OpenACC による共役勾配法カーネルコードの並列化と実行性能評価

川口 優樹[1,a)]　宮島 敬明[2]　藤田 直行[2]

概要：近年，HPC 分野では様々な並列計算機が登場しており，効率的にプログラムの並列化をする手法が検討されている．特に GPU の利用には専用開発環境 CUDA を用いる必要があるため開発の敷居が高いとされていたが，ディレクティブを記述するだけで GPU が利用可能となる OpenACC が登場した．しかしながら，OpenACC は発展途上の手法であり CUDA での実装や他の並列計算機との比較研究が少ない．本研究の目的は，連立一次方程式ソルバーの 1 つである共役勾配法のカーネルコードを OpenACC を用いて並列化した際の実行性能を各環境で評価することである．評価には共役勾配法のカーネルコードに対して OpenACC, OpenMP, CUDA を適用し, Xeon (Skylake), GPU (P100), FX100 環境を用いて実行性能の比較を行う．結果，FX100 に対し OpenACC を用いた P100 の実行性能は 3.6 倍以上の高速化を達成した．

キーワード：C++, OpenACC, OpenMP, CUDA, 共役勾配法, JSS2

1. 序論

工学的, 物理的現象を定式化すると偏微分方程式の初期値境界値問題が得られ, 差分法や有限要素法を用いて離散化すると大規模な連立一次方程式を解くことに帰着する．多くの数値シミュレーションでは連立一次方程式の求解に時間を要するため, 求解の高速化が求められている．近年では連立一次方程式ソルバーを様々なアーキテクチャに実装し, 求解の高速化を図る研究がされている [1], [2].

GPU を用いることで連立一次方程式の求解の高速化が期待できる．GPU 向けアプリケーションの開発には CUDA と呼ばれる開発環境を用いるのが主流である．しかしながら, CUDA を用いる場合は利用者に対象 GPU のアーキテクチャを考慮した並列計算プログラミングや最新 GPU アーキテクチャが登場するたびにソースコード最適化が必要になるなどの課題がある．同課題の解決方法として OpenACC が注目されている [3].

OpenACC は近年登場した GPU 向け API であり, コンパイラ・ディレクティブやランタイム・ライブラリ, 環境変数を提供する．ユーザがディレクティブとディレクティブに付随する節 (clause) を既存のコードに追加すると, コンパイラが自動的に GPU 向けのコードを生成する．また, コンパイラによってはコンパイルオプションで CPU 向けに並列化したコードを生成することも可能であり, ソースコードの移植性及び保守性も高い.

本研究の目的は, 連立一次方程式の解法の 1 つである共役勾配 (Conjugate Gradient, CG) 法の

1　宇宙航空研究開発機構 セキュリティ・情報化推進部 スーパーコンピュータ活用課
2　宇宙航空研究開発機構 航空技術部門 数値解析技術研究ユニット 計算情報基盤セクション
a)　kawaguchi.yuuki@jaxa.jp

カーネルコードに対し, OpenACC を適用した際の実行性能の比較をすることである. CPU 向け OpenACC は OpenMP, GPU 向け OpenACC は CUDA と実行性能を比較する. また, 同 OpenACC の実行性能と JAXA Supercomputer System Generation 2 (JSS2) の計算機の実行性能を比較する.

2. 共役勾配法のカーネルコード

工学的, 物理的現象から得られる連立一次方程式の係数行列は零要素を多く含む疎行列になることが極めて多い. 一般的に同分野の連立一次方程式に対しては前処理付き Krylov 部分空間解法が広く利用される. 共役勾配法は Krylov 部分空間解法の一種であり, 連立一次方程式の係数行列が正定値対象行列の場合によく利用される.

