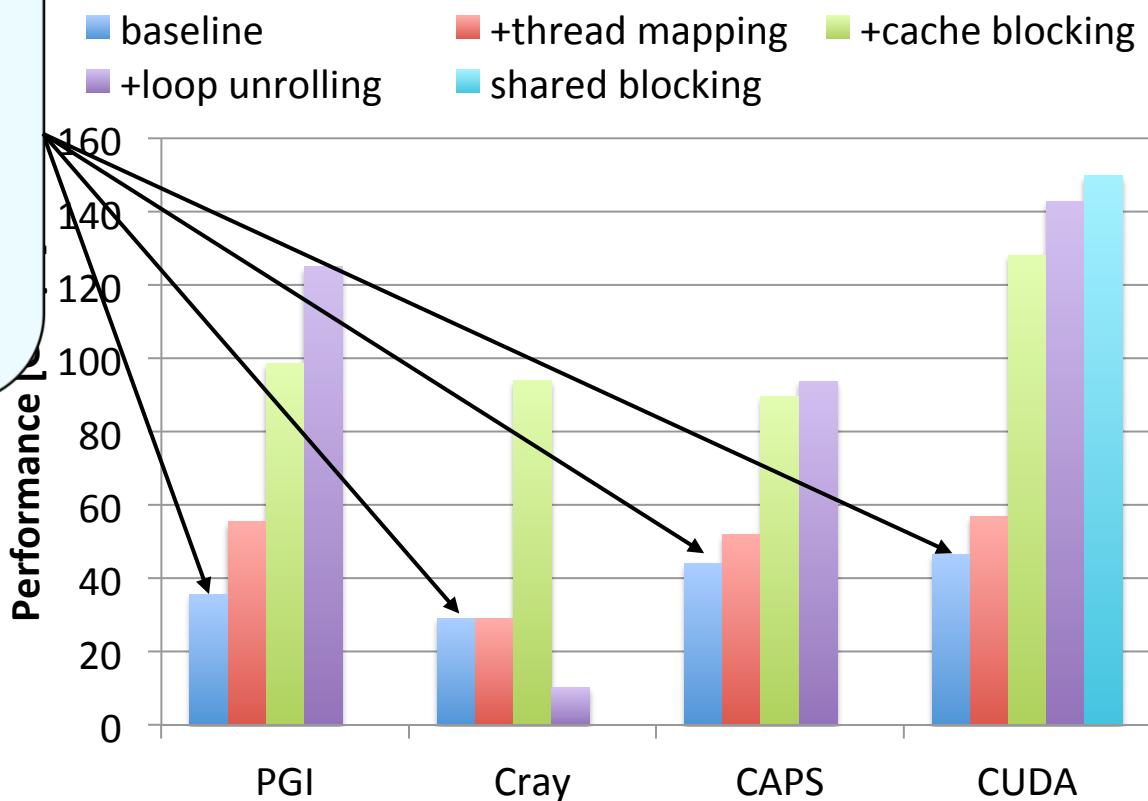
- 表は逐次のCPUコードからの変更・追加行数

OpenACC の baseline 実装では CUDA の **62%~94%** の性能

追加行数は CUDA の半分以下

オリジナルからの変更・追加行数

	Baseline	Thread mapping	Cache blocking	loop unrolling	Shared blocking
OpenACC	9	11	62	302	
CUDA	26	26	77	317	45



性能評価：行列積

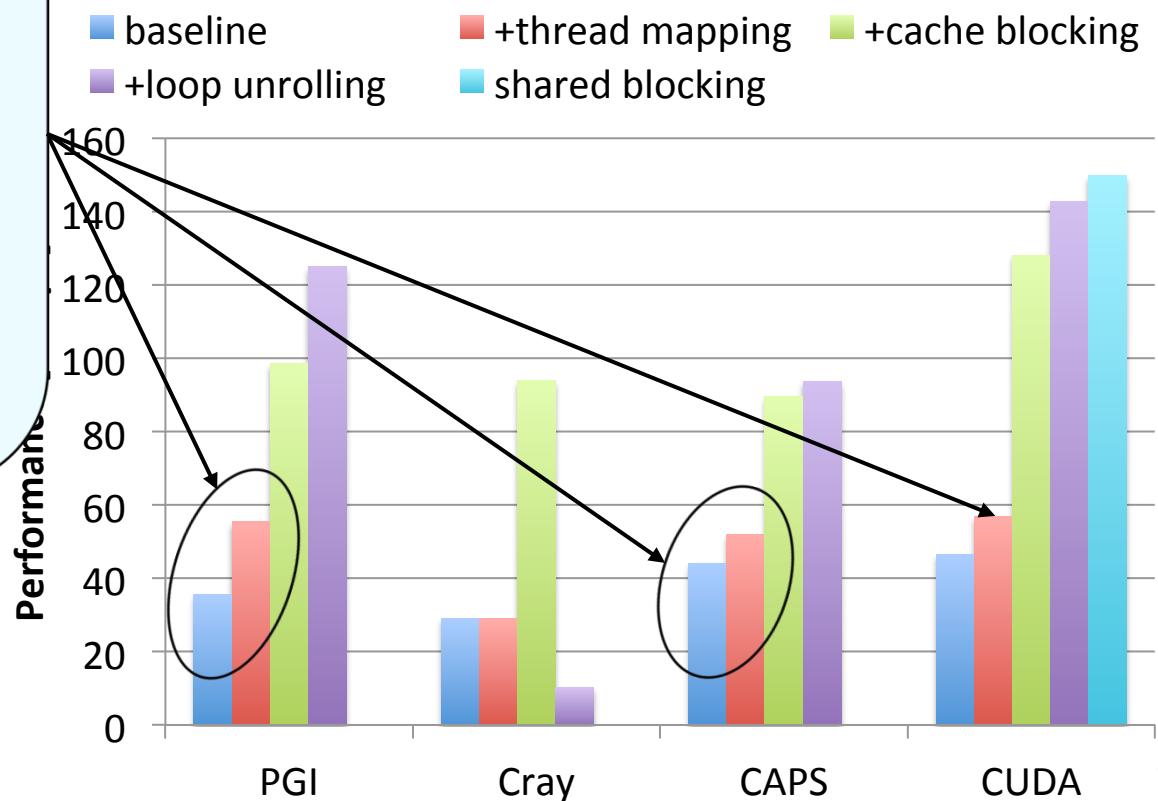
- 表は逐次のCPUコードか

スレッドブロックの調節により、PGI, CAPS でそれぞれBaselineの
1.6倍, 1.2倍の性能向上

特にPGI ではCUDAの
98% の性能を達成

オリジナルからの変更・追加行数

	Baseline	Thread mapping	Cache blocking	loop unrolling	Shared blocking
OpenACC	9	11	62	302	
CUDA	26	26	77	317	45



