筑波大学計算科学研究センター GPUプログラミング講習会

2012/12/21(金) 第4回

PGI CUDA Fortran によるGPUプログラミング

本日の内容

- PGI CUDA Fortranの解説
  - PGIコンパイラの概要
  - コンパイラの使い方
  - CUDA Fortranの書き方
  - CUDA C/C++との対比
- HA-PACS
  - HA-PACSにおける使い方
  - HA-PACS上で簡単な実演
- 質疑応答

PGIコンパイラ

- Portland Groupが開発しているコンパイラ+デバッガ+プロファイラなどを含む総合コンパイラスイート
  - C, C++, Fortran
- アクセラレータ対応が進んでいる
  - CUDA Fortran
  - OpenACC
  - 独自ディレクティブ (!$CUF)

PGI + OpenACC

- OpenACC
  - PGI 12.6からOpenACC 1.0をサポート
    - PGI 12.3からベータ品質でサポート
  - HA-PACSでは、次回メンテナンスでPGI 12.10が導入され、使えるようになる予定

```
!$acc kernels
do i = 1, N
  do j = 1, N
    do k = 1, N
      c(j, i) = c(j, i) + a(k, i) * b(j, k)
    end do
  end do
end do
!$acc end kernels
```

PGI + CUDA

- PGI CUDA Fortran
  - FortranをベースにCUDAに対応させるために、仕様が拡張されている
  - CUDA APIのFortranラッパー
  - CUDA周辺ライブラリのラッパー

PGI CUDA Fortran

- CUDA C/C++で書かれたコードをFortranに変換するイメージ
  - APIやコードの書き方がほぼ同一
    - CUDA C/C++とFortranを両方知る人なら、簡単に利用できる
  - 一方で、NVIDIA GPUアーキテクチャやCUDAのプログラミング・実行モデルの知識が必須
    - あまり必要でないのがOpenACC
  - CUDAドキュメントはCUDA C/C++で書かれているので、C言語の知識も多少必要になる

PGI CUDA Fortran

- CUDA C/C++とPGI CUDA Frotranの対応例
  - CUDA APIの名前は変化しない
    - cudaMemcpy → cudaMemcpy
    - cudaThreadSynchronize → cudaThreadSynchronize
  - 素のFortranにない記法は、近い形で導入される
    - __global__ → attributes(global)
    - (カーネル起動の) <<< >>> → <<< >>>

PGI CUDA Fortran

- CUDA C/C++とPGI CUDA Frotranの対応例

```
CUDA C/C++:
  my_kernel<<<N, M>>>(a, b, c, d);
```

```
CUDA Fortran:
  call my_kernel<<<N, M>>>(a, b, c, d)
```

PGI CUDA Fortran

- 内部的には、GPUカーネル部分のFortranコードをCコードに変換しnvccを起動している
  - GPU関連のコンパイルの実質的な作業はnvccが行う
  - 機能差や性能差は出にくいと考えられる

PGI CUDA Fortran

- PGI CUDA FortranはCUDAを後追いで対応している
  - CUDA C/C++で行える全てがCUDA Fortranで使えるわけではない
  - CUDAで追加された新機能がCUDA Fortranで使えるようになるまでタイムラグがある
  - 同様にToolkitやDriverの新版に対応することにもタイムラグがある
  - 現在の最新版(12.10)でもCUDA5はまだ未対応

コンパイラの使い方

- コンパイラのコマンド名はpgfortran
  - -c，-D，-I，-l，-L，-o などはgccと同様
  - 最適化をしたい場合は -fast を付ける
  - 自由形式を明示的に指定する場合 -Mfree
  - 固定形式を明示的に指定する場合は -Mfixed
  - プリプロセスを行いたい場合は -Mpreprocess
  - CUDA関係のオプションは -Mcuda を使用する
    - 詳細は後述

コンパイラの使い方

- PGI CUDA Fortranを示す拡張子は
  .cuf or .CUF
  - .cuf は自由形式(free form)・プリプロセスなし
  - .CUF は自由形式・プリプロセスあり
  - 明示的にオプションを与えれば変更可能
- .Fや.f90のままでも明示的にCUDA Fortranを有効にすればコンパイル可能
  - -Mcuda

コンパイラの使い方

- -McudaでCUDA関係のオプションを詳細に指定できる
- -Mcuda
  - CUDA Fortran拡張の有効化
- -Mcuda=xxx,yyy,zzz
  - CUDA Fortran拡張の有効化に加えて
  - サブオプション xxx,yyy,zzz を指定

コンパイラの使い方

- -Mcudaのサブオプションで最も重要なものはCompute Capability(以下CC)の指定
  - 実行対象とするGPUのCCを指定した方が良い
  - 性能・機能面でGPUの機能をフルに引き出せる
- HA-PACSのGPUはM2090 (CC 2.0)
  - `-Mcuda=cc20` と指定する