共役勾配法のアルゴリズムを図 1 に示す [4]. 共役勾配法のアルゴリズムはベクトル同士の加算演算, 内積, 行列ベクトル積で構成されており, 並列化がしやすいという特徴がある. また, 共役勾配法の主要な演算は行列ベクトル積であり, 係数行列の非零要素のみを効率的にメモリに格納及び計算することで, 演算量とメモリ容量の削減が可能である. 疎行列の格納形式として Compressed Row Storage (CRS) 形式や Ellpack-Itpack (ELL) 形式がよく利用される. CRS 形式の行列格納形式を図 2, ELL 形式の行列格納形式を図 3 に示す [5]. なお, 同疎行列の格納形式を用いた場合の疎行列ベクトル積の計算方法は後述する.

本研究では上述のベクトル同士の加算演算, 内積, 密行列ベクトル積, CRS 形式の疎行列ベクトル積, ELL 形式の疎行列ベクトル積を共役勾配法のカーネルコードと呼称する. 本カーネルコードを十分に最適化することで, 共役勾配法においても十分な実行性能を確立できると考えている.

3. C++ に対する OpenACC の適用

OpenACC は C/C++ や Fortran のプログラムに対して OpenMP の様にディレクティブ (`#pragma acc kernels`) を記述することでコンパイラに GPU 化したコードや CPU でマルチコア化したコードを生成させる. 本研究では C++ の

Let $\boldsymbol{x}_0$ be an initial guess.
$\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$
$\boldsymbol{p}_0 = \boldsymbol{r}_0$
for $k = 0, 1, \cdots$, until $\|\boldsymbol{r}_k\|_2/\|\boldsymbol{b}\|_2 \le \varepsilon$ **do**
$\quad \alpha_k = \dfrac{(\boldsymbol{r}_k, \boldsymbol{r}_k)}{(\boldsymbol{p}_k, A\boldsymbol{p}_k)}$
$\quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
$\quad \boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k A\boldsymbol{p}_k$
$\quad \beta_k = \dfrac{(\boldsymbol{r}_{k+1}, \boldsymbol{r}_{k+1})}{(\boldsymbol{r}_k, \boldsymbol{r}_k)}$
$\quad \boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$
end for

図 1 共役勾配 (CG) 法のアルゴリズム.

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 5 & 0 & 6 & 7 \\ 0 & 8 & 0 & 9 \end{bmatrix}$$

```
mat = [1 2 3 4 5 6 7 8 9]
col = [0 2 1 3 0 2 3 1 3]
row = [0 2 4 7 9]
```

図 2 CRS 形式の格納例. ただし, `mat` は行列の非零要素を格納する配列, `col` は非零要素の列番号を格納する配列, `row` は各行の先頭非零要素の位置を格納する配列であり, `row` の最後の要素には行列の非零要素数を格納する.

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 5 & 0 & 6 & 7 \\ 0 & 8 & 0 & 9 \end{bmatrix}$$

```
m   = 3
mat = [1 3 5 8 2 4 6 9 0 0 7 0]
col = [0 1 0 1 2 3 2 3 * * 3 *]
```

図 3 ELL 形式の格納例. ただし, `m` は 1 行に含まれる最大非零要素数, `mat` は行列の非零要素を格納する配列, `col` は非零要素数の列番号を格納する配列とする. また, * はパディングデータである.

```
void axpy(std::vector<double>      & ret,
          double              const  alpha,
          std::vector<double> const& x,
          std::vector<double> const& y){
 auto const n = ret.size();
# pragma acc kernels
 for(decltype(ret.size()) i=0; i<n; ++i){
  ret[i] = alpha * x[i] + y[i];
 }
}
```

図 4　std::vector を用いたベクトル同士の加算プログラムに対する OpenACC の適用例. 本例は PGI コンパイラ 17.10 の OpenACC では並列化できない.

プログラムに対して OpenACC の適用を検討する.

一般的に C++ のプログラムでは std::vector などの Standard Template Library (STL) のコンテナクラスやユーザー定義のクラスを多用したプログラムが極めて多い. std::vector を用いたベクトル同士の加算プログラムに対して OpenACC を適用した例を図 4 に示す. しかしながら, 本例は PGI コンパイラ 17.10 の C++ OpenACC では派生クラスのメンバ変数を含めたデータを GPU に代入する full deep copy が未サポートであるため並列化することができない [6].

