スケジュール

- 09：30 - 10：00　受付
- 10：00 - 12：00　Reedbush-Hログイン、GPU入門
- 13：30 - 15：00　OpenACC入門
- 15：15 - 16：45　OpenACC最適化入門と演習
- 17：00 - 18：00　OpenACCの活用（CUDA連携とライブラリの活用）

OpenACCの活用

CUDA連携とライブラリの活用

OpenACCとCUDAの相互運用

- OpenACC
  - 簡単にGPUプログラムが作成できる、それなりの性能が得られる
  - 様々な環境で利用できる
- CUDA
  - OpenACCと比べると使用が大変だが、より高い性能が期待できる
  - （基本的に）NVIDIA GPU専用

- OpenACCプログラムの一部をCUDA化することで簡単さと高性能を両立できるのでは？
  - 誰かが書いた（公開している）CUDAプログラム（関数・ライブラリ）を自分のOpenACCプログラムから使わせてもらう
  - 自分が作成したOpenACCプログラムの一部をCUDAで高速化する

（ほんの少しだけ）CUDAの概要

- NVIDIA GPUのハードウェアアーキテクチャに対応した言語
  - 適切な記述をすることでNVIDIA GPUの性能を引き出せる可能性がある
  - C言語版はCUDA CとしてNVIDIAが提供、開発環境は無償
  - Fortran版はCUDA FortranとしてPGIが提供、PGIコンパイラが必要、無料プランも有
- 言語拡張仕様・コンパイラ・ライブラリを提供
  - 言語拡張：GPUカーネルや使用するメモリの種類を明示する記述
  - コンパイラ：nvcc
  - ライブラリ：数値計算ライブラリや機械学習ライブラリなど
- GPUカーネルを動かす単位は関数
  - 関数単位で並列度を指定してGPUカーネルを起動
  - グローバルメモリに置かれたデータのみが関数間で引き継がれる

CUDA C プログラムの例

cuda_c.cu

```
__global__ void gpukernel
(int N, float *C, float *A, float *B)
{
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  if(id<N)C[id] += A[id] * B[id];
}
```

GPU上の各計算コアが行う処理を
__global__ void関数として記述
（右上へ続く）

```
int main(int argc, char **argv)
{
 int i, N;
 float *A, *B, *C;
 float *dA, *dB, *dC;

 N = 128;
 A = (float*)malloc(sizeof(float)*N);
 B = (float*)malloc(sizeof(float)*N);
 C = (float*)malloc(sizeof(float)*N);
 for(i=0;i<N;i++){
  C[i] = 0.0f;    B[i] = 2.0f;
  A[i] = (float)(i+1)/(float)(N);
 }

 cudaMalloc((void**)&dA, sizeof(float)*N);
 cudaMalloc((void**)&dB, sizeof(float)*N);
 cudaMalloc((void**)&dC, sizeof(float)*N);

 cudaMemcpy(dA, A, sizeof(float)*N, cudaMemcpyHostToDevice);
 cudaMemcpy(dB, B, sizeof(float)*N, cudaMemcpyHostToDevice);
 cudaMemcpy(dC, C, sizeof(float)*N, cudaMemcpyHostToDevice);

 dim3 grids;
 dim3 blocks;
 grids = dim3(4, 1, 1);
 blocks = dim3(64, 1 ,1);
 gpukernel<<<grids,blocks>>>(N, dC, dA, dB);

 cudaMemcpy(C, dC, sizeof(float)*N, cudaMemcpyDeviceToHost);

 cudaFree(dA); cudaFree(dB); cudaFree(dC);
 free(A); free(B); free(C);
 return 0;
}
```

※並列度はGPUカーネル呼び出し時に個別に指定

典型的CUDA Cプログラムの処理の流れ
1. GPU上のメモリを確保
2. CPUからGPUへのデータ転送
3. 特殊な記法でGPUカーネルを起動
4. GPUからCPUへのデータ転送
5. GPU上のメモリを破棄