コンパイラの使い方 (まとめ)

```
$ pgfortran -fast -Mcuda=cc20 -o test test.cuf
```

- 最適化を有効に (-fast)
- CUDA Fortranを有効にして (-Mcuda)
- CC20向けのコード生成 (-Mcuda=cc20)

CUDA Fortranを用いるプログラミング

- GPUメモリの確保
- CPU～GPU間のデータ転送
- カーネルの記述・起動など

CUDA Fortranを用いるプログラミング

- CUDAのAPIは cudafor モジュールに定義されている
  - 以降の話は全てcudaforモジュールをuseしているものとする
  - CUBLASはcublas、CUFFTはcufftモジュールを使う

```
use cudafor
```

GPUのメモリ確保

- 配列をGPUのGlobal Memoryに確保したい場合は、配列に対してdevice属性を付与する
  - attributes(device)
  - 単に device でも可
- ライフタイムはFortranの仕様に倣う
  - 関数内ならば、関数を抜けるまで
  - モジュール内ならば、アプリケーション終了まで

```
subroutine f()
  double precision, device :: array(100)
end subroutine f
```

GPUのメモリ確保

- device配列はallocatableにすることも可能
  - 確保・開放にはallocate, deallocate文を使用する

```
subroutine f()
  double precision, device, allocatable :: a(:)

  allocate(a(100))
  ! do something…
  deallocate(a)
end subroutine f
```

Pinnedメモリの確保

- pinned属性を用いると、pinnedメモリを確保できる
  - allocatable属性が必須
  - 確保・開放にはallocate, deallocate文を用いる

```
subroutine f()
  double precision, pinned, allocatable :: a(:)

  allocate(a(100))
  ! do something…
  deallocate(a)
end subroutine f
```

CPU～GPU間のメモリ転送

- CUDA Fortranでは2つの方法がある
  - 代入文を用いる転送
    - 記述は非常に簡単で、できる事は多い
    - 非同期転送は記述できない
  - CUDA APIを用いる転送
    - CUDA C/C++で用いる手法と等価
    - 非同期転送などが利用でき高機能だが
    - 記述は代入文を用いる場合よりも複雑

CPU～GPU間のメモリ転送

- 代入文を記述すると自動的に転送される
  - CPU→GPU, GPU→CPU, GPU→GPU どれでも可

```
double precision :: host(100)
double precision, device :: dev(100)

dev = host
```

CPU～GPU間のメモリ転送

- 配列全体ではなく、一部分のみの転送も可能

```
double precision :: host(10, 10)
double precision, device :: dev(10, 10)

dev(2:5, 3:8) = host(2:5, 3:8)
```

CPU～GPU間のメモリ転送

- 定数や(配列でない)変数を代入することで、memsetの様な動作ができる

```
double precision, device :: dev(10, 10)
double precision :: initial_value

dev = 0.0d0

initial_value = 3.14d0
dev = initial_value
```

CPU～GPU間のメモリ転送?

- 単なる転送だけでなく、演算も可能
- ただし、制限が厳しい
  - 式の右辺中に表われるデバイス配列は1つまで
  - 加算などの算術演算はホストで行われる
  - テンポラリ配列が自動的に生成されることがある

```
double precision, device :: dev(10)
double precision :: host(10)

host = host + 2.0d0 * dev
```

CPU～GPU間のメモリ転送

- CUDA APIを用いた転送も可能
  - 代入文によるデータ転送では非同期転送ができなかったが、CUDA APIを使えば可能
  - API名や引数はCUDA C/C++と同じだが、転送する要素数の指定方法が異るので、プログラムを移植する場合は注意が必要

CPU～GPU間のメモリ転送

- 転送範囲の指定方法
  - C/C++: 転送するバイト数
  - Fortran: 転送する要素数

倍精度で要素数100の配列を転送する例:

C:

```
cudaMemcpy(dest, src, 100 * sizeof(double),
    cudaMemcpyHostToDevice);
```

Fortran:

```
double precision, device :: dest(100)
double precision :: src(100)
cudaMemcpy(dest, src, 100,
    cudaMemcpyHostToDevice)
```

カーネルの書き方

- subroutineとfunctionにattributesを付ける
  - __global__ → attributes(global)
    - CUDA C/C++で返り値型がvoidしか書けない制限と同様に、subroutineにしか付与できない
  - __device__ → attributes(device)
  - __host__ → attributes(host)

```
attributes(global) subroutine f()
end subroutine f
```

カーネルの書き方

- 引数のデータはGPUから参照できる位置になければならない
  - device変数, device配列, 即値
- Fortranの引数は参照渡しが基本のため、注意が必要