上述の課題はメンバ変数をローカル変数に代入することで回避が可能である. std::vector においては C++11 から data メンバ関数が追加され, 同メンバ関数を利用することでローカル変数に代入が可能となる. しかしながら, 代入するメンバ変数がポインタ変数の場合は__restrict__ 修飾子を用いて変数の依存関係を明示的にコンパイラに指定しないと並列化が促進されない.

GPU の利用を考慮する場合, #pragma acc kernels だけでは該当コードが呼ばれるたびにメインメモリから GPU のデバイスメモリにデータ転送が発生する可能性がある. data 構文を用いることでデータ転送を制御することができるため実行性能が向上する可能性がある.

上述の最適化を適用したベクトル同士の加算例を図 5, 内積プログラム例を図 6, 密行列ベクトル積のプログラム例を図 7, CRS 形式の疎行列ベクトル積のプログラム例を図 8, ELL 形式の疎行列ベクトル積のプログラム例を図 9 にそれぞれ示す. また, 同プログラムは上述の最適化に加えて OpenMP を用いた並列化手法も記述している.

```
void kernel(int const n,
            double      * __restrict__ ret,
            double const               alpha,
            double const* __restrict__ x,
            double const* __restrict__ y){
#if defined (_OPENMP)
# pragma omp parallel for
#  if defined (__FUJITSU) && dedined (__sparc)
#  pragma loop xfill
#  endif
#elif defined (_OPENACC)
# pragma acc kernels present(ret, x, y)
#endif
 for(int i=0; i<n; ++i){
  ret[i] = alpha * x[i] + y[i];
 }
}

void axpy(std::vector<double>      & ret,
          double              const  alpha,
          std::vector<double> const& x,
          std::vector<double> const& y){
 auto const n  = ret.size();

 auto const px = x.data();
 auto const py = y.data();
 auto       pz = z.data();

 // host to device
 #pragma enter data copyin(px[0:n])
 #pragma enter data copyin(py[0:n])
 #pragma enter data copyin(pz[0:n])

 kernel(n, pz, alpha, px, py);

 // device to host
 #pragma acc exit data copyout(pz[0:n])
}
```

図 5　最適化後のベクトル同士の加算の例. ただし, n はベクトルの要素数, ret は計算結果を格納するベクトル, alpha は演算対象スカラ変数, x と y は演算対象のベクトルとする.

```
double kernel(int const n,
              double const* __restrict__ x,
              double const* __restrict__ y){
 double sum = 0;

#if defined (_OPENMP)
# pragma omp parallel for reduction(+: sum)
#elif defined (_OPENACC)
# pragma acc present(x, y)
#endif
 for(int i=0; i<n; ++i){
  sum += x[i] * y[i];
 }

 return sum;
}

double dot(std::vector<double> const& x,
           std::vector<double> const& y){
 auto const n  = x.size();
 auto const px = x.data();
 auto const py = y.data();

 #pragma enter data copyin(px[0:n])
 #pragma enter data copyin(py[0:n])

 auto ret = kernel(n, px, py);

 return ret;
}
```

図 6 最適化後の内積の例. ただし, n はベクトルの要素数, x と y は演算対象のベクトルとする.

```
void kernel(int const n, int const m,
            double      * __restrict__ ret,
            double const* __restrict__ mat,
            double const* __restrict__ vec){
#if defined (_OPENMP)
# pragma omp parallel for
#elif defined (_OPENACC)
# pragma acc kernels present(ret, mat, vec)
#endif
 for(int i=0; i<n; i+=2){ // outerloop unrolling
  double tmp0 = 0.0, tmp1 = 0.0;
  for(int j=0; j<m; ++j){
   tmp0 += mat[(i  )*m + j] * vec[j];
   tmp1 += mat[(i+1)*m + j] * vec[j];
  }
  ret[i  ] = tmp0;
  ret[i+1] = tmp1;
 }
 // remainder loop is omitted
}

void gemv(std::vector<double>      & ret,
          std::vector<double> const& mat,
          std::vector<double> const& vec){
 auto const n = ret.size();
 auto const m = vec.size();

 auto       pret = ret.data();
 auto const pmat = mat.data();
 auto const pvec = vec.data();

 #pragma enter data copyin(pret[0:n])
 #pragma enter data copyin(pmat[0:n*m])
 #pragma enter data copyin(pvec[0:n])

 kernel(n, m, pret, pmat, pvec);

 #pragma acc exit data copyout(pret[0:n])
}
```

