OpenACCの理想と現実

東京工業大学 星野哲也

GTC JAPAN2013 @六本木

自己紹介

- 東工大 松岡研究室 修士２年
- JAXA との共同研究
  - 数値流体アプリのメニーコア化
- 昨年から OpenACC を使用
  - Cray, CAPS, PGI 全てのコンパイラを利用可能
- 最近 OpenACC のコミッティに参加

OpenACCとは

- メニーコアアクセラレータ用の新しいプログラミングインターフェース
- NVIDIA, Cray, CAPS, PGI
  - Cray, CAPS, PGI 3社のコンパイラが現在利用可能
- C言語・Fortranに対応
- OpenMPのようなディレクティブベース
  - CPUプログラムに数行の指示文を挿入
  - アプリケーションの移植が比較的簡単
  - CPU・アクセラレータ用コードを単一プログラムとして記述, メンテナンスが容易

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

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

OpenACCのディレクティブ

webサイト: http://www.openacc.org/

最低限動くプログラムを作るまで

- OpenACCとCUDAで最低限動く行列積プログラムを作るまで

OpenACC

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc

  do j = 1, n
    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

end subroutine matmul
```

CUDA

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc

  do j = 1, n
    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

end subroutine matmul
```

最低限動くプログラムを作るまで

1. オフロードする領域を決める
   - OpenACC は **kernels** ディレクティブを挿入して終了

OpenACC

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
!$acc kernels
   do j = 1, n
      do i = 1, n
         cc = 0
         do k = 1, n
            cc = cc + a(i,k) * b(k,j)
         end do
         c(i,j) = cc
      end do
   end do
!$acc end kernels
end subroutine matmul
```

CUDA

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc

  do j = 1, n
     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

end subroutine matmul
```

最低限動くプログラムを作るまで

2. オフロード領域の並列化、カーネルコードの記述

OpenACC

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
!$acc kernels
   do j = 1, n
      do i = 1, n
         cc = 0
         do k = 1, n
            cc = cc + a(i,k) * b(k,j)
         end do
         c(i,j) = cc
      end do
   end do
!$acc end kernels
end subroutine matmul
```

CUDA

```
attribute(global) subroutine mm_cuda(a, b, c, n)
  integer, value :: n
  real(8), dimension(n, n) :: a, b, c
  integer :: i, j, k
  real(8) :: cc
  i = (blockidx%x-1) * blockdim%x + threadidx%x
  j = (blockidx%y-1) * blockdim%y + threadidx%y
  cc = 0.0
  do k = 1, n
    cc = cc + a(i, k) * b(k, j)
  end do
  c(i, j) = cc
end subroutine mm_cuda

subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
```

```
end subroutine matmul
```

最低限動くプログラムを作るまで

3. GPU用のメモリを確保し、明示的にデータ転送する

OpenACC

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
!$acc kernels
   do j = 1, n
      do i = 1, n
         cc = 0
         do k = 1, n
            cc = cc + a(i,k) * b(k,j)
         end do
         c(i,j) = cc
      end do
   end do
!$acc end kernels
end subroutine matmul
```

CUDA

```
attribute(global) subroutine mm_cuda(a, b, c, n)
  integer, value :: n
  real(8), dimension(n, n) :: a, b, c
  integer :: i, j, k
  real(8) :: cc
  i = (blockidx%x-1) * blockdim%x + threadidx%x
  j = (blockidx%y-1) * blockdim%y + threadidx%y
  cc = 0.0
  do k = 1, n
    cc = cc + a(i, k) * b(k, j)
  end do
  c(i, j) = cc
end subroutine mm_cuda

subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
  real(8), device, allocatable, dimension(:, :) ::   &
     a_dev, b_dev, c_dev


  allocate(a_dev(n, n), b_dev(n, n), c_dev(n, n))
  a_dev(:, :) = a(:, :)
  b_dev(:, :) = b(:, :)



  c(:, :) = c_dev(:, :)
end subroutine matmul
```

最低限動くプログラムを作るまで

4. 明示的にスレッドを割り当てる

OpenACC

```
subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
!$acc kernels
   do j = 1, n
      do i = 1, n
         cc = 0
         do k = 1, n
            cc = cc + a(i,k) * b(k,j)
         end do
         c(i,j) = cc
      end do
   end do
!$acc end kernels
end subroutine matmul
```

CUDA

```
attribute(global) subroutine mm_cuda(a, b, c, n)
  integer, value :: n
  real(8), dimension(n, n) :: a, b, c
  integer :: i, j, k
  real(8) :: cc
  i = (blockidx%x-1) * blockdim%x + threadidx%x
  j = (blockidx%y-1) * blockdim%y + threadidx%y
  cc = 0.0
  do k = 1, n
    cc = cc + a(i, k) * b(k, j)
  end do
  c(i, j) = cc
end subroutine mm_cuda

subroutine matmul(a, b, c, n)
  real(8), dimension(n, n) :: a, b, c
  integer :: n
  integer :: i, j, k
  real(8) :: cc
  real(8), device, allocatable, dimension(:, :) ::   &
     a_dev, b_dev, c_dev
  type(dim3) :: dimGrid, dimBlcok

  allocate(a_dev(n, n), b_dev(n, n), c_dev(n, n))
  a_dev(:, :) = a(:, :)
  b_dev(:, :) = b(:, :)
  dimGrid = dim3( n/16, n/16, 1)
  dimBlock = dim3( 16, 16, 1)
  call mm_cuda<<<dimGrid, dimBlock>>>(a, b, c, n)
  c(:, :) = c_dev(:, :)
end subroutine matmul
```

OpenACC の主要ディレクティブ

1. **kernels**, **parallel** ディレクティブによるオフロード領域指定
2. **data** ディレクティブによるデータのマネージメント
3. **loop** ディレクティブによるスレッドマッピングの指示