性能評価：行列積

ループアンローリングにより
cache blocking 版と比較して

PGI: 27% 性能向上

Cray: 89% 性能低下

CAPS: 4% 性能向上

CUDA: 11% 性能向上

この違いはレジスタスピルに
によるものと考えられる

PGI: 96 bytes

Cray: 2528 bytes

CAPS: 152 bytes

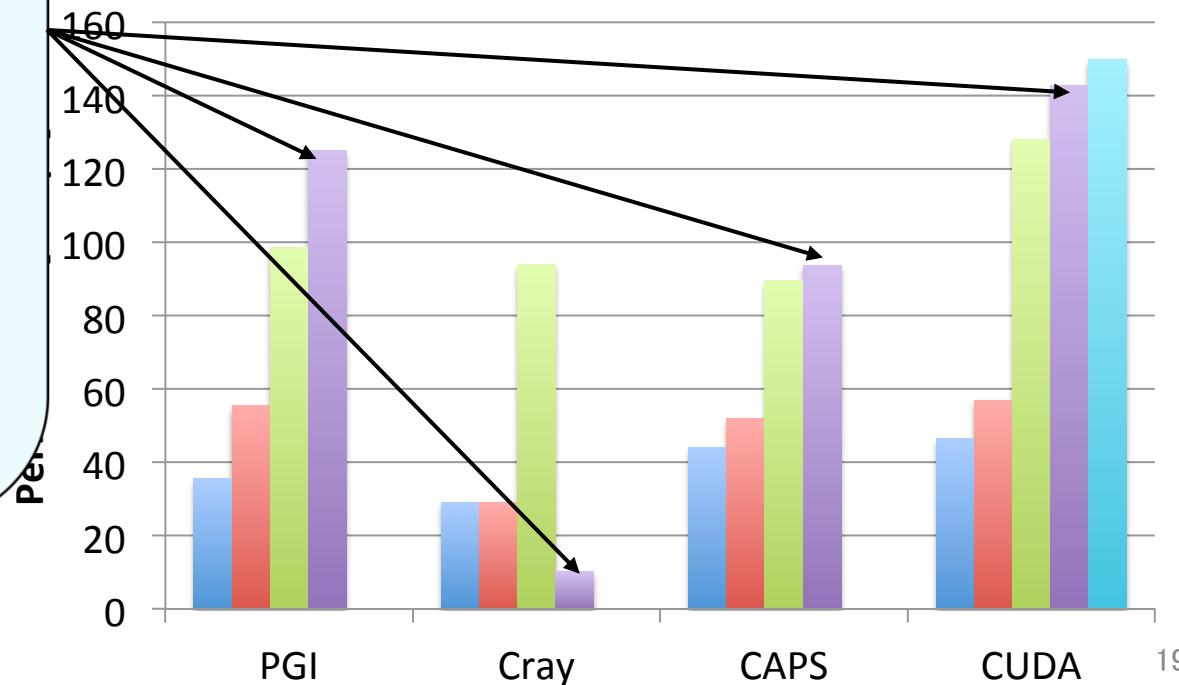
CUDA: 36 bytes

spill load

オリジナルからの変更・追加行数

	Baseline	Thread mapping	Cache blocking	loop unrolling	Shared blocking
OpenACC	9	11	62	302	
CUDA	26	26	77	317	45

■ baseline ■ +thread mapping ■ +cache blocking
■ +loop unrolling ■ shared blocking



