GPUチュートリアル：OpenACC篇

Himeno benchmark を例題として

高エネルギー加速器研究機構 (KEK)

松古 栄夫 (Hideo Matsufuru)

1 December 2018

HPC-Phys 勉強会@理化学研究所

Contents

- Hands On 環境について
- Introduction to GPU computing
- Introduction to OpenACC
- 実習： Himeno benchmark の offload

Contents

- Hands On 環境について
- Introduction to GPU computing
- Introduction to OpenACC
- 実習： Himeno benchmark の offload

Hands on

- HPCtech製 ポータブルGPUワークステーション
  - NVIDIA GeForce GTX 680M (Kepler architecture)
  - CUDA Toolkit 10.0
  - PGI compiler Community Edition 16.4 (無償版)
- OpenACC in a Nutshell
  - 簡単なまとめ

Hands on

- Himeno benchmark
  - http://accc.riken.jp/en/supercom/documents/himenobmt/
  - 3次元Poisson方程式をJacobi反復法で解く (疎行列, stencil)
  - 理研・情報基盤センターの姫野龍太郎氏による
- 例題
  - Himeno benchmark を多少改変して使う (Cの場合)
  - 性能評価: subroutine jacobi → ここをオフロード

    (いじりやすいようにしたもの: jacobi2)
  - 反復ループ自体が一つの subroutine の中にまとまっていてオフロードしやすい

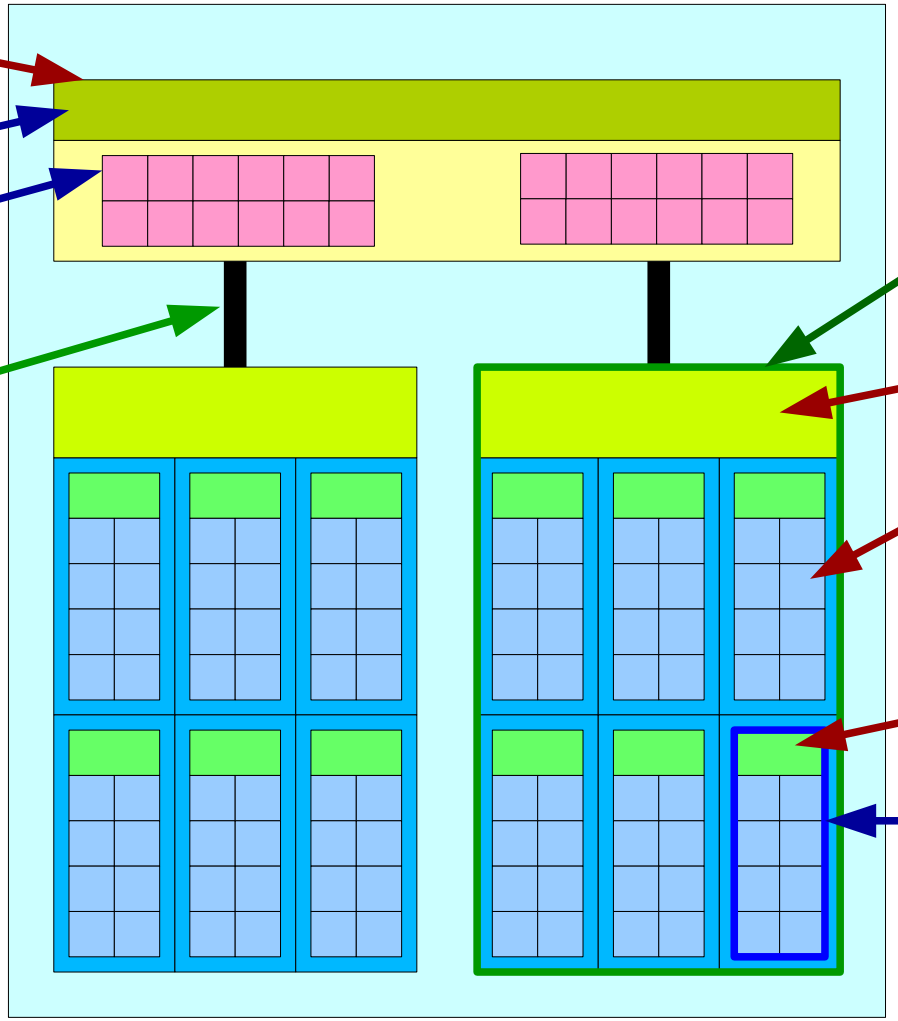
Hands on

- 例題コード: /home/workshop/
  - himeno-benchmark/ host用コード: ここからスタート

    → 各自 login したらコピーして、compile, 実行してみる
  - himeno_C_OpenACC/: C での実装例
  - himeno_F77_OpenACC/: Fortran 77 での実装例
- C code
  - Jacobi2: 構造体のままだと処理が難しいので、オフロードの前に要素に対するポインタを設定しておく → 配列のように処理 (_mod.c)
- F77 code
  - S size のものを利用
- F90 code
  - OpenACC篇ではフォローしない (CUDA篇で使用)