（1,2,4,5は用意されている専用の関数によって行う）

nvccでコンパイルして実行する

```
$ nvcc cuda_c.cu; ./a.out
```

CUDA Fortran プログラムの例

cuda_f.cuf

```
module gpukernel
contains
 attributes(global) subroutine gpukernel(N, C, A, B)
  integer,value :: ID, N
  real(kind=4), device, dimension(N), intent(in) :: A, B
  real(kind=4), device, dimension(N), intent(inout) :: C
  ID = (blockIdx%x-1)*blockDim%x + threadIdx%x
  if(ID.le.N)then
    C(ID) = C(ID) + A(ID) * B(ID)
  endif
 end subroutine gpukernel
end module gpukernel
```

GPU上の各計算コアが行う処理を
attribute(global)サブルーチン関数
として記述　　（右上へ続く）

```
program main
 use cudafor
 use gpukernel
 implicit none

 real(4), allocatable, dimension(:) :: A, B, C
 real(4), allocatable, dimension(:),device :: dA, dB, dC
 integer :: I, N
 type(dim3) :: dimGrid, dimBlock

 N = 128
 allocate(A(N), B(N), C(N))
 allocate(dA(N), dB(N), dC(N))

 C = 0.0;  B = 2.0
 do I=1, N
  A(I) = real(I)/real(N)
 enddo

 dA = A;   dB = B;   dC = C

 dimGrid = dim3(2,1,1)
 dimBlock = dim3(64,1,1)
 call gpukernel<<<dimGrid, dimBlock>>>(N, dC, dA, dB)

 C = dC

 deallocate(dA, dB, dC);  deallocate(A, B, C)
end program main
```

cudaforモジュールを使う

CPUからGPUへの
データ転送

GPUからCPUへのデータ転送

- CUDA Fortranプログラムの方が少し簡単
- CPU上の配列とGPU上の配列を明示的に宣言できるため、データのコピーが自動的に行われる

-Mcudaを指定してコンパイルして実行する

```
$ pgf90 -Mcuda cuda_f.cuf; ./a.out
```

OpenACCとCUDAの「連携」？

- そもそも、OpenACC指示文を含むソースコードとCUDA記法を含むソースコードを分けておいて個別にコンパイルし、１つのプログラムにまとめて使うこと自体は可能
- 単純にOpenACCとCUDAのソース（関数）を組み合わせて利用した場合、OpenACCとCUDAを行き来する度にデータのコピーが必要になってしまい性能低下要因となる
- OpenACCによるデータ送受信（data指示文による処理）とCUDAにおけるデータの扱い（接頭辞やAPIによる指定と処理）の橋渡し役が必要

使用イメージ

- C言語ベースの場合

```
#pragma acc enter data copyin(…)

#pragma acc kernels
  …
#pragma acc end kernels

  ……

#pragma acc exit data copyout(…)
```

- Fortranベースの場合

```
!$acc enter data copyin(…)

!$acc kernels
  …
!$acc end kernels

  ……

!$acc exit data copyout(…)
```

← OpenACCによる計算 →

← CUDAによる計算 →

➢ 一つのdata指示文の中ではOpenACCカーネルとCUDAカーネルでデータを共有させたい

解決策：host_data と use_device 指示文

- GPU上に存在するデータ（配列）の存在を伝えるための指示文を用いる

- 使い方

```
acc host_data use_device(対象とする配列名)
```

OpenACCからCUDA Cカーネルを使う：GPUカーネルの用意

cudakernel.cu

```
__global__ void gpukernel(int N, float *C, float *A, float *B)
{
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  C[id] += A[id] * B[id];
}
```

- GPUカーネルはCUDA Cのみの場合と同様

```
extern "C" void gpukernel_wrapper(int N, float *C, float *A, float *B)
{
  dim3 grids;
  dim3 blocks;
  grids = dim3(2, 1, 1);
  blocks = dim3(64, 1 ,1);
  gpukernel<<<grids,blocks>>>(N, C, A, B);
}
```

- GPUカーネルを起動する関数
- C++ではなくCから使う場合は「extern “C”」が必要
- CPU-GPU間のデータ転送については何も記述しなくて良い