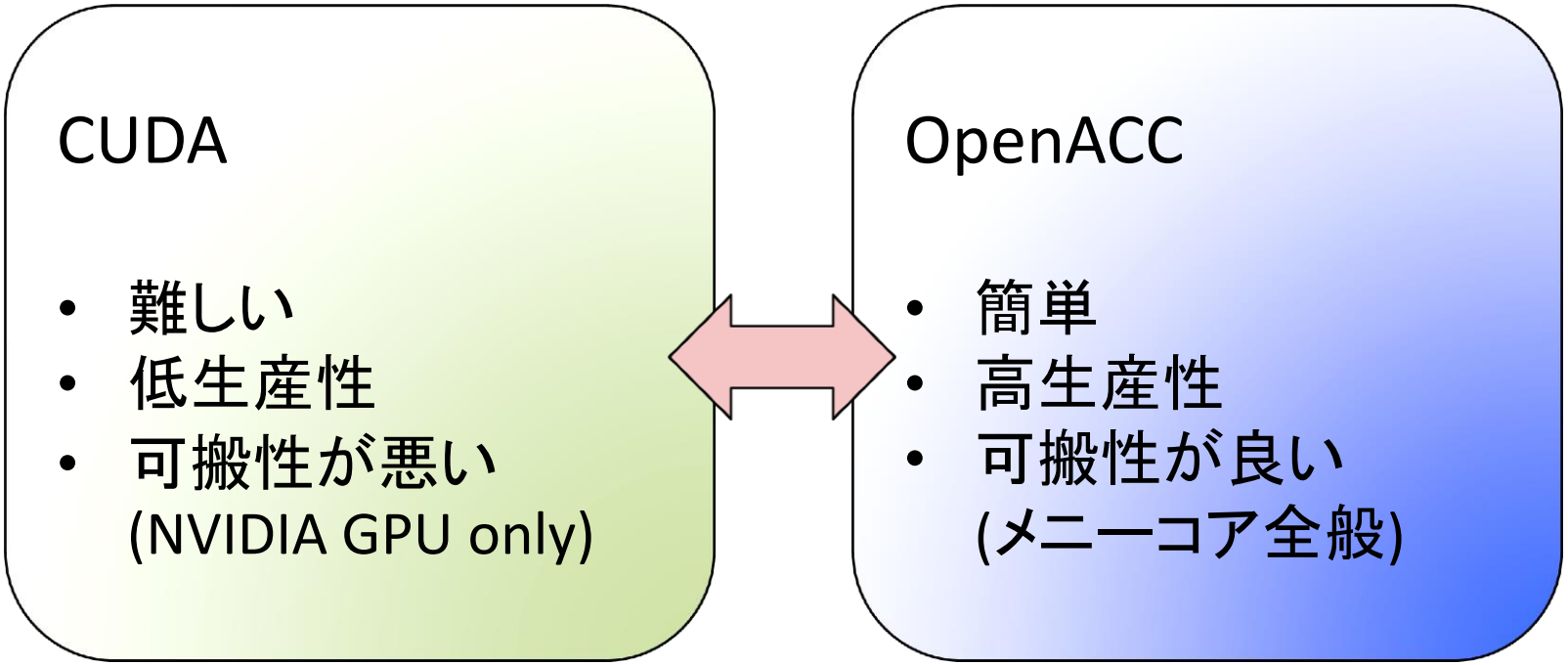
OpenACC と CUDAの違い

	OpenACC	CUDA
オフロード領域の指定	explicit	explicit
データマネージメント	implicit or explicit	explicit
ループマッピング	implicit or explicit	explicit

例: C行列積(PGI版)

```
#pragma data copy(c[0:n*n]),
copyin(a[0:n*n], b[0:n*n])
#pragma kernels
#pragma loop independent gang vector(16)
  for (i = 0; i < n; i++){
#pragma loop independent gang vector(16)
    for (j = 0;j < n; j++){
      cc = 0
      for (k = 0; k < n; k++){
        cc += a[i*n+k] * b[k*n+j]
      }
      c[i*n+j] = cc
    }
}
```

OpenACC の理想



理想は・・・

難しいCUDAに代わるメニーコア版のOpenMP

OpenACC の現実

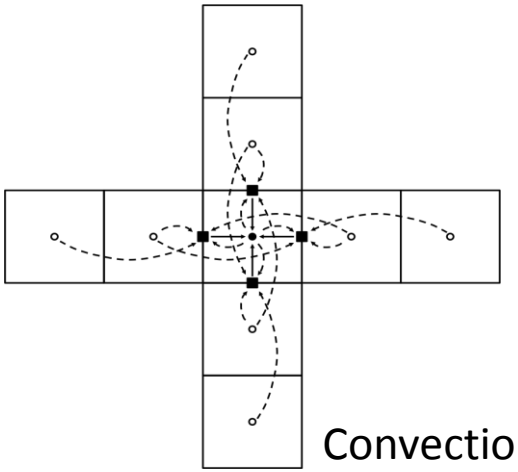
- 検証すべき点
  - OpenACC は簡単か？
  - OpenACC で性能は得られるか？
  - OpenACC の可搬性は？
- 検証方法
  - いくつかのカーネルベンチマーク、JAXAの流体アプリケーションを用いて検証

OpenACC は簡単か？

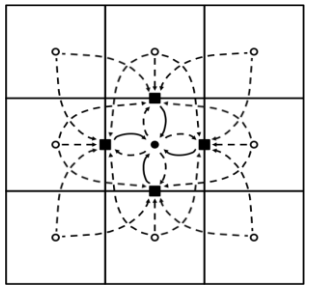
- そもそもCUDAは難しいか？
  - 最低限動くプログラムを作るのは大して難しくない！
  - CUDAで難しいのは最適化
    - コアレッシング・シェアードメモリ・ウォープダイバージェンス etc.
    - OpenACCでの最適化は簡単か？
- OpenMP と比較して煩わしい点
  - CPU-GPU間のデータ転送最適化がほぼ必須
  - 構造体が使えない

UPACS
(Unified Platform for Aerospace Computational Simulation)

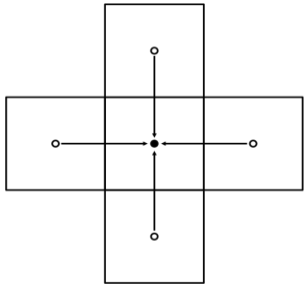
- 宇宙航空研究開発機構(JAXA)により研究・開発
- 10万行程度の Fortran 90 コード
- 様々なCFDソルバを提供
  - この研究では Navier-Stokes 方程式を解くソルバを選択
  - Convection, Viscosity, Time Integration の3つフェーズで実行時間の約90%を占める



Convection



Viscosity



Time Integration

カーネルベンチマークの実装

- 行列積と7点のステンシル計算
  1. kernels でオフロード領域指定
  2. data でデータ転送
  3. loop でマッピング指示

```
!$acc data pcopyin(a, b), pcopy(c)
!$acc kernels
!$acc loop
   do j = 1, n
!$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
!$acc end kernels
!$acc end data
```

行列積

```
!$acc data pcopy(f1, f2)
!$acc kernels
!$acc loop
   do z = 1, nz
!$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
!$acc end data
```