Introduction to GPU computing

GPU システムの一般的構成



NVIDIA GPU

- NVIDIA GPU

	GTX 680M (Kepler)	Tesla K40 (Kepler)	Tesla P100 (Pascal)	Tesla V100 (Volta)
Number of SM	7	15	56	80
FP64 Cores/GPU	448	960	1792	2560
FP32 Cores/GPU	1344	2880	3584	5120
Peak FP64 [TFlops/s]	0.68	1.7	5.3	7.8
Peak FP32 [TFlops/s]	2	5	10.6	15.7
Memory B/W [GB/s]	115	288	732	900
PCIe Gen3x16 [GB/s]	31.5 (x0.15)	31.5	31.5	31.5
NVLink [GB/s/port]	N/A	N/A	40	50

性能値はboost clock**時**　　* Tensor Core

(Hands on machine real status: 2.5GT/s x8)

HPC on GPU

- GPUを効果的に使える計算のタイプ
  - 一部に計算時間が集中（ホットスポット）

    → その部分だけGPUに**「オフロード」**することで性能向上
  - 多くの独立な計算に分割可能 → 多数のスレッドによる並列処理
    - 長いループ、依存関係のない配列
    - ある程度複雑な処理
  - データの局所性
    - Host (CPU) と device (GPU) の間のデータ転送が bottleneck

      → このデータ転送を少なく／演算とオーバーラップできると高速

Coding framework

- Device を host (CPU) から制御
- Device 上で実行されるコード (kernel code)
  - ユーザが作成 (API-based): CUDA, OpenCL
  - コンパイラが生成 (directive based): OpenACC
  - ライブラリを利用 (cuBLAS など)

	API based	Directive based
分散並列	MPI	XcalableMP
スレッド並列	Pthread TBB, C++ thread	OpenMP 自動並列化＋指示文
オフロード	OpenCL **CUDA (NVIDIA)**	**OpenACC** OpenMP 4.x Hitachi Fortran

Coding framework

- Device 上のメモリ
  - Host ⇔ device のメモリ空間は一般に独立
    - 最近は unified memory も登場
  - Host-device 間のバンド幅が bottleneck: PCIe, NVLink

    → データ転送を最小化する必要
- Device 上のカーネルコード
  - 数千のコアで並列実行できるように演算を配置
    - 互いに依存関係のないタスクに分解
    - SM (streaming muliprocessor) 内では共有メモリを通してデータ交換
  - アーキテクチャの特長を理解して最適化
    - SIMD実行されるスレッド数、共有メモリの利用など
    - Coalesced access: スレッドからのメモリアクセスを最適化するように配列順序を調整 → Array of Structure (of Array)

Coding framework

一般的な処理の流れ

- デバイスを使う準備
  - どのデバイスを使うか、など
- デバイス上のメモリ領域の確保
  - Host での malloc に対応 (C++ の new)
- データ転送: host → device
- デバイス上のカーネル実行
  - Host からカーネル実行を発行する
- データ転送: device → host
- メモリ領域の解放
  - Host での free に対応 (C++ では delete)

OpenACC

- OpenACC とは
  - 演算加速器を使うためのディレクティブ・ベースのライブラリ
    - スレッド並列の OpenMP に対応
  - C/C++, Fortran で利用可能
  - 対応しているcompiler
    - PGI compiler (NVIDIA 傘下)　(← 今回利用)
    - Cray compiler (使ったことない)
- References
  - OpenACC Home: http://www.openacc.org/
  - 「OpenACC ディレクティブによるプログラミング」(PGI)

    https://www.softek.co.jp/SPG/Pgi/OpenACC/ (和訳版)

    ← 以下ではこの文書をかなり参考にしている

OpenACC

- OpenACC の基礎
  - Directive のフォーマット

```
#pragma acc directive [clause[[,] clause]...]
{
  構造化ブロック
}
```

  - コンパイラは directive を解釈し、カーネルコードを生成
    - 「カーネル」：アクセラレータ側で動作するコード
    - OpenACCの並列化の対象は loop
    - 対象部分を procedure として切り出し、device code を生成