OpenACCからCUDA Cカーネルを使う：OpenACC側の記述

acc_main.c

```
#include <stdio.h>

extern void gpukernel_wrapper(int N, float *C, float *A, float *B);

int main(int argc, char **argv){
  int i, N;
  float *A, *B, *C;
  N = 128;
  A = (float*)malloc(sizeof(float)*N);
  B = (float*)malloc(sizeof(float)*N);
  C = (float*)malloc(sizeof(float)*N);
  for(i=0;i<N;i++){
    C[i] = 0.0f;    B[i] = 2.0f;
    A[i] = (float)(i+1)/(float)(N);
  }
#pragma acc enter data copyin(A[0:N],B[0:N],C[0:N])
#pragma acc kernels present(A,B,C)
#pragma acc loop independent
    for(i=0; i<N; i++){
      C[i] += A[i] * B[i];
    }
#pragma acc host_data use_device(A,B,C)
    {
      gpukernel_wrapper(N, C, A, B);
    }
#pragma acc exit data copyout(C[0:N])
  free(A); free(B); free(C);
  return 0;
}
```

OpenACC

CUDA

「GPUカーネルを起動する関数」を実行することでGPUを動かしている

OpenACCからCUDA Cカーネルを使う：コンパイルとリンク

- CUDA C部分はnvccでコンパイルする必要がある点に注意
- 最低限必要な引数指定などの例

```
nvcc -c cudakernel.cu
pgcc -acc -c acc_main.c
pgcc -Mcuda -acc cudakernel.o
acc_main.o
```

-Mcudaと-acc両方の指定が必要なところに注意する

- 最適化オプションなどを加えた例

```
nvcc -O2 -gencode arch=compute_60,code=¥"sm_60,compute_60¥“ -c cudakernel.cu
pgcc -acc -O2 -ta=tesla,cc60 -Minfo -c acc_main.c
pgcc -Mcuda -acc cudakernel.o acc_main.o
```

- 実行

```
./a.out
```

OpenACCからCUDA Fortranカーネルを用いる：GPUカーネルの用意

cudakernel.cuf

```
module gpukernel
contains
  attributes(global) subroutine gpukernel(N, C, A, B)
    integer,value :: ID, N
    real(kind=4), device, dimension(N), intent(in) :: A, B
    real(kind=4), device, dimension(N), intent(inout) :: C
    ID = (blockIdx%x-1)*blockDim%x + threadIdx%x
    if(ID.le.N)then
       C(ID) = C(ID) + A(ID) * B(ID)
    endif
  end subroutine gpukernel

  subroutine gpukernel_wrapper(N, C, A, B)
    use cudafor
    integer, intent(in) :: N
    real(kind=4), device, dimension(N), intent(in) :: A, B
    real(kind=4), device, dimension(N), intent(inout) :: C
    type(dim3) :: dimGrid, dimBlock
    dimGrid = dim3(2,1,1)
    dimBlock = dim3(64,1,1)
    call gpukernel<<<dimGrid, dimBlock>>>(N, C, A,
  end subroutine gpukernel_wrapper
end module
```

GPUカーネルはCUDA Fortranと同様

どちらもdeviceによる指定は必要

- GPUカーネルを起動する関数
- CPU-GPU間のデータ転送については何も記述していない

OpenACCからCUDA Fortranカーネルを用いる：OpenACC側の記述

acc_main.f90

```
program main
  use gpukernel
  implicit none

  real(4), allocatable, dimension(:) :: A, B, C
  integer :: I, N

  N = 128
  allocate(A(N), B(N), C(N))

  C = 0.0;  B = 2.0
  do I=1, N
     A(I) = real(I)/real(N)
  enddo
!$acc enter data copyin(A(1:N), B(1:N), C(1:N))
!$acc kernels
!$acc loop
  do I=1, N
     C(I) = C(I) + A(I) * B(I)
  enddo
!$acc end kernels
!$acc host_data use_device(A, B, C)
  call gpukernel_wrapper(N, C, A, B)
!$acc end host_data
!$acc exit data copyout(C(1:N))

  deallocate(A, B, C)
end program main
```