性能評価：行列積

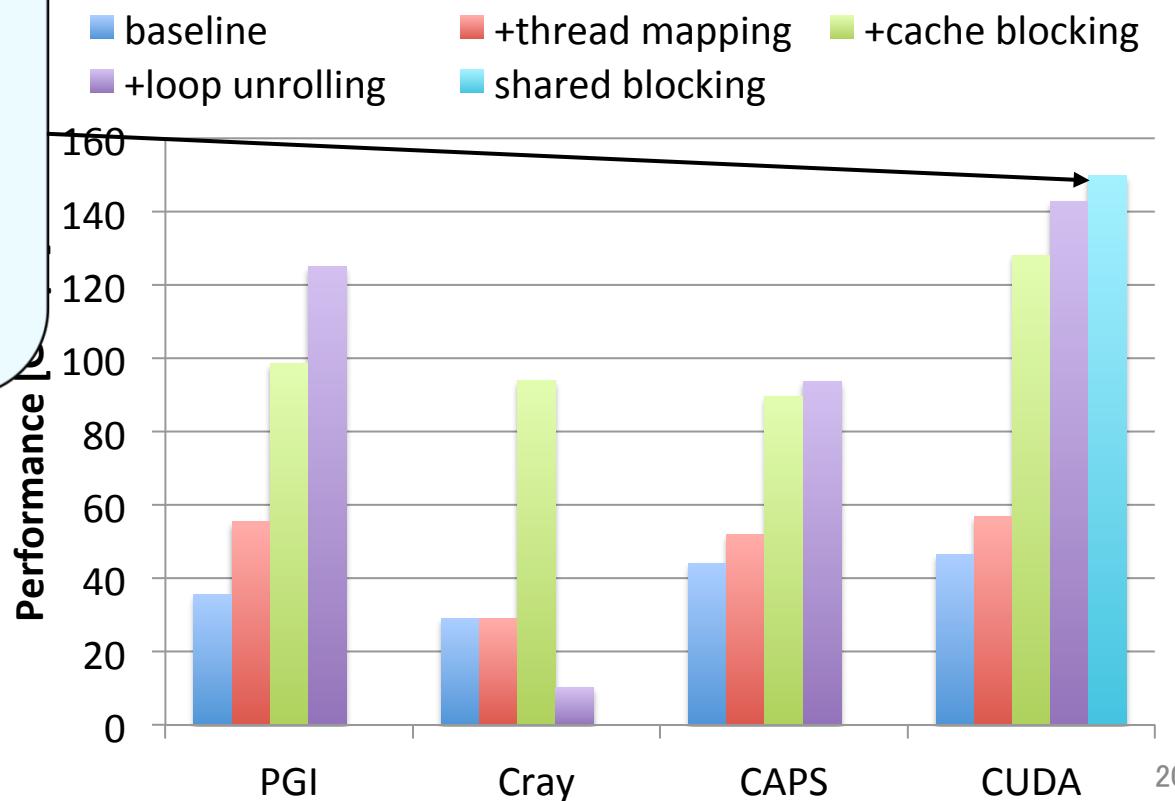
- 表は逐次のCPUコードからの変更・追加行数

CUDAの shared blocking を適用したものが最も高性能

ただし、OpenACC版では同期のための指示文がないため、shared blocking は適用できない

オリジナルからの変更・追加行数

	Baseline	Thread mapping	Cache blocking	loop unrolling	Shared blocking
OpenACC	9	11	62	302	
CUDA	26	26	77	317	45



最適化: 7点ステンシル

- Baseline
 - kernels, 3つのloop ディレクティブを用いる
 - data ディレクティブを用いて配列は転送済み
 - CUDA 版では x, y ループを並列化し, zループを逐次計算. スレッドブロックサイズは(64,4)
- Thread Mapping
- Branch Hoisting
- Register Blocking

```
!$acc kernels present(f1, f2)
 !$acc loop
    do z = 1, nz
 !$acc loop
    do y = 1, ny
 !$acc loop
    do x = 1, nx
        w = -1; e = 1; n = -1;
        s = 1; b = -1; t = 1;
        if(x==1) w=0; if(x==nx) e=0;
        if(y==1) n=0; if(y==ny) s=0;
        if(z==1) b=0; if(z==nz) t=0;
        f2(x,y,z) = cc*f1(x,y,z) &
                    + cw*f1(x+w,y,z) + ce*f1(x+e,y,z) &
                    + cs*f1(x,y+s,z) + cn*f1(x,y+n,z) &
                    + cb*f1(x,y,z+b) + ct*f1(x,y,z+t)
    end do
 end do
 end do
 !$acc end kernels
```

最適化: 7点ステンシル

- Baseline
- Thread Mapping
 - 行列積同様, gang, worker, vector の調節を行う
 - この際, 最外のzループを最内にする, ループインターチェンジも行う
 - CAPS コンパイラが最外ループとそのひとつ内側のループしか並列化できないため
 - この変更で Cray, PGI のコンパイラの性能に変化は出ない
- Branch Hoisting
- Register Blocking

```
!$acc kernels present(f1, f2)
!$acc loop gang vector(THREAD_Y)
    do y = 1, ny
!$acc loop gang vector(THREAD_X)
    do x = 1, nx
!$acc loop seq
    do z = 1, nz
        w = -1; e = 1; n = -1;
        s = 1; b = -1; t = 1;
        if(x==1) w=0; if(x==nx) e=0;
        if(y==1) n=0; if(y==ny) s=0;
        if(z==1) b=0; if(z==nz) t=0;
        f2(x,y,z) = cc*f1(x,y,z) &
                    + cw*f1(x+w,y,z) + ce*f1(x+e,y,z) &
                    + cs*f1(x,y+s,z) + cn*f1(x,y+n,z) &
                    + cb*f1(x,y,z+b) + ct*f1(x,y,z+t)
    end do
end do
end do
!$acc end kernels
```

最適化: 7点ステンシル

- Thread Mapping
- Branch Hoisting
 - 最内のzループ中にある、境界条件のための6つの分岐をループ外に括りだす
 - 最初と最後のイテレーションのみを別に実行することで、分岐を除去
- Register Blocking

```
!$acc kernels present(f1,f2)
!$acc loop gang vector(THREAD_Y)
    do y = 1, ny
!$acc loop gang vector(THREAD_X)
    do x = 1, nx
        z = 1;
        w = -1; e = 1; n = -1; s = 1;
        if(x == 1) w = 0; if(x == nx) e = 0
        if(y == 1) n = 0; if(y == ny) s = 0

        f2(x,y,z) = cc * f1(x,y,z) + cw * f1(x+w,y,z) &
                    + ce * f1(x+e,y,z) + cs * f1(x,y+s,z)      &
                    + cn * f1(x,y+n,z) + cb * f1(x,y,z) + ct * f1(x,y,z+1)
        do z = 2, nz-1

            f2(x,y,z) = cc * f1(x,y,z) + cw * f1(x+w,y,z) &
                        + ce * f1(x+e,y,z) + cs * f1(x,y+s,z)      &
                        + cn * f1(x,y+n,z) + cb * f1(x,y,z-1) + ct * f1(x,y,z+1)
        end do
        z = nz

        f2(x,y,z) = cc * f1(x,y,z) + cw * f1(x+w,y,z) &
                    + ce * f1(x+e,y,z) + cs * f1(x,y+s,z)      &
                    + cn * f1(x,y+n,z) + cb * f1(x,y,z-1) + ct * f1(x,y,z)
        end do
        end do
    !$acc end kernels
```

最適化: 7点ステンシル

- Thread Mapping
- Branch Hoisting
- Register Blocking
 - 最内のzループはひとつのスレッドで逐次実行されるため、いくつかのデータに再利用性がある
 - $(x, y, z-1), (x, y, z), (x, y, z+1)$ のデータをローカル変数に保存することで、再利用する