図 7 最適化後の密行列ベクトル積の例. ただし, n と m は行列の行数と列数, ret は計算結果を格納するベクトル, mat は行列の要素を格納する配列, vec は演算対象のベクトルとする. また, 本コードには Outerloop Unrolling を明示的に適用している.

```
void kernel(int const n,
            double      * __restrict__ ret,
            double const* __restrict__ mat,
            int    const* __restrict__ col,
            int    const* __restrict__ row,
            double const* __restrict__ vec){
#if defined (_OPENMP)
# pragma omp parallel for
#elif defined (_OPENACC)
# pragma acc kernels present(ret,mat,col,row,vec)
#endif
 for(int i=0; i<n; ++i){
  double tmp = 0.0;
  for(int j=row[i], end=row[i+1]; j<end; ++j){
   tmp += mat[j] * vec[ col[j] ];
  }
  ret[i] = tmp;
 }
}

void crs_spmv(std::vector<double>      & ret,
              std::vector<double> const& mat,
              std::vector<int>    const& col,
              std::vector<int>    const& row,
              std::vector<double> const& vec){
 auto const n = ret.size();

 auto       pret = ret.data();
 auto const pmat = mat.data();
 auto const pcol = col.data();
 auto const prow = row.data();
 auto const pvec = vec.data();

 #pragma enter data copyin(pret[0:n])
 #pragma enter data copyin(pmat[0:mat.size()])
 #pragma enter data copyin(pcol[0:col.size()])
 #pragma enter data copyin(prow[0:row.size()])
 #pragma enter data copyin(pvec[0:n])

 kernel(n, pret, pmat, pcol, prow, pvec);

 #pragma acc exit data copyout(pret[0:n])
}
```

図 8 最適化後の CRS 形式を用いた疎行列ベクトル積の例. ただし, `n` は行列の次元数, `ret` は計算結果を格納するベクトル, `mat` は行列の非零要素を格納する配列, `col` は非零要素の列番号を格納する配列, `row` は各行の先頭非零要素の位置を格納する配列, `vec` は演算対象のベクトルとする.

```
void kernel(int const n, int const m,
            double      * __restrict__ ret,
            double const* __restrict__ mat,
            int    const* __restrict__ col,
            double const* __restrict__ vec){
#if defined (_OPENMP)
# pragma omp parallel for
#elif defined (_OPENACC)
# pragma acc kernels present(ret)
#endif
 for(int i=0; i<n; ++i) ret[i] = 0;
#if defined (_OPENMP)
# pragma omp paralel
 {
#elif defined (_OPENACC)
# pragma acc kernels present(ret, mat, col, vec)
#endif
 for(int j=0; j<m; ++j){
#if defined (_OPENMP)
# pragma omp for
#endif
  for(int i=0; i<n; ++i){
   int const jn = j*n + i;
   ret[i] += mat[jn] * vec[ col[jn] ];
  }
#if defined (_OPENMP)
 }
#endif
}

void ell_spmv(int const m,
              std::vector<double>      & ret,
              std::vector<double> const& mat,
              std::vector<int>    const& col,
              std::vector<double> const& vec){
 auto const n    = ret.size();
 auto       pret = ret.data();
 auto const pmat = mat.data();
 auto const pcol = col.data();
 auto const pvec = vec.data();

 #pragma enter data copyin(pret[0:n])
 #pragma enter data copyin(pmat[0:mat.size()])
 #pragma enter data copyin(pcol[0:col.size()])
 #pragma enter data copyin(pvec[0:n])

 kernel(n, m, pret, pmat, pcol, pvec);

 #pragma acc exit data copyout(pret[0:n])
}
```

図 9 最適化後の ELL 形式を用いた疎行列ベクトル積の例. ただし, `n` は行列の次元数, `m` は 1 行に含まれる最大非零要素数, `ret` は計算結果を格納するベクトル, `mat` は行列の非零要素を格納する配列, `col` は非零要素の列番号を格納する配列, `vec` は演算対象のベクトルとする.