OpenACC

CUDA

「GPUカーネルを起動する関数」を実行することでGPUを動かしている

OpenACCからCUDA Fortranカーネルを用いる：コンパイルとリンク

- nvccは使わない
- 最低限必要な引数指定などの例

```
pgf90 -Mcuda -c cudakernel.cuf
pgf90 -acc -c acc_main.f90
pgf90 -Mcuda -acc cudakernel.o acc_main.o
```

- 最適化オプションなどを加えた例

```
pgf90 -Mcuda=cc20 -O2 -Minfo -c cudakernel.cuf
pgf90 -acc -O2 -ta=tesla,cc60 -Minfo -c acc_main.f90
pgf90 -Mcuda -acc cudakernel.o acc_main.o
```

- 実行

```
./a.out
```

OpenACCとCUDAの連携：逆バージョン

- CUDA C/Fortranで書かれたプログラムに対してOpenACCコードを追加したいこともあるかもしれない
- CUDA C/Fortranプログラムによって用意された配列をOpenACCカーネルから利用する必要がある
- deviceptr節を使用する
  - OpenACC側では配列の確保や転送を書く必要がない

CUDA C + OpenACC

cuda_main.cu

```
extern "C" void acckernel(int N, float *A, float *B, float *C);

__global__ void gpukernel(int N, float *C, float *A, float *B)
{
  通常のCUDAカーネル記述（省略）
}

main関数内（通常のCUDA C記述、メモリ解放は省略）
  A = (float*)malloc(sizeof(float)*N);
  B = (float*)malloc(sizeof(float)*N);
  C = (float*)malloc(sizeof(float)*N);
  cudaMalloc((void**)&dA, sizeof(float)*N);
  cudaMalloc((void**)&dB, sizeof(float)*N);
  cudaMalloc((void**)&dC, sizeof(float)*N);
  cudaMemcpy(dA, A, sizeof(float)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(dB, B, sizeof(float)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(dC, C, sizeof(float)*N, cudaMemcpyHostToDevice);
  gpukernel<<<grids,blocks>>>(N, dC, dA, dB);

  acckernel(N, dC, dA, dB);

  cudaMemcpy(C, dC, sizeof(float)*N, cudaMemcpyDeviceToHost);
```

acckernel.c

```
void acckernel
(int N, float *C, float *A, float *B)
{
#pragma acc kernels deviceptr(A,B,C)
#pragma acc loop independent
  for(int i=0; i<N; i++){
    C[i] += A[i] * B[i];
  }
}
```

cudaMallocで確保した配列を渡し、deviceptrで受ける

```
pgcc -acc -O2 -Minfo -ta=tesla,cc60 -c acckernel.c
nvcc -O2 -gencode arch=compute_60,code=¥"sm_60,compute_60¥" -c cuda_main.cu
pgcc -Mcuda=cc20 -acc -o hybrid2 acckernel.o cuda_main.o
```

CUDA Fortran + OpenACC

cuda_main.cuf

```
module cudakernel
contains
  attributes(global) subroutine cudakernel(N, C, A, B)
    integer,value :: ID, N
    real(4), device :: A(:), B(:), C(:)
    ID = (blockIdx%x-1)*blockDim%x + threadIdx%x
    ※GPUカーネルの記述は省略
  end subroutine cudakernel
end module cudakernel

program main
  use cudafor
  use cudakernel
  use acckernel
  implicit none
  real(4), allocatable, dimension(:) :: A, B, C
  real(4), allocatable,device :: dA(:), dB(:), dC(:)
  ※allocateと初期化は省略、解放も省略
  dA = A;   dB = B;   dC = C
  dimGrid = dim3(2,1,1)
  dimBlock = dim3(64,1,1)
  call cudakernel<<<dimGrid, dimBlock>>>(N, dC, dA, dB)

  call acckernel(N, dC, dA, dB)

  C = dC
```

acckernel.f90

```
module acckernel
contains
  subroutine acckernel(N, C, A, B)
    integer :: I, N
    real(4), device :: A(:), B(:), C(:)
!$acc kernels deviceptr(A,B,C)
!$acc loop
    do I=1, N
       C(I) = C(I) + A(I) * B(I)
    enddo
    !$acc end kernels
  end subroutine acckernel
end module acckernel
```