7点ステンシル

実アプリケーションでの実装

- UPACS のOpenACC 化手順

```
do n=1,bdtv_nFlowVar
  do k=1,blk%kn
    do j=1,blk%jn
      do i=1,blk%in
        im = i - idelta(1)
        jm = j - idelta(2)
        km = k - idelta(3)
        blk%dq(i,j,k,n) = blk%dq(i,j,k,n) - blk%inv_vol(i,j,k) &
            * ( cface(i,j,k)%flux(n) - cface(im,jm,km)%flux(n) )
      end do
    end do
  end do
end do
```

convection フェーズの一部

実アプリケーションでの実装

- UPACS のOpenACC 化手順
  1. 構造体は使えないので、構造体をバラす
     - Structure of Array はポインタで取り出すだけ
     - Array of Structure はコピーを作る必要がある

```
double precision, pointer :: dq(:,:,:,:), inv_vol(:,:,:)
double precision :: flux(:,:,:,:)
integer :: in, jn, kn
dq => blk%dq;  inv_vol => blk%inv_vol
in = blk%in; jn = blk%jn; kn = blk%kn
do k = 1, kn
  do j = 1, jn
    do i = 1, in
      flux( :, i, j, k) = cface(i, j, k)%flux(:)
    end do
  end do
end do

do n = 1, bdtv_nFlowVar
   do k = 1, kn
     do j = 1, jn
       do i = 1, in
         im = i - idelta(1)
         jm = j - idelta(2)
         km = k - idelta(3)
         dq(i,j,k,n) = dq(i,j,k,n) - inv_vol(i,j,k) &
             * ( flux(n,i,j,k) - flux(n,im,jm,km) )
       end do
     end do
   end do
 end do
```

convection フェーズの一部

実アプリケーションでの実装

- UPACS のOpenACC 化手順
  1. 構造体は使えないので、構造体をバラす
     - Structure of Array はポインタで取り出すだけ
     - Array of Structure はコピーを作る必要がある
  2. kernels を挿入

```
double precision, pointer :: dq(:,:,:,:), inv_vol(:,:,:)
double precision :: flux(:,:,:,:)
integer :: in, jn, kn
dq => blk%dq;  inv_vol => blk%inv_vol
in = blk%in; jn = blk%jn; kn = blk%kn
do k = 1, kn
  do j = 1, jn
    do i = 1, in
      flux( :, i, j, k) = cface(i, j, k)%flux(:)
    end do
  end do
end do
!$acc kernels
do n = 1, bdtv_nFlowVar
   do k = 1, kn
     do j = 1, jn
       do i = 1, in
         im = i - idelta(1)
         jm = j - idelta(2)
         km = k - idelta(3)
         dq(i,j,k,n) = dq(i,j,k,n) - inv_vol(i,j,k) &
             * ( flux(n,i,j,k) - flux(n,im,jm,km) )
       end do
     end do
   end do
 end do
!$acc end kernels
```

convection フェーズの一部

実アプリケーションでの実装

- UPACS のOpenACC 化手順
  1. 構造体は使えないので、構造体をバラす
     - Structure of Array はポインタで取り出すだけ
     - Array of Structure はコピーを作る必要がある
  2. kernels を挿入
  3. 毎回データ転送を行うと大変遅いため、data ディレクティブでデータ転送の最適化は必須

```
double precision, pointer :: dq(:,:,:,:), inv_vol(:,:,:)
double precision :: flux(:,:,:,:)
integer :: in, jn, kn
dq => blk%dq;  inv_vol => blk%inv_vol
in = blk%in; jn = blk%jn; kn = blk%kn
do k = 1, kn
  do j = 1, jn
    do i = 1, in
      flux( :, i, j, k) = cface(i, j, k)%flux(:)
    end do
  end do
end do
!$acc data &
!$acc present(dq, inv_vol, flux, idelta)
!$acc kernels
do n = 1, bdtv_nFlowVar
   do k = 1, kn
     do j = 1, jn
       do i = 1, in
         im = i - idelta(1)
         jm = j - idelta(2)
         km = k - idelta(3)
         dq(i,j,k,n) = dq(i,j,k,n) - inv_vol(i,j,k) &
             * ( flux(n,i,j,k) - flux(n,im,jm,km) )
       end do
     end do
   end do
 end do
!$acc end kernels
!$acc end data
```

convection フェーズの一部

実アプリケーションでの実装

- UPACS のOpenACC 化手順
  1. 構造体は使えないので、構造体をバラす
     - Structure of Array はポインタで取り出すだけ
     - Array of Structure はコピーを作る必要がある
  2. kernels を挿入
  3. 毎回データ転送を行うと大変遅いため、data ディレクティブでデータ転送の最適化は必須
  4. loop ディレクティブ挿入

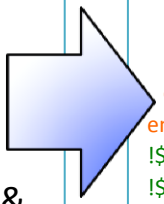
```
double precision, pointer :: dq(:,:,:,:), inv_vol(:,:,:)
double precision :: flux(:,:,:,:)
integer :: in, jn, kn
dq => blk%dq;  inv_vol => blk%inv_vol
in = blk%in; jn = blk%jn; kn = blk%kn
do k = 1, kn
  do j = 1, jn
    do i = 1, in
      flux( :, i, j, k) = cface(i, j, k)%flux(:)
    end do
  end do
end do
!$acc data &
!$acc present(dq, inv_vol, flux, idelta)
!$acc kernels
!$acc loop seq
do n = 1, bdtv_nFlowVar
!$acc loop seq
   do k = 1, kn
!$acc loop gang vector(4)
      do j = 1, jn
!$acc loop gang vector(64)
        do i = 1, in
          im = i - idelta(1)
          jm = j - idelta(2)
          km = k - idelta(3)
          dq(i,j,k,n) = dq(i,j,k,n) - inv_vol(i,j,k) &
              * ( flux(n,i,j,k) - flux(n,im,jm,km) )
        end do
      end do
   end do
 end do
!$acc end kernels
!$acc end data
```

convection フェーズの一部

実アプリケーションでの実装

Before

```
do n=1,bdtv_nFlowVar
  do k=1,blk%kn
    do j=1,blk%jn
      do i=1,blk%in
        im = i - idelta(1)
        jm = j - idelta(2)
        km = k - idelta(3)
        blk%dq(i,j,k,n) = blk%dq(i,j,k,n) - blk%inv_vol(i,j,k) &
            * ( cface(i,j,k)%flux(n) - cface(im,jm,km)%flux(n) )
      end do
    end do
  end do
end do
```



