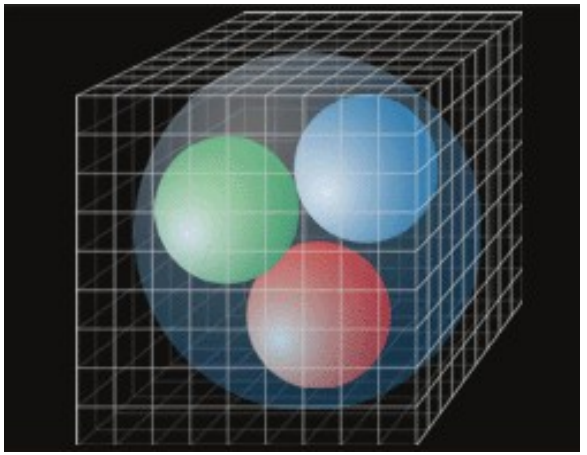
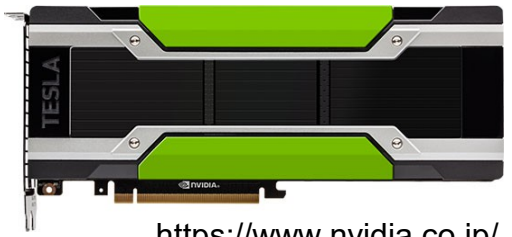
OpenACCで簡単GPU並列化

格子QCDと球対称超新星シミュレーションを例として





https://www.nvidia.co.jp/

高エネルギー加速器研究機構 (KEK)

松古 栄夫 (Hideo Matsufuru)

22 August 2018

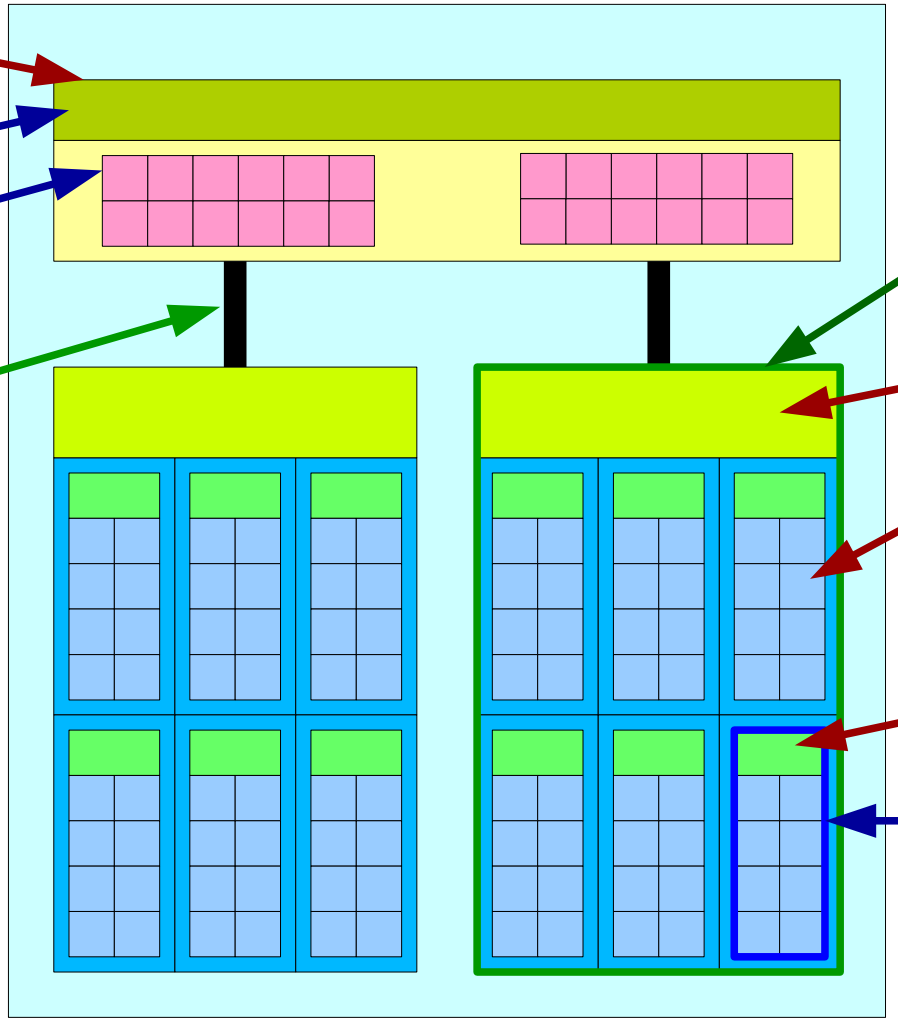
HPC-Phys 勉強会@京都大学基礎物理学研究所

Contents

- Introduction to GPU computing
- Introduction to OpenACC
- A recipe of offloading
  - Offloading matrix-vector multiplication
  - Example 1: Lattice QCD
  - Example 2: spherically symmetric supernova
- Summary