デバイス用に確保した配列を渡し、deviceptrで受ける

「,device」の為にacckernel.f90にも-Mcudaオプションが必要

```
pgf90 -Mcuda -acc -ta=tesla,cc60 -O2 -Minfo -c acckernel.f90
pgf90 -Mcuda=cc60 -O2 -Minfo -c cuda_main.cuf
pgf90 -Mcuda=cc60 -acc -o hybrid2 cuda_main.o acckernel.o
```

OpenACCとCUDA用ライブラリの連携

- CUDA用に用意されているライブラリをOpenACCから利用したい
- グローバルメモリにデータを配置した状態から関数を呼び出すだけのものであればhost_data / use_deviceを利用することで実現が可能
  - ライブラリの提供する専用関数で値を設定するようなものは困難

OpenACC + CUBLAS / C

※メモリ解放などの処理は省略

```
  cudaMalloc((void**)&dA, sizeof(float)*N*N);
  cudaMalloc((void**)&dB, sizeof(float)*N*N);
  cudaMalloc((void**)&dC, sizeof(float)*N*N);
  cudaMemcpy(dA, A, sizeof(float)*N*N, cudaMemcpyHostToDevice);
  cudaMemcpy(dB, B, sizeof(float)*N*N, cudaMemcpyHostToDevice);
  cudaMemcpy(dC, C, sizeof(float)*N*N, cudaMemcpyHostToDevice);
  cublasCreate(&handle);
  cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N,
    &alpha, dA, N, dB, N, &beta, dC, N);
  cudaMemcpy(C, dC, sizeof(float)*N*N, cudaMemcpyDeviceToHost);
```

上：CUDA C + CUBLAS

cudaMallocとcudaMemcpyでデータを準備し、GPU側の配列を引数に与えてcublas関数を実行

コンパイル例：nvcc -O3 -lcublas cublas.c

下：OpenACC + CUBLAS

data指示文でデータ転送、host_data/use_deviceで指示をしてからcublas関数を実行

コンパイル例：pgcc -Mcuda -acc -O3 -ta=tesla,cc60 -lcublas hybrid.c

```
  cublasCreate(&handle);
#pragma acc enter data copyin(A[0:N*N], B[0:N*N], C[0:N*N])
#pragma acc host_data use_device(A, B, C)
  {
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N,
      &alpha, A, N, B, N, &beta, C, N);
  }
#pragma acc exit data copyout(C[0:N*N])
```

OpenACC + CUBLAS / Fortran

※メモリ解放などの処理は省略

```
  use cublas
  real(4), allocatable, dimension(:,:) :: A, B, C
  real(4), allocatable, dimension(:,:),device :: dA, dB, dC
  allocate(A(N,N), B(N,N), C(N,N))
  allocate(dA(N,N), dB(N,N), dC(N,N))
  dA = A;   dB = B;   dC = C
  call cublassgemm('n','n',N,N,N,alpha,dA,N,dB,N,beta,dC,N)
  C = dC
```