表 1 Skylake の計算機環境. ただし, CPU は OpenMP と OpenACC で 72 スレッド用いて並列化する.

OS	RHEL 7.3
Kernel	3.10.0
CPU	Intel Xeon Gold 6150 x 2
CPU Memory	DDR4 2600MHz 192 GB
GPU	NVIDIA Tesla P100
GPU Memory	HBM2 16 GB
Compiler	PGI 17.10
Compile Option (OpenMP)	`-std=c++11 -fast -mp`
Compile Option (OpenACC for CPU)	`-std=c++11 -fast -acc -ta=multicore`
Compile Option (OpenACC for GPU)	`-std=c++11 -fast -acc -ta=tesla,cc60`

4. 数値実験

本研究では, 前述の共役勾配法のカーネルコードに対し, 計算対象の問題規模を変化させた際の実行性能の比較を行う. また, 評価に利用する計算精度はいずれも倍精度とする.

ベクトル同士の加算演算, 内積, 密行列ベクトル積の計算対象はすべて乱数で生成する. CRS 形式と ELL 形式の疎行列ベクトル積で計算に利用する疎行列は対角要素の値を 2, 対角要素の両隣の要素の値を 1 とする三重対角行列とし, 演算対象のベクトルの要素は全て乱数で生成する.

4.1 OpenACC 向け最適化の有無による実行性能比較

本節では, OpenACC 向け最適化の有無による実行性能の比較を行う. また, CPU 向け OpenACC は OpenMP で記述したプログラムと実行性能を比較し, GPU 向け OpenACC は CUDA で記述したプログラムと実行性能を比較する [7]. さらに, 我々が作成した CUDA のプログラムが十分に最適化されているか判断するために CUDA の数値計算ライブラリである cuBLAS 及び cuSPARSE とも実行性能を比較する [8]. 本評価で利用する計算機環境を表 1 に示す.

CPU におけるベクトル同士の加算演算の比較結果を図 10, 内積の比較結果を図 11, 密行列ベクトル積を図 12, CRS 形式の疎行列ベクトル積を図 13, ELL 形式の疎行列ベクトル積を図 14 に示す. 問題を大規模化させた際に, OpenACC 向け最適化をすることでベクトル同士の加算演算は 5.3 倍, 密行列ベクトル積は 17.7 倍, CRS 形式の疎行列ベクトル積は 6.8 倍, ELL 形式の疎行列ベクトル積は 4 倍の高速化に成功している. しかしながら, 内積に関しては有意な差はみられない.

OpenMP を用いるよりも OpenACC を用いたほうがベクトル同士の加算演算は 5.5 倍, 内積の比較結果は 5.9 倍, 密行列ベクトル積は 7 倍, CRS 形式の疎行列ベクトル積は 6.1 倍, ELL 形式の疎行列ベクトル積は 4.3 倍高速である. 我々は CPU においては OpenMP と OpenACC は同じ結果になると考えていたが, PGI コンパイラにおいては両実装が異なるのが原因と考えている.

GPU におけるベクトル同士の加算演算の比較結果を図 15, 内積の比較結果を図 16, 密行列ベクトル積を図 17, CRS 形式の疎行列ベクトル積を図 18, ELL 形式の疎行列ベクトル積を図 19 に示す. OpenACC 向け最適化をすることで問題を大規模化させた際に, ベクトル同士の加算演算は 3607 倍, 密行列ベクトル積は 9455 倍, CRS 形式の疎行列ベクトル積は 7313 倍, ELL 形式の疎行列ベクトル積は 7156 倍の高速化に成功している. CPU 同様に内積に関しては有意な差はみられない.