After

```
double precision, pointer :: dq(:,:,:,:), inv_vol(:,:,:)
double precision :: flux(:,:,:,:)
integer :: in, jn, kn
dq => blk%dq;  inv_vol => blk%inv_vol
in = blk%in; jn = blk%jn; kn = blk%kn
do k = 1, kn
  do j = 1, jn
    do i = 1, in
      flux( :, i, j, k) = cface(i, j, k)%flux(:)
    end do
  end do
end do
!$acc data &
!$acc present(dq, inv_vol, flux, idelta)
!$acc kernels
!$acc loop seq
do n = 1, bdtv_nFlowVar
!$acc loop seq
   do k = 1, kn
!$acc loop gang vector(4)
      do j = 1, jn
!$acc loop gang vector(64)
        do i = 1, in
          im = i - idelta(1)
          jm = j - idelta(2)
          km = k - idelta(3)
          dq(i,j,k,n) = dq(i,j,k,n) - inv_vol(i,j,k) &
              * ( flux(n,i,j,k) - flux(n,im,jm,km) )
        end do
      end do
    end do
  end do
!$acc end kernels
!$acc end data
```

convection フェーズの一部

OpenACC の性能は？

- 実験環境

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

- コンパイラ

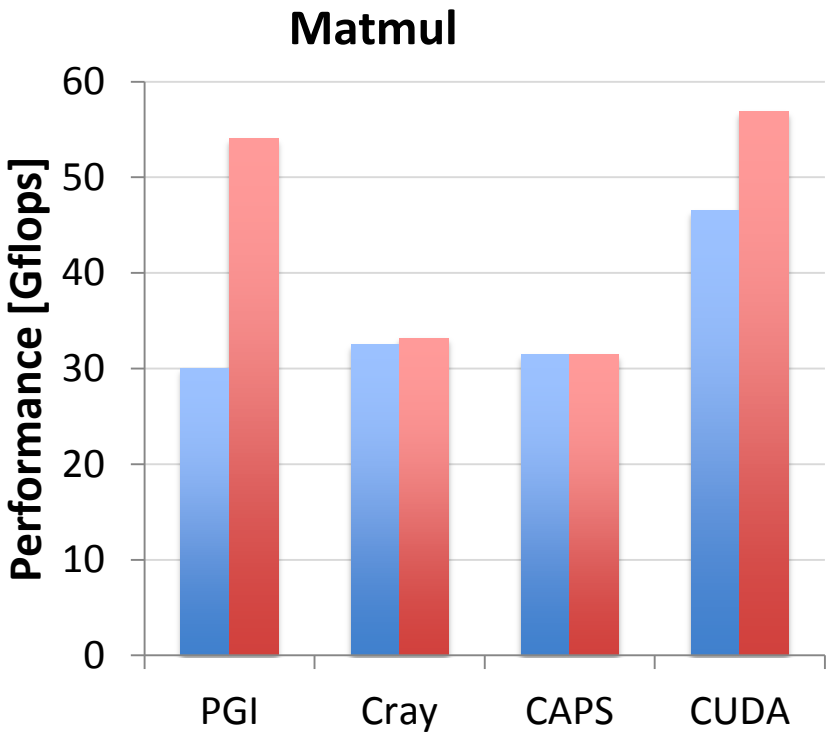
	compiler version	option
PGI	pgfortran 13.3	-O3 -acc -ta=nvidia,cc20
Cray	ftn 8.1.3	-O3 -h accel=nvidia_20 -h system_alloc –h acc_model=fast_addr
CAPS	capsmc 3.3.3 host compiler ifort 11.1	--nvcc-options -arch,sm_20 --nvcc-options -O3 ifort -O3
CUDA C	nvcc 4.1	-O3 -arch=sm_20
Fortran	pgfortran 13.3	-O3 –Mcuda=cc20