GPUの利用

- GPU システムの一般的構成



NVIDIA GPU

- NVIDIA GPU (Tesla series)

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

性能値はboost clock**時**　　* Tensor Core

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)　(← 今回利用した)
    - 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-name [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
  - 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 (←こちらを使う)
  - 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
    - 他にも何種類かの構文

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ータ領域を生成、データ転送される
  - 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、最外 loop
  - worker: warp に対応
  - 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
  - My recipe: lattice QCD, 超新星シミュレーションに適用
  - C-based (Fortran の場合は C-code に移植して Fortran から呼ぶ)
  - 行列・ベクトル積のコードを例に

- Step (0): device 環境の準備
  - デバイス環境の設定 (ここは CUDA)
    - 使えるデバイス数を取得: acc_get_num_devices() 関数
    - 使うデバイスを指定:　acc_set_device_num() 関数

  → 複数GPUの使用 (MPI並列化で各プロセスに割り当てる、等)

A recipe of offloading

- 以下のようなコードを考える (行列ベクトル積)

```
double *A_c, *v_c, *w_c;   // as global variables
int Nv = 100;

inline int index_mat(int i, int j){ return i + Nv * j; }
inline int index_vec(int i){ return i; }

...

void mult(double* v2, double* v1){  // v2 = A * v1

  for(int i = 0; i < Nv; ++i){
       v2[index_vec(i)] = 0.0;
    for(int j = 0; j < Nv; ++j){
       v2[index_vec(i)] +=
               A_c[index_mat(i,j)] * v1[index_vec(j)];
    }
  }

}
```

多次元配列っぽく使うためと、
メモリ上の配置を変更するため導入

A recipe of offloading

- Step (1): device memory の確保
  - Host で memory を確保する malloc のすぐ後で確保
  - 以後 host のポインタで device のデータを参照できる

```
double *A_c, *v_c, *w_c;   // as global variables
int Nv = 100;
...

init(){

  A_c = (double*)malloc(Nv * Nv * sizeof(double));
  #pragma acc enter data create(A_c[0:Nv*Nv])

  w_c = (double*)malloc(Nv * sizeof(double));
  #pragma acc enter data create(w_c[0:Nv])

  v_c = (double*)malloc(Nv * sizeof(double));
  #pragma acc enter data create(v_c[0:Nv])

}
```

配列サイズを指定

A recipe of offloading

- Step (2), (4): データ転送
  - Device で実行したい部分の前後でデータを移動

```
main(){

 init();

  #pragma acc update device(A_c[0:Nv*Nv])
  #pragma acc update device(w_c[0:Nv])
                  host → device の転送

  for(i=0; i<10; ++i){
    mult(v_c, w_c);
    mult(w_c, v_c);
  }

  #pragma acc update host(w_c[0:Nv]);
                  device → host の転送
  tidyup();
}
```

コピーする起点とサイズを指定する
(一部分の転送も可能)

A recipe of offloading

- Step (3): device kernel 実行
  - data 構文で変数の扱いを指定
    - copyin, copyout, copyinout, present, create
  - parallel 構文で device 上で実行する範囲を指定
    - worker, vector のサイズを指定
  - loop 構文でスレッド並列化するループを指定
    - gang, worker, vector の階層を指定
  - 非同期実行
    - parallel 構文に async(<num>) clause をつける (<num> は数値)
    - #pragma acc wait(<num>) で同期を取る
    - update 構文でも使用可

A recipe of offloading

```
void mult(double* v2, double* v1)  // v2 = A * v1
{
 #pragma acc data present(v2[0:Nv], v1[0:Nv], A_c[0:Nv*Nv]) \
                  copyin(Nv)
 {
  #pragma acc parallel num_workers(NUM_WORKERS) \
                       vector_length(VECTOR_LENGTH)
  {
   #pragma acc loop independent ganga worker vector
   for(int i = 0; i < Nv; ++i){
       v2[index_vec(i)] = 0.0;
     for(int j = 0; j < Nv; ++j){
       v2[index_vec(i)] +=
              A_c[index_mat(i,j)] * v1[index_vec(j)];
     }
   }
   ...  ここにもう一つループを入れると上のループ終了を待たずに発行

  } // acc parallel
 } // acc data

}
```

データは既にdevice上にある

Nv は copyin する (メモリ確保していない変数)

この領域 { ... }が device 上で実行される
(#define で NUM_WORKERS, VECTOR_LENGTH を定義しておく)

このループをスレッド並列化
(3階層すべて利用)

スレッド間の同期を取る

A recipe of offloading

- (5) device memory の解放
  - Host のメモリ領域解放の直前で実行

```
tidyup(){  // function to free the memory resources

  #pragma acc exit data delete(A_c[0:Nv*Nv])
  free(A_c);

  #pragma acc exit data delete(w_c[0:Nv])
  free(w_c);

  #pragma acc exit data delete(v_c[0:Nv])
  free(v_c);

}
```