OpenACC

- 3種類の directive 構文が基本
  - 並列領域の指定 = Accelerator compute 構文
    - アクセラレータ上にオフロードするループ対象部分を指定する
    - 並列化階層それぞれのサイズを指定
    - Parallel 構文、kernels 構文, (Serial 構文) がある
  - メモリの確保とデータ移動（host ←→ device）= data 構文
    - データの転送(存在場所)を明示的に指示
    - enter, exit, update 構文を使う方法もある (Cf. 第一回の資料)
  - スレッド並列にするタスクの指定 = loop 構文
    - 並列化する for を指定
    - どの階層にそのループを割り当てるかを指定
    - Reduction 処理も可能

OpenACC

- Accelerator Compute 構文: 並列実行領域の指定
  - OpenMP の #pragma omp prallel に対応
  - Kernels 構文 → コンパイラまかせ型
    - 並列化のための依存性解析や並列性能に関するスケジューリングなどの責任はコンパイラが負う
    - 「tightly nested loop」に適用
  - Parallel 構文 → ユーザまかせ型 (←こちらを使う)
    - 並列分割方法やその場所を指定、依存性などにはユーザが責任を持つ
    - この構文から後、スレッドに対する冗長実行が開始
    - Work-sharing を行うループに対し、loop directive で指示
    - Work-sharing loop の終わりで同期は取らない
    - Parallel region の最後で同期を取る

OpenACC

- デバイスメモリの確保とデータ転送
  - コンパイラは並列領域内で使用する変数のメモリをデバイス上に確保
  - data 構文や declair 構文で明示的に生成できる
  - 配列は明示的に生成する必要あり
- data 構文
  - プログラムがデータ構文に到達したときにデバイスメモリ領域を生成、データ転送 → 出る時に(ホストへ)データ転送、メモリ解放
  - コピーの方法を clause で指定
    - copyin(array[size]): host → device のコピー
    - copyout：終了後、device → host のコピー
    - copy：copyin & copyout
    - present：既に存在するデータであることを指定

OpenACC

- もう一つの方法
  - enter 構文で確保、exit 構文で解放、update 構文で更新
    - OpenACC 2.0 から可能
    - オブジェクトの生成・破棄時にメモリ領域を確保・解放できる
    - Parallel region の前では　#pragma acc data present(var-list)
- アクセラレータ上のデータ
  - private: parallel 構文で指定、gang に private
  - Gang 内(worker, vector)で共有される

OpenACC

- 並列実行の階層構造
  - 3階層 – それぞれのサイズを clause で指定
  - gang: streaming multi-processor (NVIDIA) に相当、同期無し
  - worker: warp に対応 (同期可)
  - vector: warp 内 thread に相当、最内loop
- Loop構文
  - parallel 構文の中では work-share する loop を指定するために必須
  - collapse(n): n個のnestされたloopをまとめる
  - reduction(operation:var-list)
  - private(var-list)そのloopでプライベートな変数にする

A recipe of offloading

- 以上のOpenACC 機能をどう使ってオフロードコードを書くか？
  - 典型的な使い方: “recipe” があると便利
  - Recipe (1)：data構文でメモリ確保＋転送, parallel 構文で並列化
  - Recipe (2)：enter data でメモリ確保、update で転送、parallel 構文で並列化 (my favorite → 第1回 HPC-phys talk)
  - 最小限のまとめ：「OpenACC in a nutshell」
  - 各自の recipe を探してください

CUDA, OpenCL との比較

- CUDA, OpenCL
  - API ベースの framework
  - Kernel コードを自分で書く必要あり
  - きめ細かく最適化できる
  - OpenCL ではデバイス環境の設定がちょっと面倒
- OpenACC から CUDA, OpenCL へ
  - 上の recipe は、CUDA/OpenCL での手順とほぼ 1対1 対応
  - → それぞれの directive を API に置き換えていける
  - #pragma acc data 構文で指定した変数 = kernel code の引数
  - #pragma acc loop で括った内部を kernel code として抽出

Summary

- メニーコア並列化が可能なタイプの計算に GPU は効果的
- OpenACC を利用すると比較的手軽にGPUを使える
- GPU アーキテクチャの構造を理解することが高速化に重要
  - ホスト-デバイス間データ転送の最小化
  - メモリアクセスの最適化
  - スレッドの階層性
  - 共有メモリ
- CUDA, OpenCL へ向けての準備としても有効

Hands On

(1) まずはそのままコンパイル → 実行

(2) OpenACC directive を挿入 → コンパイル → 実行

- Rehersal での gosa の値は2桁くらいは合ってないとおかしい

(3) いろいろ試してみる

- num_workers, vector_length を変えてみる
- 正しくない結果になるような変更
- Etc.