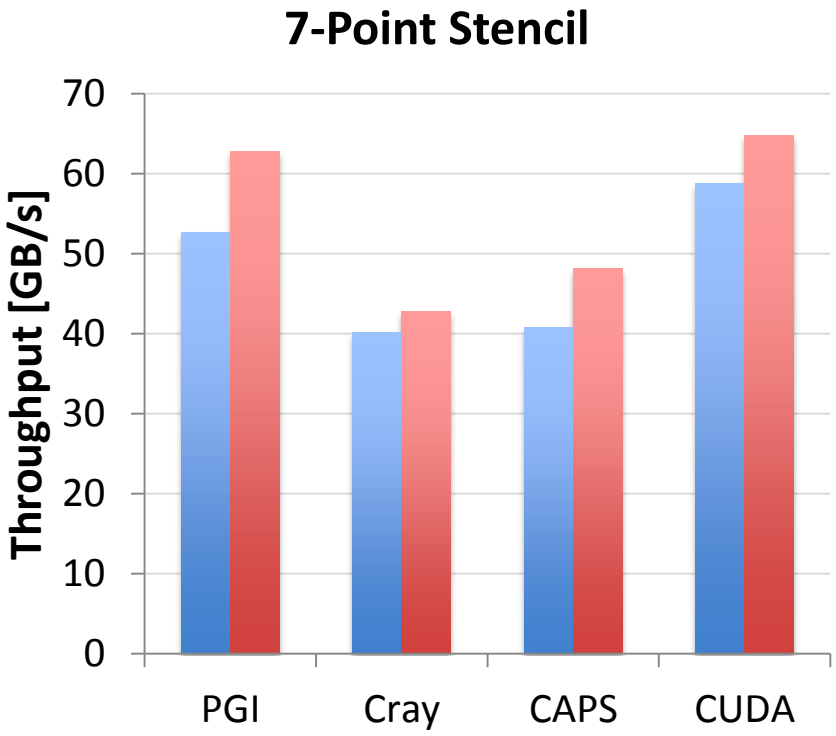
カーネルベンチマークの性能

- 行列積 : 倍精度, 2048^2
- 7-点ステンシル: 単性度, 256^3

書き換え行数

	行列積	7点ステンシル
OpenACC	9	7
CUDA	26	35



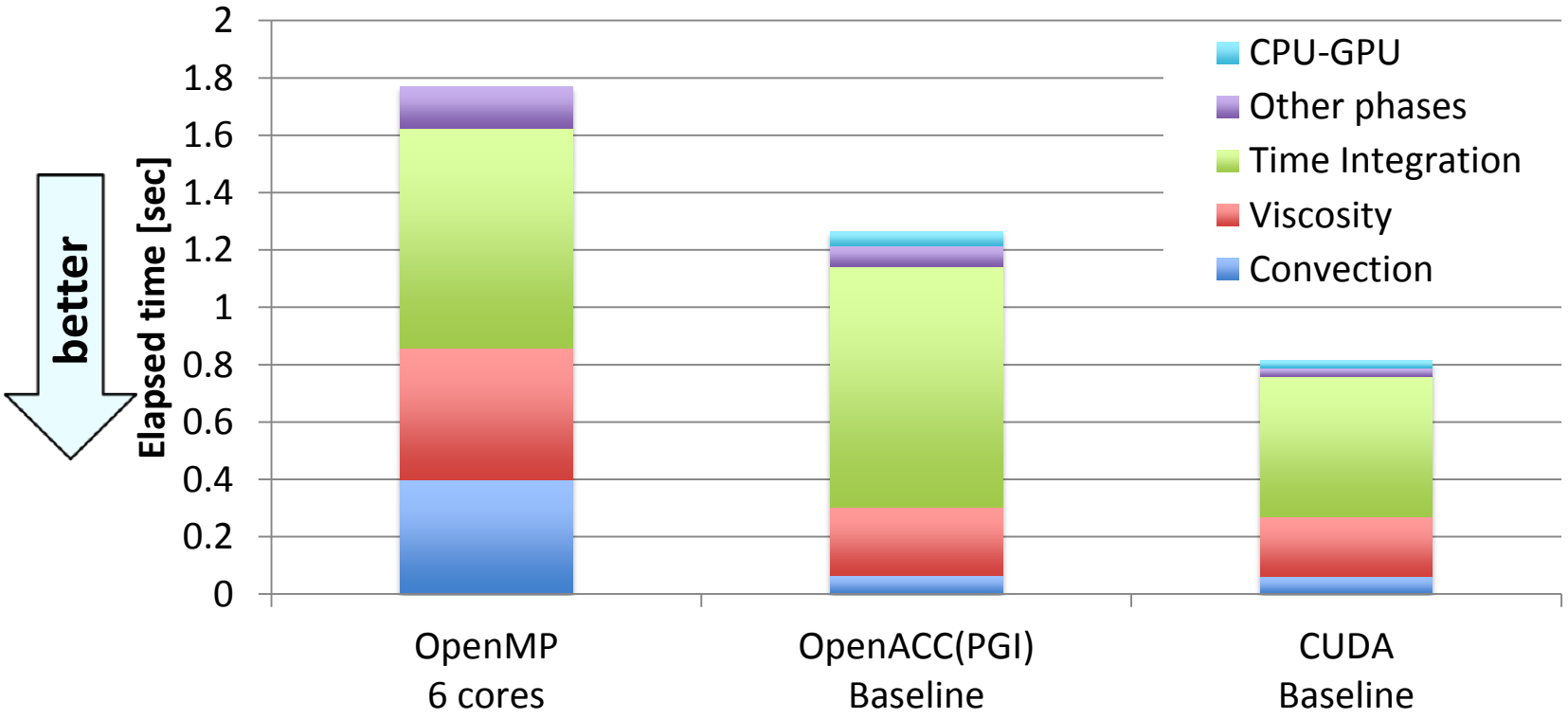


実アプリケーションでの性能

- UPACS を最低限動くように移植，データ転送の最適化を済ませた場合の性能

Table: Number of modified lines of code

	Baseline
OpenACC	1788
CUDA	5607

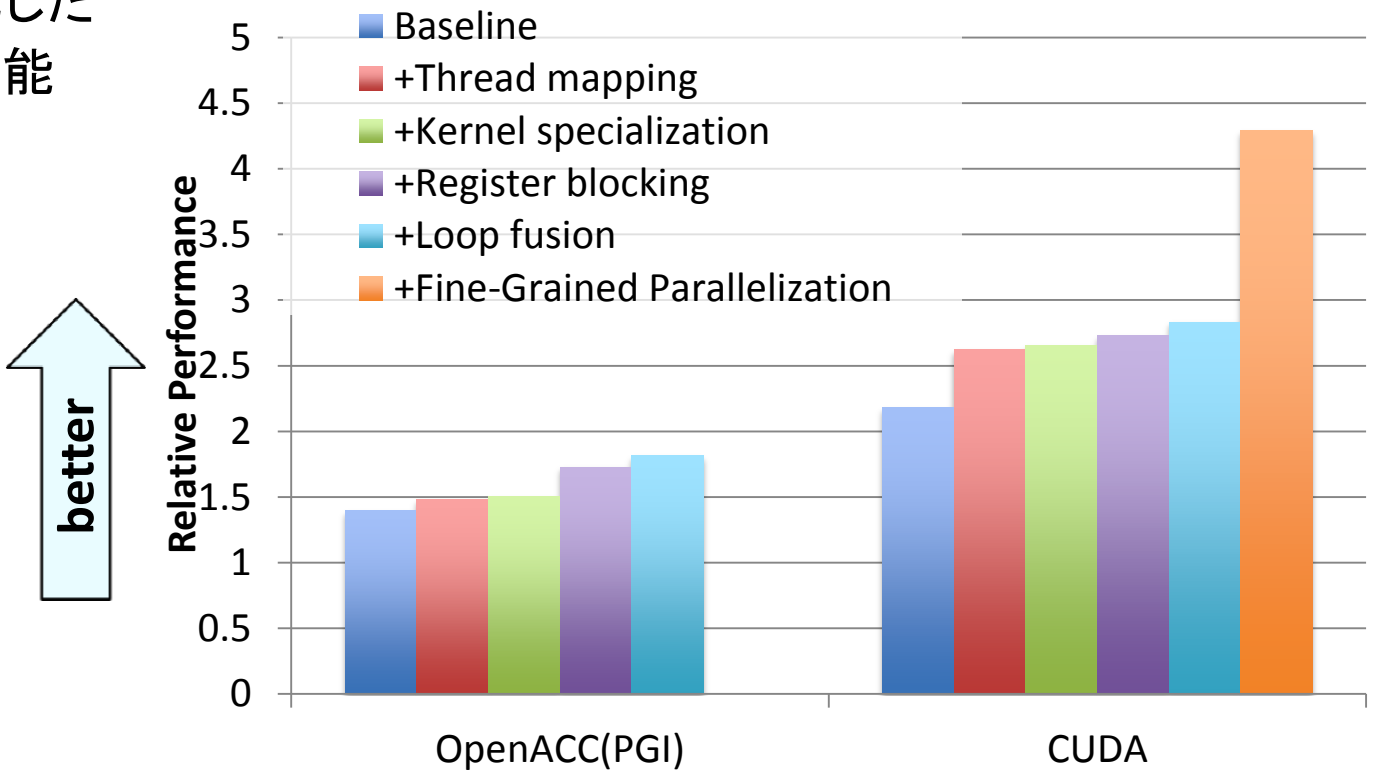


最適化を施した場合の性能

- 最適化なしの Baseline で CUDAの 64%の性能
- 最大限最適化した場合 42%の性能

Table: Number of modified lines of code

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



OpenACCでできないこと

- プログラムのしやすさに影響する制限
  - 構造体の利用
  - 並列実行領域内からの関数呼び出し
    - OpenACC 2.0 の仕様ではできるようになる
- 性能に影響する制限
  - shared を使ってスレッド間データ通信をする
  - texture等のハードウェアを明示的に使う
  - 配列の袖領域を通信する