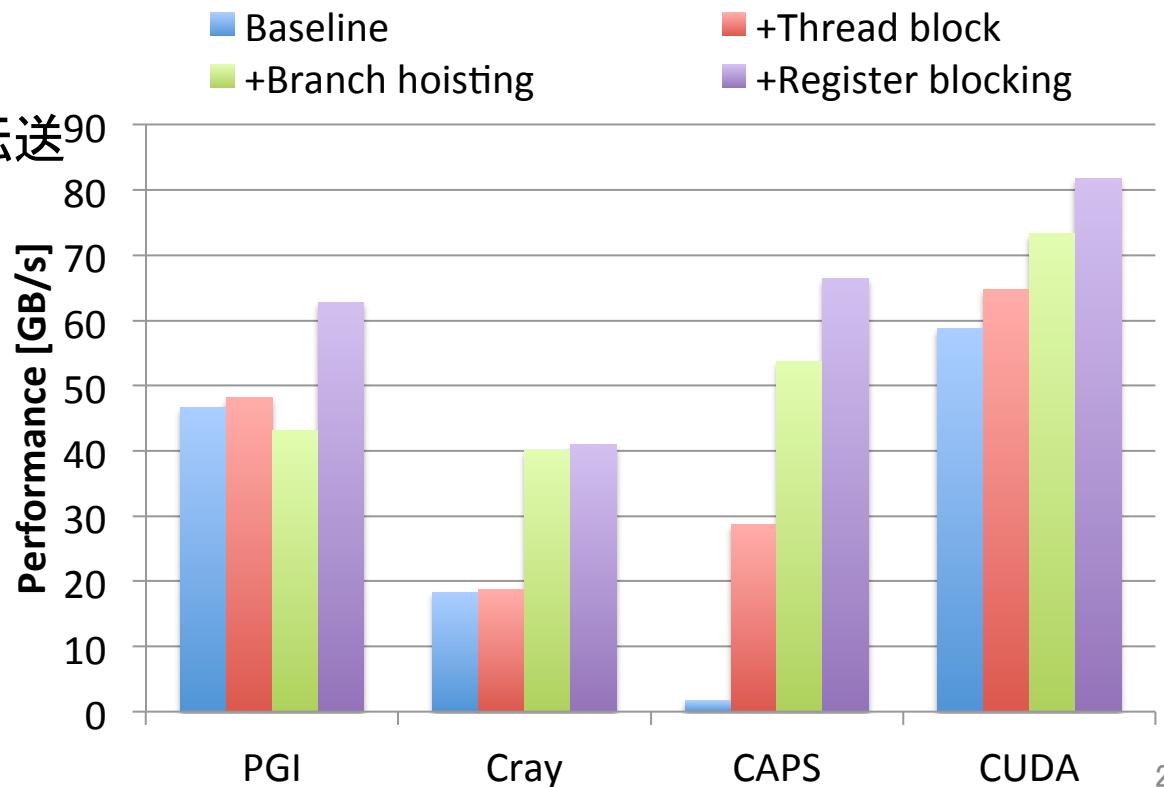
```
!$acc kernels present(f1,f2)
!$acc loop gang vector(THREAD_Y)
    do y = 1, ny
!$acc loop gang vector(THREAD_X)
    do x = 1, nx
        z = 1;
        w = -1; e = 1; n = -1; s = 1;
        if(x == 1) w = 0; if(x == nx) e = 0
        if(y == 1) n = 0; if(y == ny) s = 0
f_t = f1(x,y,z+1); f_c = f1(x,y,z); f_b = f_c
f2(x,y,z) = cc * f_c + cw * f1(x+w,y,z) &
+ ce * f1(x+e,y,z) + cs * f1(x,y+s,z) &
+ cn * f1(x,y+n,z) + cb * f_b + ct * f_t
do z = 2, nz-1
f_b = f_c; f_c = f_t; f_t = f1(x,y,z+1)
f2(x,y,z) = cc * f_c + cw * f1(x+w,y,z) &
+ ce * f1(x+e,y,z) + cs * f1(x,y+s,z) &
+ cn * f1(x,y+n,z) + cb * f_b + ct * f_t
end do
z = nz
f_b = f_c; f_c = f_t; f_t = f_t
f2(x,y,z) = cc * f_c + cw * f1(x+w,y,z) &
+ ce * f1(x+e,y,z) + cs * f1(x,y+s,z) &
+ cn * f1(x,y+n,z) + cb * f_b + ct * f_t
end do
end do
!$acc end kernels
```

性能評価: 7点ステンシル

- 表は逐次のCPUコードからの変更・追加行数
- グラフは各最適化を適用した時の性能[GB/s]
- 行列サイズは 256^2
- CPU-GPU間のデータ転送時間は含まない

オリジナルからの変更・追加行数

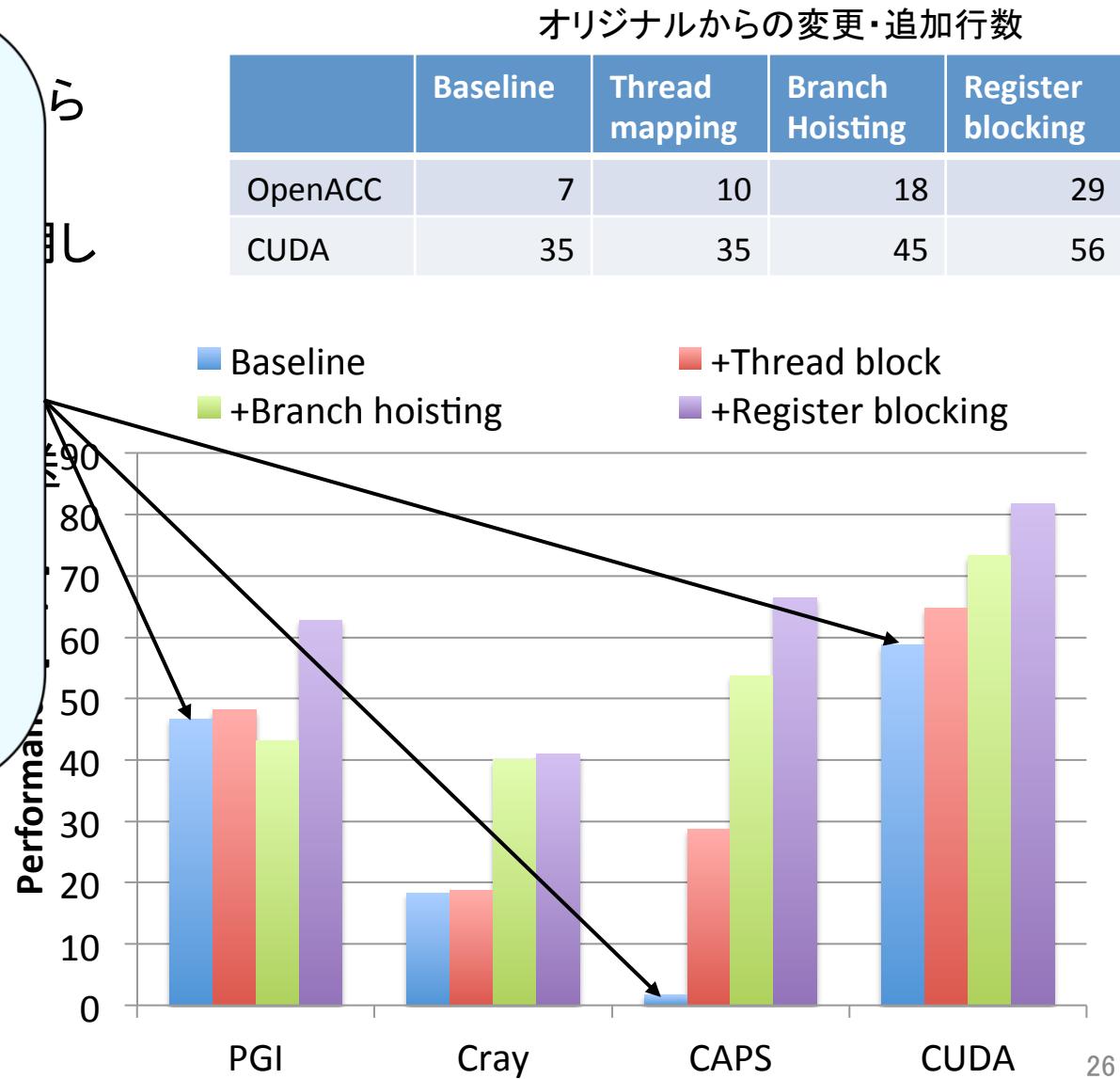
	Baseline	Thread mapping	Branch Hoisting	Register blocking
OpenACC	7	10	18	29
CUDA	35	35	45	56