上：CUDA Fortran + CUBLAS

CUDA Fortranの書き方でGPU上のメモリを準備し、GPU側の配列を引数に与えてcublas関数を実行

コンパイル例：pgf90 -O3 cublas.cuf

どちらの実装もC版と異なり-lcublas指定がないが、use cublasが入っているためライブラリがリンクされる

下：OpenACC + CUBLAS

data指示文でデータ転送、host_data/use_deviceで指示をしてからcublas関数を実行

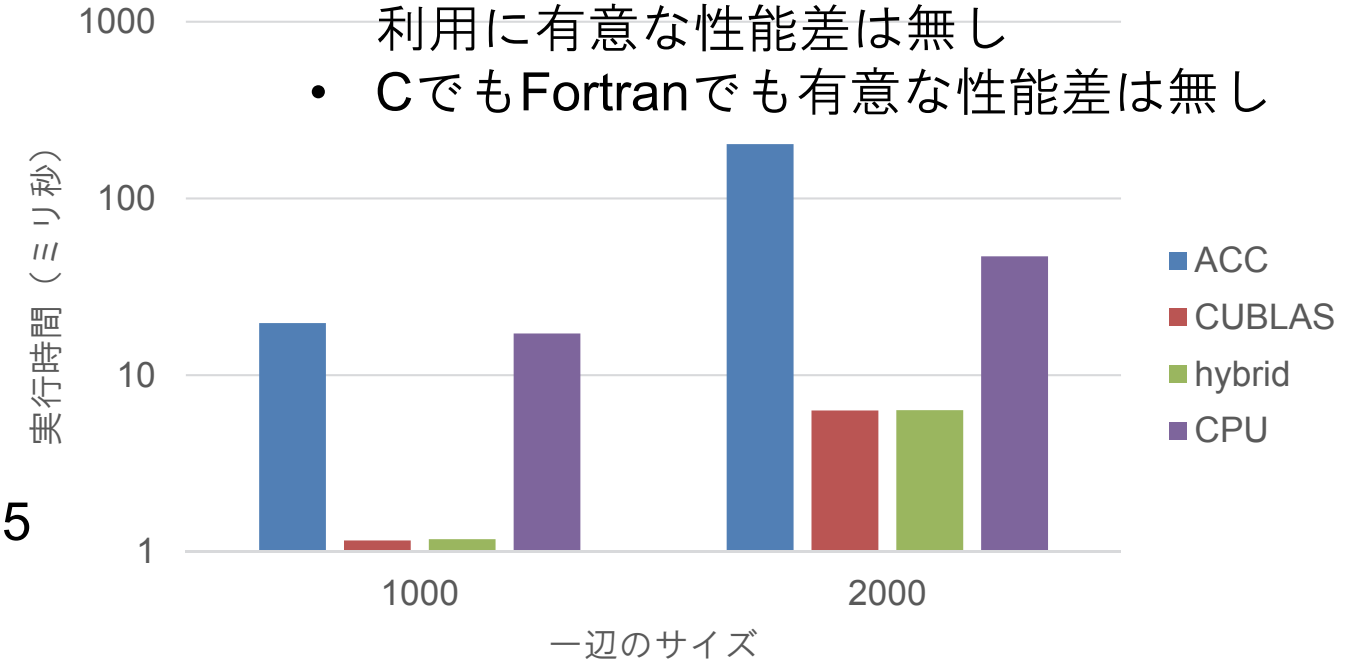
コンパイル例：pgf90 -Mcuda -acc -O3 -ta=tesla,cc60 hybrid.f90

```
  use cublas
  real(4), allocatable, dimension(:,:) :: A, B, C
  allocate(A(N,N), B(N,N), C(N,N))
!$acc enter data copyin(A(1:N,1:N), B(1:N,1:N), C(1:N,1:N))
!$acc host_data use_device(A, B, C)
  call cublassgemm('n','n',N,N,N,alpha,A,N,B,N,beta,C,N)
!$acc end host_data
!$acc exit data copyout(C(1:N,1:N))
```

性能比較

- OpenACCによる単純な行列積実装とCUBLASによる高速な行列積実装の性能を比較した
  - 正方行列同士の単純な行列積計算
  - ACC：OpenACCによる単純な実装（外部2重ループの並列化）
  - CUBLAS：cublassgemmを利用
  - hybrid：OpenACCからcublassgemmを呼び出し
  - CPU : MKL sgemm

- CUBLASの直接利用とOpenACCからの利用に有意な性能差は無し
- CでもFortranでも有意な性能差は無し



CPU : Xeon E5-2680 v2
  icc 16.0.3, mkl=parallel
GPU : Tesla K40c
  pgcc 16.9, -O3 -ta=tesla,cc35

まとめ

- OpenACCとCUDAの連携、OpenACCと（CUDA向け）ライブラリの連携について紹介した
- 連携させる方法自体はあまり難しくはないため、使い勝手と性能を考えて適切な実装方法を選ぶのが良い
  - 対象とする問題にあった高性能な実装やライブラリが存在する場合には積極的に活用するべき