→ これらの制限が CUDA と比較した際の性能ギャップの原因になることがある

OpenACC の可搬性は？

- 異なるアーキテクチャ間での可搬性
  - Intel MIC など，NVIDIA 以外のメニーコアへの対応は始まったばかり
- コンパイラ間での可搬性
  - 現行の Ver. 1.0 の仕様においては、統一されていない部分が多々ある

まとめ

- OpenACCの簡単さ
  - カーネルベンチマークでは、数行の挿入で移植可能
  - 構造体(特にArray of Structure)を使う実アプリでは、書き換え行数が増加
- OpenACCの性能
  - カーネルベンチマーク、実アプリケーションでそれぞれCUDAの8割、6割程度の性能
  - シェアードメモリ等を用いた最適化を行った場合、CUDAと大きな性能ギャップが生じる
- OpenACCの可搬性
  - 仕様解釈の違いからか、コンパイラ間で統一されていない
- OpenACCはまだ新しい仕様であるため、今後に期待

Q & A

Optimizations: Thread Mapping

- OpenACC Implementation
  - Explicitly mapping threads by using gang, worker and vector clauses
  - Choose the best size of (THREAD_X, THREAD_Y) by parameter survey
    - ex. (32, 4), (64, 4), (128, 2)
- CUDA Implementation
  - Choose the best size of thread block by parameter survey
  - Baseline thread block size
    - Matrix Multiplication: (16, 16)
    - 7-point stencil and UPACS: (64, 4)

```
!$acc kernels present(a, b, c)
!$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 matlrix multiplication (PGI version)

Optimizations: Shared Memory Blocking

- Use shared memory
- All threads in the same thread block use the on-chip data to compute inner products
- This optimization CANNOT be applied to OpenACC version
  - Because of inter-thread data communication

```
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
```

Optimizations: Branch Hoisting

- Move the six branches to outside of innermost z loop

```
!$acc kernels present(f1,f2)
!$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
```

Optimizations: Register Blocking

- Innermost z loop is executed sequentially by single thread
- Reuse data (x, y, z-1), (x, y, z), and (x, y, z+1) by using three local variables