性能評価: 7点ステンシル

PGIコンパイラでは、
CUDAの80%の性能

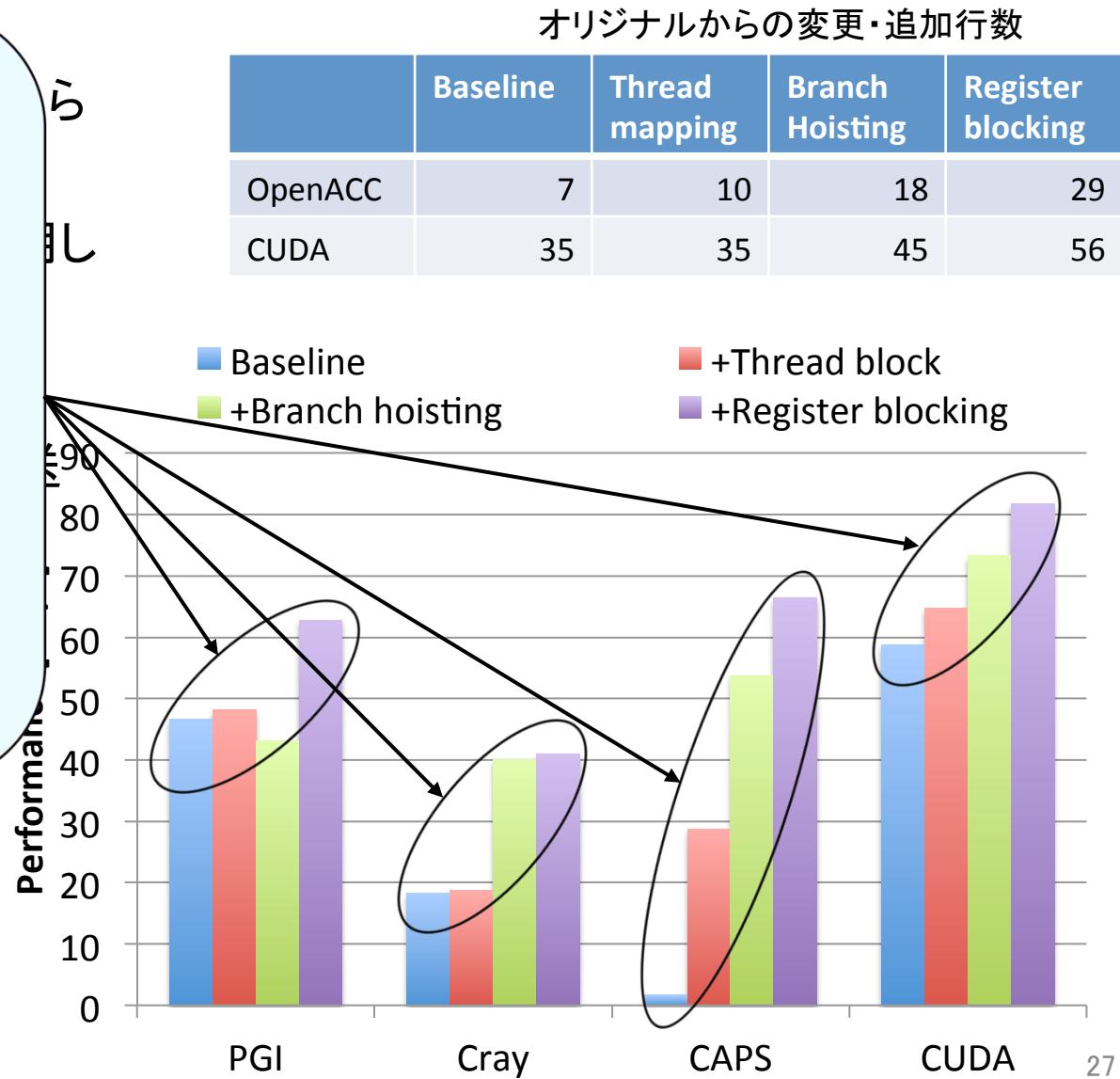
CAPSコンパイラは、
最外ループとその
ひとつ内側のループ
しか並列化できないた
め、性能が出ていない