A recipe of offloading

- コンパイル、リンク
  - PGI コンパイラの場合、-acc オプションでOpenACC を解釈
    - 最適化: -fast
    - ターゲットアーキテクチャ指定: -ta=tesla:ptxinfo,cc60,cuda8.0
      (P100 の場合: cc は computger capability, cuda のバージョン)
    - 情報の表示: -Minfo
    - 3重に inline 展開: -Minline=levels:3
    - メモリアドレスの拡張: -mcmodel=medium (必要なら large)
  - Fortran と C の interoperavility
    - C-code: void, 関数名は小文字、最後に “_” を付ける、引数はポインタ
    - Fortran: C関数の “_” を外して call, 渡しているのは配列の先頭アドレス
    - C++ では名前のマングリングに注意 (extern “C” で括るなど)

A recipe of offloading

- 実行
  - 使用メモリの拡張: ulimit -s unlimit
  - メモリチェックしながら実行: cuda-memcheck ./a.out
  - PGI のプロファイラもある

Example 1: Lattice QCD

- Bridge++ benchmark

  H. Matsufuru et al., Proc. Comp. Sci. 51 (2015) 1313

  - 2015年初めに実装 (PGI 14.10 使用)
  - C++ template の中で使うと正しく動作しない

    → C のコードとして外部で実装した関数を template クラスから呼ぶ
  - 現在は解消されているかも知れない
  - Wilson fermion solver: coalesced access へのデータ配列変換
- Complex Langevin コード
  - Original code: Fortran code by KEK 西村さん
  - Recipe 通りの手順でオフロード :
    - Step (1) Solver → NVIDIA P100 で 38 GFlops 程度
    - Step (2) Other parts (staple → force など)

Example 1: Lattice QCD

- Wilson fermion benchmark

  S. Motoki et al., Lattice 2015 proceedings: PoS (LATTICE 2015) 040

  - K40: 4290(FP32)/1430(FP64) GFlops, memory B/W 288 GB/s
  - Compiler: PGI 14.10 (現在は AMD GPU はサポート外)

	OpenCL		OpenACC	
operation	float	double	float	double
NVIDIA Tesla K40 (Kepler):				
Wilson mult	232 GFlops	121 GFlops	196 GFlops	39.1 GFlops
CG solver	160 GFlops	86.0 GFlops	123.7 GFlops	35.0 GFlops
NVIDIA Tesla M2090 (Fermi):				
Wilson mult	154 GFlops	39.2 GFlops	149 GFlops	22.9 GFlops
CG solver	107 GFlops	33.7 GFlops	84.3 GFlops	21.1 GFlops
AMD FirePro W9100:				
Wilson mult	179 GFlops	110 GFlops	123 GFlops	19.0 GFlops
CG solver	120 GFlops	79.5 GFlops	130 GFlops	20.6 GFlops

Table 2: Performance with OpenCL and OpenACC for Wilson fermion matrix mult and CG solver on a $16^3 \times 32$ lattice.

Example 2: 1D Supernova

- Spherically symmetric supernova simulation

  H. Matsufuru and K. Sumiyoshi, ICCSA 2018

  - Boltzmann equation (neutrino) + Hydro (matter) + GR (重力)

    (Cf. 住吉さんの review talk)
  - 1D 近似では通常のパラメターでは爆発しない
  - 重力波、ニュートリノの観測データとの比較、多次元計算の基礎

    → 高精度の系統的計算が必要
  - Bottleneck

    (1) 陰解法で必要な線形方程式の反復解法 (全体の90%以上)

    → ブロック三重行列とベクトルの積

    (2a) 線形計算の準備 (最適化パラメターの計算): 行列演算

    (2b) Boltzmann eq. の反応項 (反応率の計算): ほぼ積分

Example 2: 1D Supernova

- (1) の反復解法部分を GPU にオフロード

H. Matsufuru and K. Sumiyoshi, ICCSA 2018

  - 行列ベクトル積に cuBLAS library を使用 (独自実装も有り)
  - NVIDIA P100 (Pascal) x 4 + POWER8 x2 + NVLink

Power/Pascal (Set-2)	Np = 16	Np = 4	GPU (cuBLAS)
Collision (2b)	8.6	35.1	
Matrix total	168.0	622.5	45.9
Setup (2a)	11.8	43.7	45.0
Inversion (7 iter) (1)	156.2	578.8	0.9

- (2a), (2b) のオフロードコード実装中
  - (2a) は行列演算なので (1) とほぼ同様
  - (2b) は反応率計算＋積分 → オフロードで大幅高速化

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

- まとめ
  - 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
  - 行列ベクトル積の場合を例に、典型的な recipe を紹介
  - 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使える