```
!$acc kernels present(f1,f2)
!$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
```

Optimizations: Loop fusion

- Fuse loop nests pairs which have producer-consumer data flow
  - Convection and Viscosity phases have producer-consumer data flow with temporary 3-D array
- Use registers or shared memory to fuse these pairs
  - Depending on the existence of an inter-thread data dependency

NO inter-thread data dependency

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

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

Optimizations: Loop fusion

- Fuse loop nests pairs which have producer-consumer data flow
  - Convection and Viscosity phases have producer-consumer data flow with temporary 3-D array
- Use registers or shared memory to fuse these pairs
  - Depending on the existence of an inter-thread data dependency

inter-thread data dependency

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

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

Optimizations: Loop fusion

- Fuse loop nests pairs which have producer-consumer data flow
  - Convection and Viscosity phases have producer-consumer data flow with temporary 3-D array
- Use registers or shared memory to fuse these pairs
  - Depending on the existence of an inter-thread data dependency

Inter-thread data communication is required
CUDA: available by using shared memory
OpenACC: NOT available

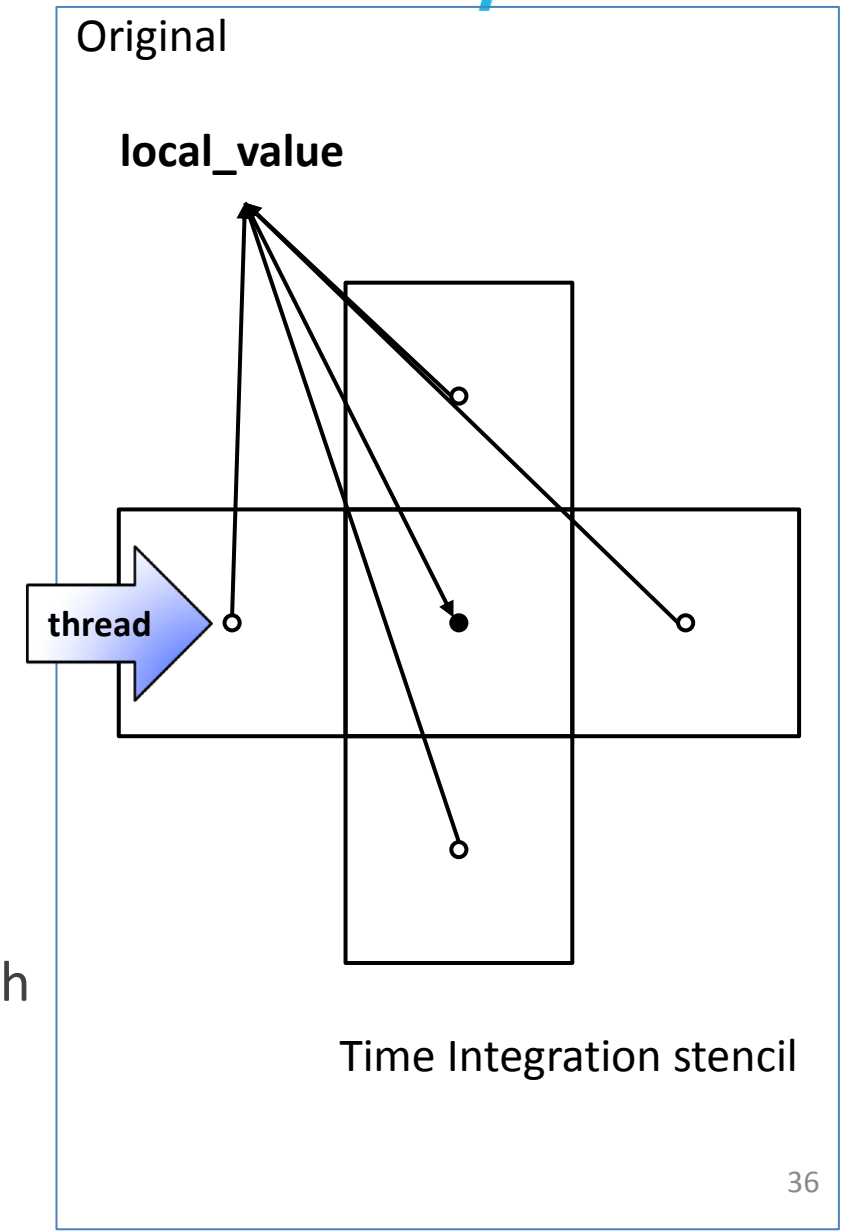
inter-thread data dependency

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

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

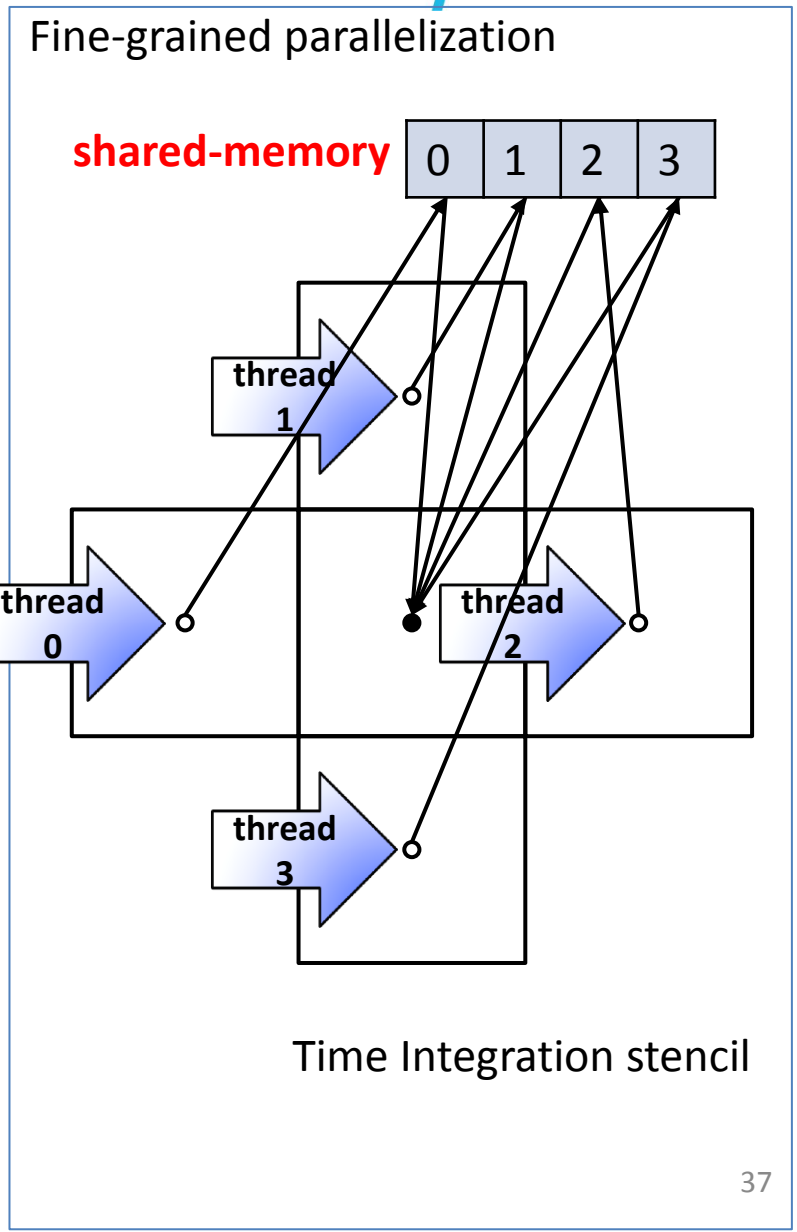
Optimizations: Fine-grained parallelization in Time Integration phase

- Optimizing Time Integration phase by using shared memory