性能評価: 7点ステンシル

最適化の適用により、
大きく性能向上

CUDAで効果のある最
適化はOpenACCでも効
果があると言える



最適化: UPACS

- **Baseline**
 - Cray, CAPS コンパイラではUPACS をコンパイルできなかったため, PGI のみで評価
 - 行列積・7点ステンシル同様, **kernel**, **loop** ディレクティブを利用
 - 現状のOpenACC では構造体が使えない, allocatable, pointer 属性の配列が使えないという制限があるため, OpenACC を適用できるようにプログラムを書き換え
 - 構造体の配列は, 構造体の各要素の配列に書き換えている
 - CUDA版は一度C言語に書き換えてからCUDAに書き換え
 - Baseline でのスレッドブロックのサイズは(64, 4)
- Kernel specialization
- Loop fusion
- Fine-grained parallelization in MFGS

最適化: UPACS

- Baseline
- Kernel specialization
 - オリジナルのUPACSでは、ひとつのサブルーチンで i, j, k 方向の差分計算を行うが、それぞれのサブルーチンに分割
 - レジスター使用量の減少、コンパイラによる最適化が効きやすくなる
 - ただしプログラミングの手間増大
- Loop fusion
- Fine-grained parallelization in MFGS

! i, j, k, and n are loop variables

im = $i - \text{idelta}(1)$

jm = $j - \text{idelta}(2)$

km = $k - \text{idelta}(3)$

$dq(i,j,k,n) = dq(i,j,k,n) - \text{inv_vol}(i,j,k) &$
* ($cflux(i,j,k,n) - cflux(im, jm, km, n)$)

↓ ↓ ↓

! i dimension

$dq(i,j,k,n) = dq(i,j,k,n) - \text{inv_vol}(i,j,k) &$
* ($cflux(i,j,k,n) - cflux(i-1, j, k, n)$)

! j dimension

$dq(i,j,k,n) = dq(i,j,k,n) - \text{inv_vol}(i,j,k) &$
* ($cflux(i,j,k,n) - cflux(i, j-1, k, n)$)

! k dimension

$dq(i,j,k,n) = dq(i,j,k,n) - \text{inv_vol}(i,j,k) &$
* ($cflux(i,j,k,n) - cflux(i, j, k-1, n)$)

最適化: UPACS

- Baseline
- Kernel specialization
- Loop fusion
 - Convection, Viscosity フェーズのいくつかの3重ループのペアは、一時領域を介してデータのやり取りを行う
 - レジスタやシェアードメモリを用いてループを融合することでグローバルメモリへのアクセスを抑制する
 - スレッド間でデータ依存がある場合にはシェアードメモリを用いる
- Fine-grained parallelization in MFGS

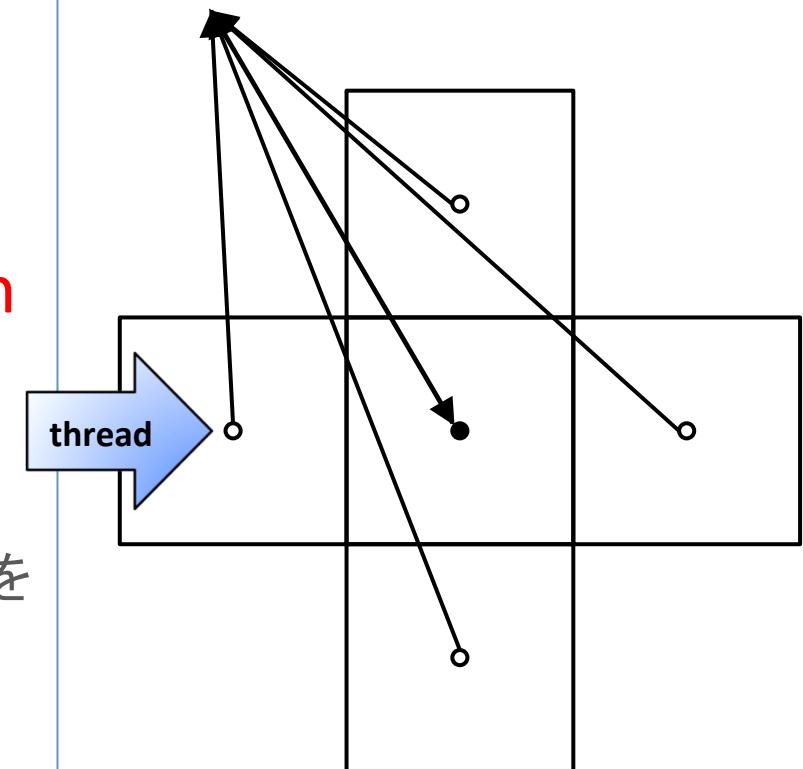
```
do k=1,kn ! loop nest A and B
  do j=1,jn
    do i=1,in
      < body of loop nest A >
      local_value = ( result of A )
      < body of loop nest B >
    end do
  end do
end do
```

最適化: UPACS

- Baseline
- Kernel specialization
- Loop fusion
- Fine-grained parallelization in the Matrix Free Gauss-Seidel Method (MFGS)
 - Time Integrationフェイズのほとんどを占めるMFGSサブルーチンの最適化
 - 隣接セルと中心セルから計算される値の総和により中心セルを更新
 - オリジナルでは各セルの計算を逐次に行い、最後に中心セルを更新