カーネルの書き方

- CUDA C/C++であれば、整数値などは引数で直接渡せる
  - プログラム上では、GPUへ転送しなくても良い

```
__global__ void f(int n) {
}

void g() {
  int n = 100;
  f<<<1, 1>>>(n);
}
```

カーネルの書き方

- CUDA Fortranでは、整数nをカーネルに引数で渡す場合、右の様には書けない
  - fの引数nはGPU上のメモリ空間に存在する必要がある
  - ただし、コンパイルは通ってしまう

```
attributes(global) &
subroutine f(n)
  integer :: n
end subroutine f

subroutine g
  integer :: n = 100
  call f<<<1, 1>>>(n)
end subroutine g
```

カーネルの書き方

- 回避するには、手動でGPU側に転送するか、引数にvalue属性を付ける
  - CUDA C/C++と同様の扱いとなる
  - Fermi以降のGPUならばConstant Memoryに入る

```
attributes(global) subroutine f(n)
  integer, value :: n
end subroutine f

subroutine g
  integer :: n = 100
  call f<<<1, 1>>>(n)
end subroutine g
```

カーネルの書き方

- shared memoryを使う場合は、変数にshared属性を付ける

```
attributes(global) subroutine f(n)
  double precision, shared :: smem(100)
end subroutine f
```

カーネルの書き方

- threadIdx, blockDimなどの定数は同じ名前で参照可能
  - ただし、数値の範囲はCUDA C/C++が0始点なのに対して、CUDA Fortranは1始点

例:

CUDA C/C++	CUDA Fortran
threadIdx.x	threadIdx%x - 1
blockIdx.y	blockIdx%y - 1
gridDim.z	gridDim%z

カーネルの書き方

- 組み込み関数についてはやや特殊
  - CUDAが提供するものについては、ほとんど使える
  - ただし、例外もあり、例えば __syncthreads はないので syncthreads を呼ぶ
  - (“__”から始まるものがない、という話でもない)

```
__syncthreads();
```

```
call syncthreads()
```

カーネルの書き方

- 組み込み関数についてはやや特殊
  - Fortranが提供するものも、一部使える
  - 主に数学関係や型変換関係
  - 例えば min, max, abs など

HA-PACSにおける使い方

- PGI関連のモジュールをロードする
- $ module load
  - pgi/12.2
  - pgi/cuda_fortran_4.1
  - mvapich2/1.8_pgi
    - medium
    - nocuda
    - など派生モジュール多数

HA-PACSにおける使い方

- pgi/12.2
  - pgf77, pgfortran, pgcc などにPATHが通る
  - 環境変数 CC, FC などが設定される
    - FC = pgfortran
    - CC = pgcc
- pgi/cuda_fortran_4.1
  - PGIコンパイラ内包のCUDA 4.1環境
  - cuda/4.1.28モジュールもほぼ必須

HA-PACSにおける使い方

- mvapich2/1.8_pgi
  - HA-PACSでは、MVAPICH2のみサポート
  - mpif77, mpif90などにPATHが通る
  - mediumメモリモデルを使う場合は mvapich2/1.8_pgi_mediumモジュールを使用する

HA-PACSにおける使い方

- まとめると…
- $ module load pgi/12.2 pgi/cuda_fortran_4.1 cuda/4.1.28 mvapich2/1.8_pgi
- $ pgfortran -fast -o test test.f

HA-PACSにおける使い方

- MPIを使う場合は
  - $ mpif90 -fast –o test_mpi test_mpi.f90
- MPI + CUDAを使う場合は
  - $ mpif90 -fast -Mcuda=cc20 -o test_mpi_cuda test_mpi_cuda.cuf

HA-PACSで簡単な実演

- DAXPYを実装する
  - メモリ確保
  - データ転送
  - カーネル記述
  - カーネル起動
  - エラー処理

HA-PACS環境の注意点

- 現在HA-PACSに導入されているPGIコンパイラのバージョンは12.2
  - 最新版は12.10
- いくつかの機能が使えない
- バグが残ったままになっている
- 12.10にアップデート予定
  - hptクラスタは更新済みなので、試すことは可能

HA-PACS環境の注意点

- 12.2では一部の機能が未実装
  - OpenACC
  - Texture Memory
  - (HA-PACSでは関係ないが)CUDA Toolkit 4.2に未対応

HA-PACS環境の注意点

- バグ
  - 最適化(-fast)を有効にした状態で、カーネルコード内で変数をvolatileで修飾すると、まったく動作しないコードが生成される
  - CUDA Toolkit 4.1付属のCUBLASを使用するとリンクエラーになる
    - CUDA Toolkit 4.0のものを使えば回避可能
- バグ？
  - CUFFTのFortranインターフェイスがないので使えない
- これらの問題は12.10では修正されている