  - The Time Integration phase computes the six neighbor cells serially to update the central cell
  - Allocate one thread per neighbor point to compute
  - This optimization requires inter-thread data communication, which is possible in CUDA, but not in OpenACC



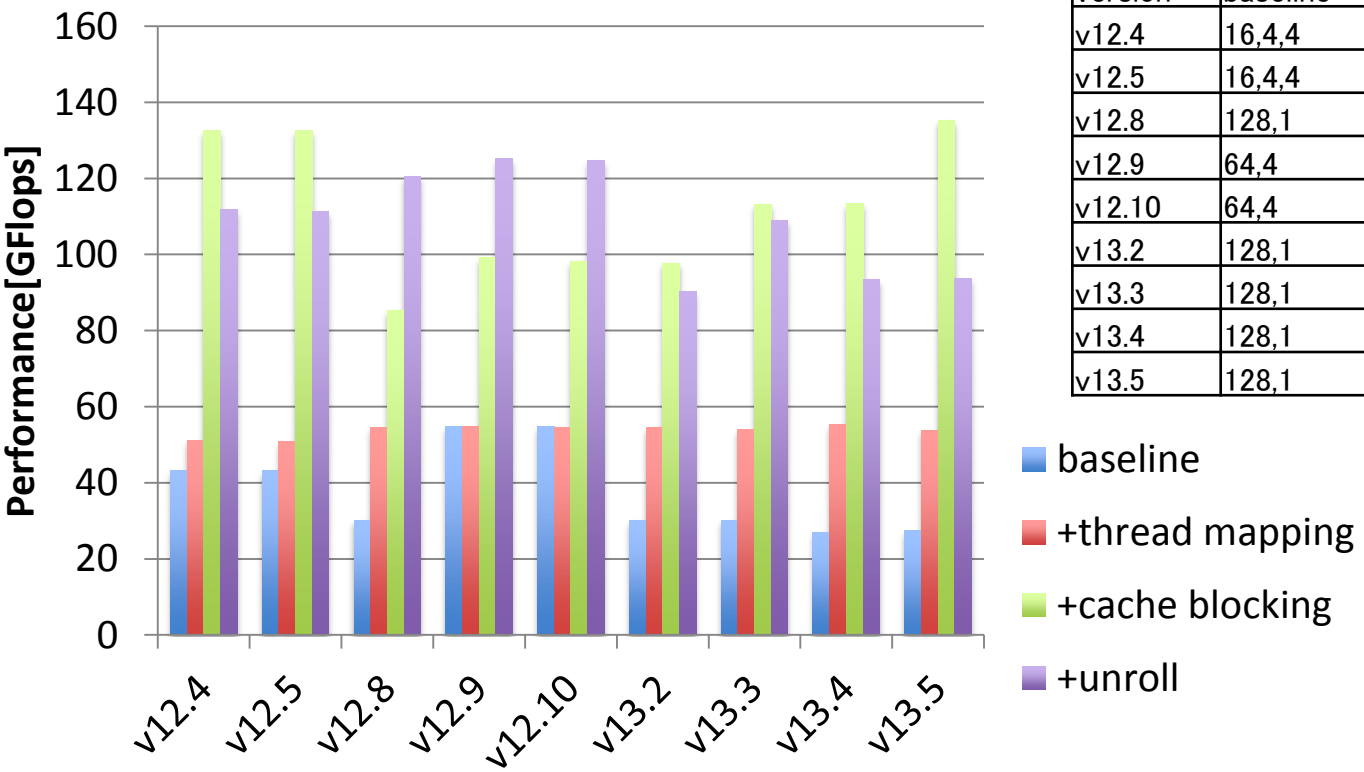
Optimizations: Fine-grained parallelization in Time Integration phase

- Optimizing Time Integration phase by using shared memory
  - The Time Integration phase computes the six neighbor cells serially to update the central cell
  - Allocate one thread per neighbor point to compute
  - This optimization requires inter-thread data communication, which is possible in CUDA, but not in OpenACC



Difference of the performance among the compiler versions
Matrix Multiplication

(see matmul_f.f90)
Problem size : 2048^2
double precision

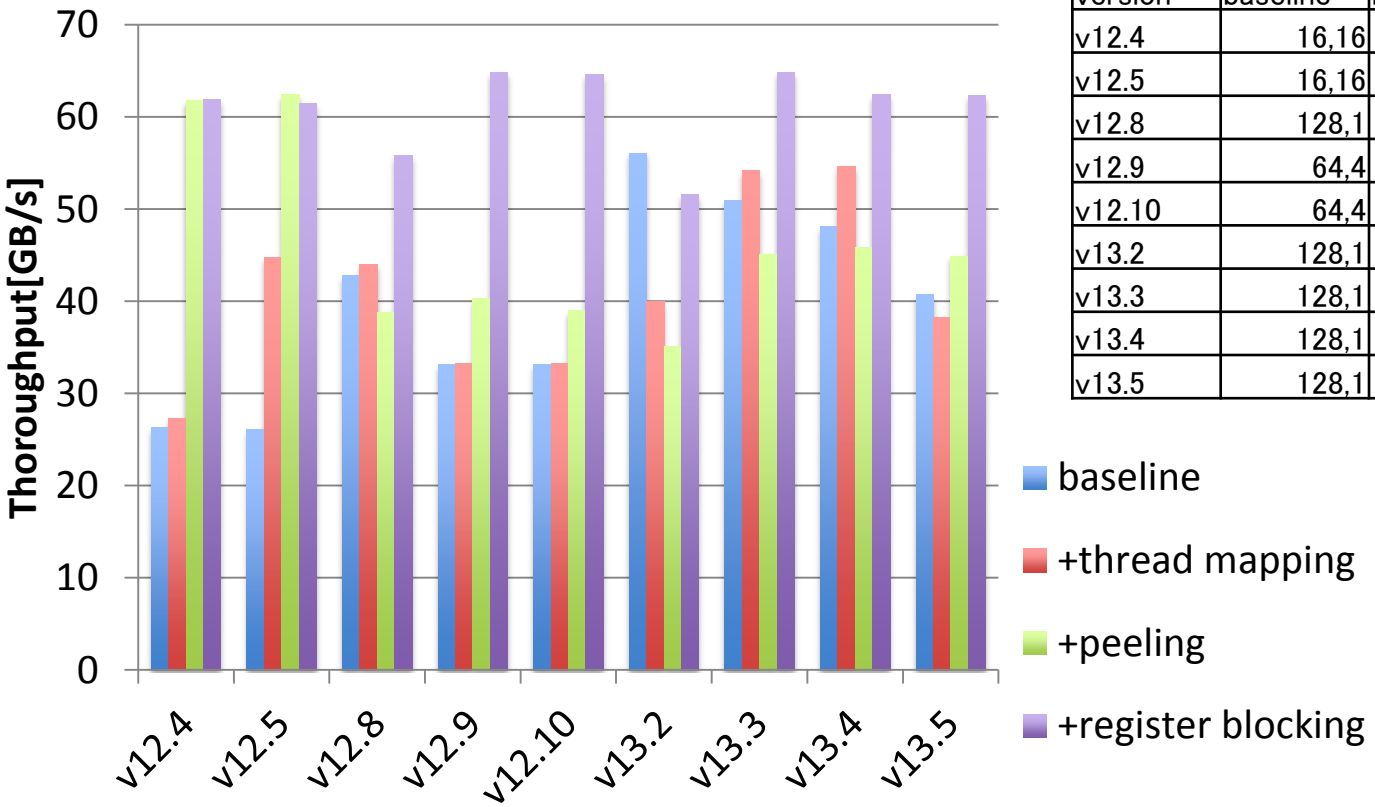


The best thread mapping parameter
(baseline : chosen by compiler)

Compiler version	baseline	+thread mapping	+peeling	+register blocking
v12.4	16,4,4	64,1	128,2	128,2
v12.5	16,4,4	32,8	128,2	128,2
v12.8	128,1	128,1	128,2	128,1
v12.9	64,4	128,2	32,4	128,1
v12.10	64,4	128,1	64,2	64,2
v13.2	128,1	64,2	128,1	128,8
v13.3	128,1	64,2	128,2	64,2
v13.4	128,1	64,2	128,2	64,2
v13.5	128,1	32,4	128,2	64,2

Difference of the performance among the compiler versions
7-point stencil

(see stencil-pgi-acc.f90)
Problem size : 128^3
single precision



The best thread mapping parameter
(baseline : chosen by compiler)

Compiler version	baseline	+thread mapping	+cache blocking	+unroll
v12.4	16,16	256,4	32,16	32,16
v12.5	16,16	256,4	32,16	32,16
v12.8	128,1	128,8	128,4	32,16
v12.9	64,4	64,4	32,16	32,16
v12.10	64,4	64,4	32,16	32,16
v13.2	128,1	256,4	128,4	32,16
v13.3	128,1	256,4	128,4	32,16
v13.4	128,1	128,4	128,4	32,16
v13.5	128,1	128,4	128,4	32,16