Original MFGS

local_value

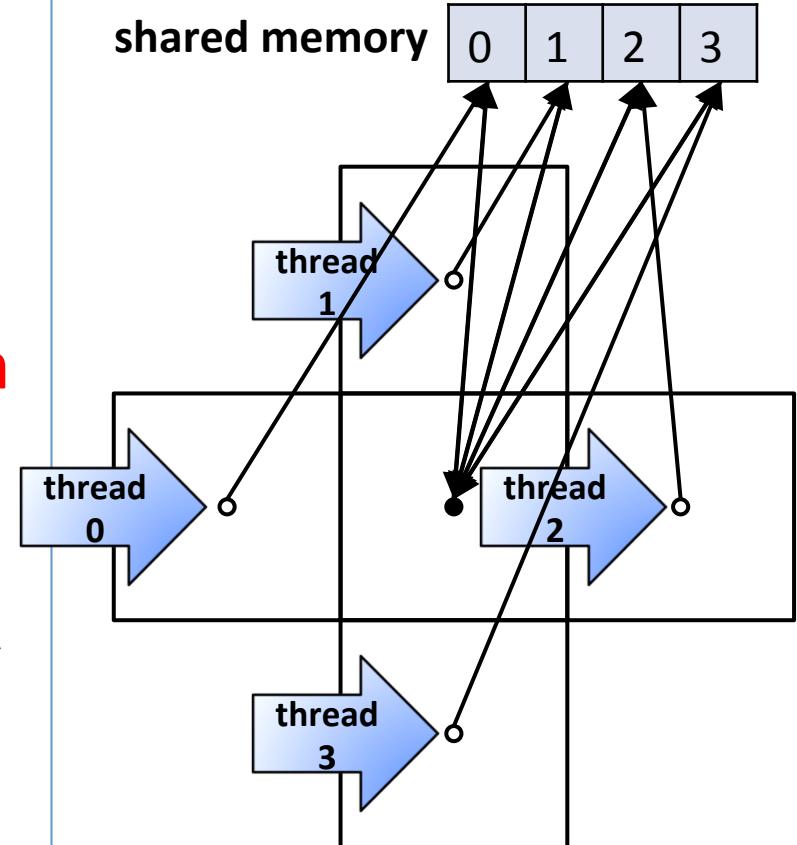


Time Integration stencil

最適化: UPACS

- Baseline
- Kernel specialization
- Loop fusion
- Fine-grained parallelization in the Matrix Free Gauss-Seidel Method (MFGS)
 - 最適化適用版では、各隣接セルごとにスレッドをたて、シェアードメモリを用いたリダクションを行い、中心セルを更新
 - シェアードメモリを用いるため、CUDAのみの適用

Fine-grained parallelization MFGS



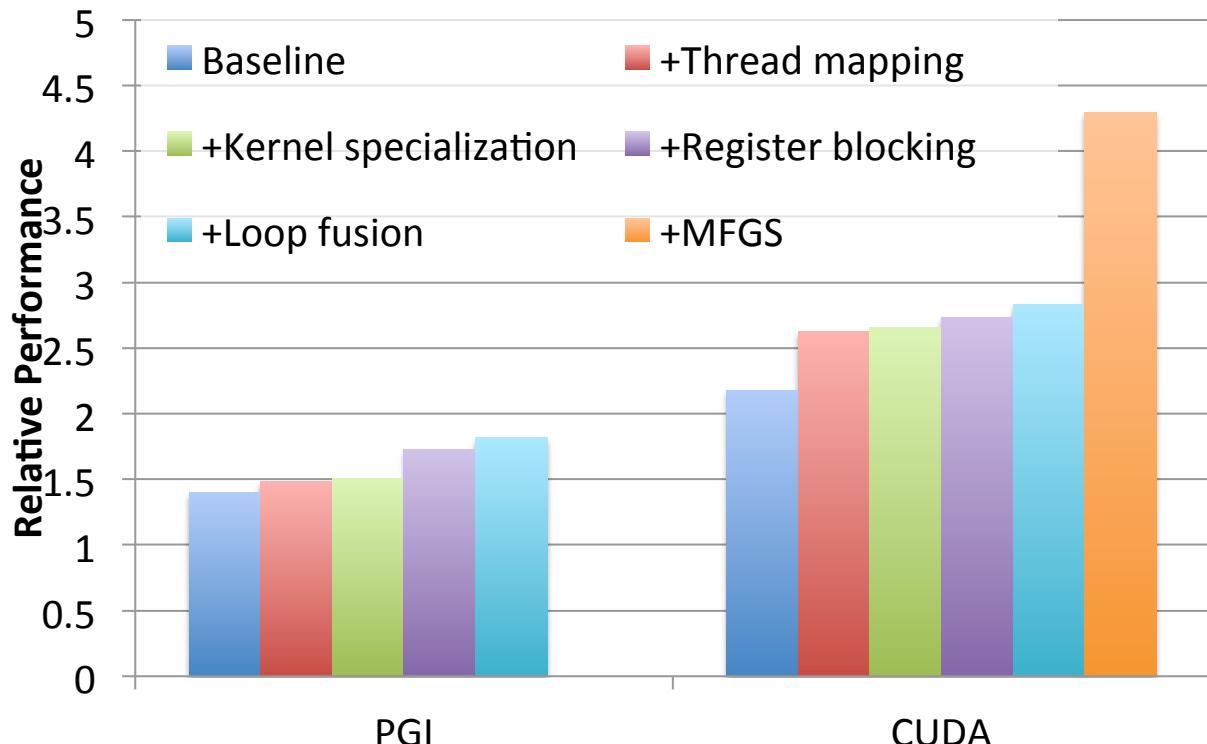
Time Integration stencil

性能評価: UPACS

- OpenMP 版の6並列と1イテレーションの実行時間を比較した相対性能
- ブロックサイズ 120^3

オリジナルからの変更・追加行数

	Baseline	Thread mapping	Kernel specialization	Register blocking	Loop fusion	MFGS
OpenACC	1788	1809	2601	2940	3296	
CUDA	5607	5743	6614	7641	8656	8870



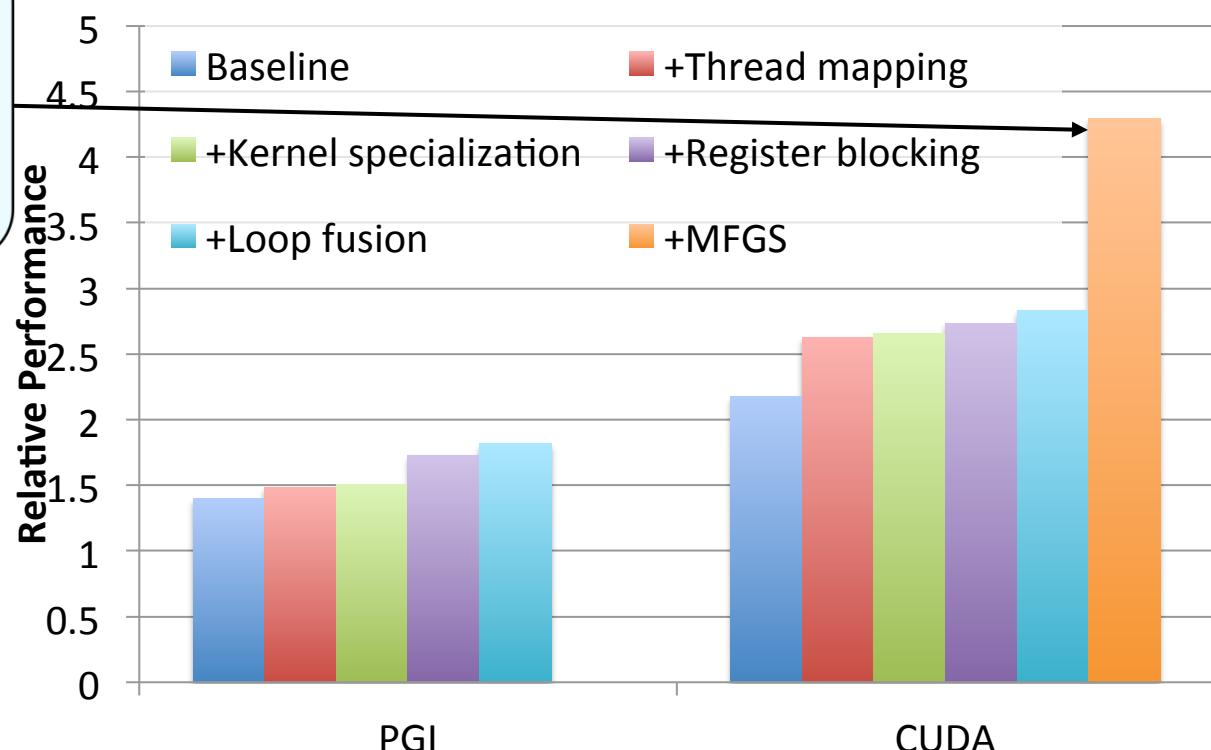
性能評価: UPACS

- OpenMP 版の6並列と1イテレーションの実行時間を比較した

shared memory を使った MFGS 最適化の効果が大きく、1.5倍程度 性能が向上している

オリジナルからの変更・追加行数

	Baseline	Thread mapping	Kernel specialization	Register blocking	Loop fusion	MFGS
OpenACC	1788	1809	2601	2940	3296	
CUDA	5607	5743	6614	7641	8656	8870



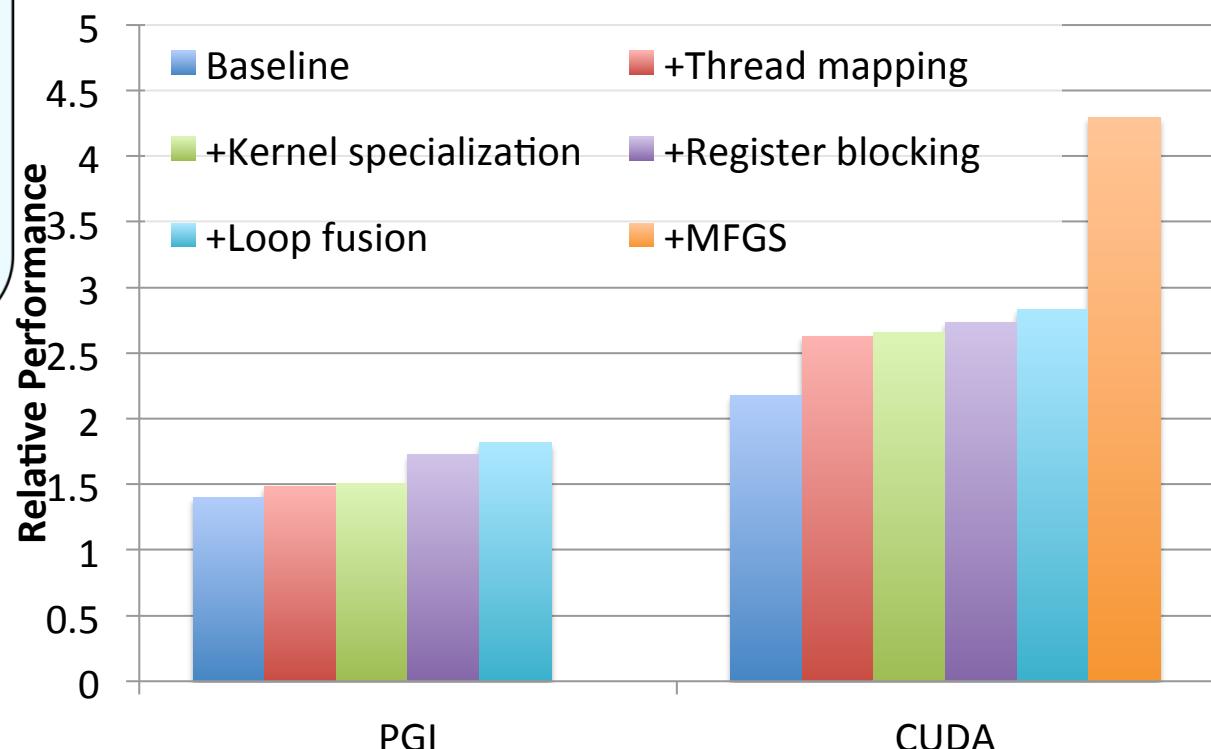
性能評価: UPACS

- OpenMP 版の6並列と1イテレーションの

OpenACC版の性能は、最大まで最適化した場合でもOpenMPの6並列の1.8倍程度、CUDAのBaseline の83% 程度の性能

オリジナルからの変更・追加行数

	Baseline	Thread mapping	Kernel specialization	Register blocking	Loop fusion	MFGS
OpenACC	1788	1809	2601	2940	3296	
CUDA	5607	5743	6614	7641	8656	8870



まとめ

- 行列積と7点移流計算ステンシル、および実アプリケーションUPACSを用いてOpenACCコンパイラを評価
- 現状のOpenACCコンパイラでは、CUDAのおよそ半分程度の性能が得られることがわかった
 - 問題・コンパイラによってはCUDAの98%の性能を達成する場合もある
- 共有メモリの制限から、CUDAと比較して大きな性能ギャップ有り
 - 対策：共有メモリの制御のような低レベルな指示文の追加拡張？
 - しかしポータビリティを損なう？

性能とプログラミングの手間・可搬性を達成するには？

- コンパイラによる最適化の限界
 - ループフュージョン→必ずしも性能向上するとは限らない
 - スレッドマッピングの最適化→コンパイラのみでは困難
- 自動チューニング必須
 - 最適化の有効化・無効化
 - パラメータ選択(スレッド数、ループアンローリング段数など)
- プログラム構造を変更するような最適化は？
 - 共有メモリを使った最適化