最適化後の OpenACC よりも CUDA を用いることで密行列ベクトル積は 1.2 倍, ELL 形式の疎行列ベクトル積は 1.36 倍の高速化結果が得られている. cuBLAS 及び cuSPARSE を用いることで内積も 1.2 倍, CRS 形式の疎行列ベクトル積は 1.19 倍の高速化に成功している. 以上より, GPU における OpenACC は CUDA 環境を用いるよりも約 20%程度実行性能が劣るということがわかる. また, 本ケースでは OpenACC を用いると CUDA で作成したプログラムよりも利用できるメモリ領域が半分以下となることがわかる. しかしながら, OpenACC は前述の最適化をすればよいのに対し, CUDA は 400 行程度の新規ソースコード作成が必要であり, 移植性や保守性を考慮すると CUDA は OpenACC よりも開発コストが高い.

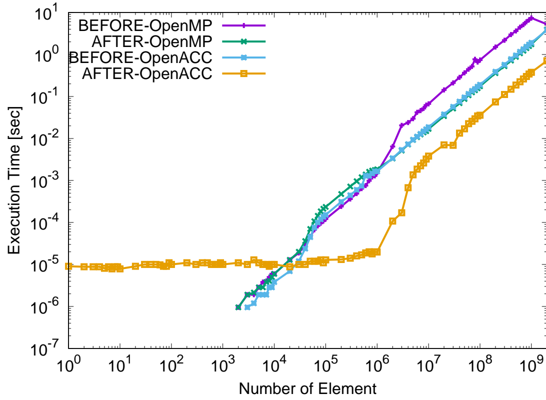

図 10 CPU におけるベクトル同士の加算演算のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

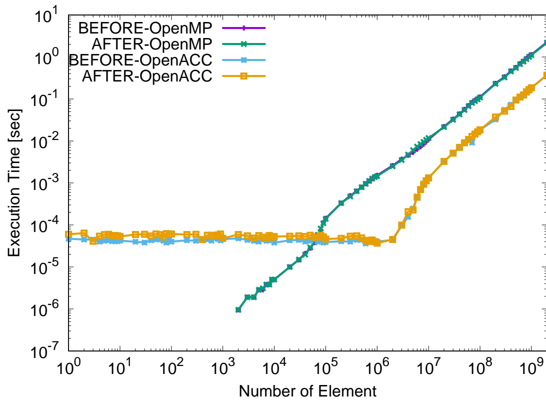

図 11 CPU における内積のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

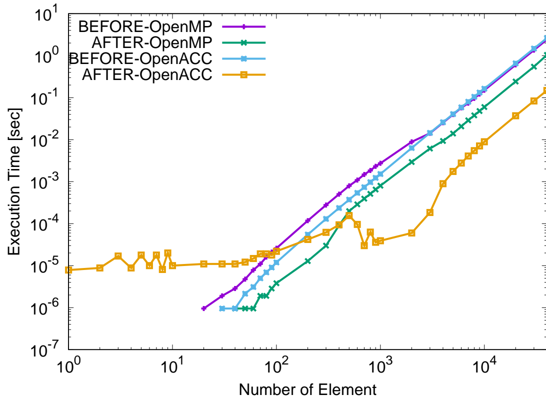

図 12 CPU における密行列ベクトル積のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

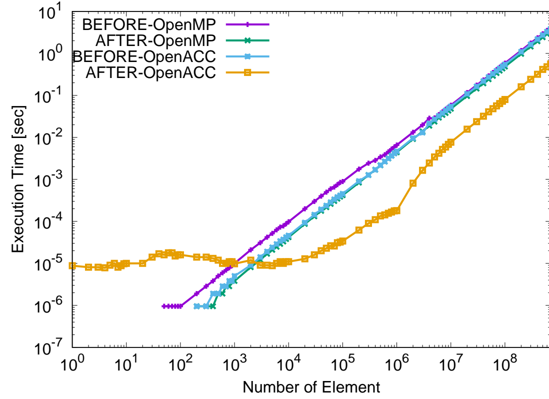

図 13 CPU における CRS 形式の疎行列ベクトル積のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

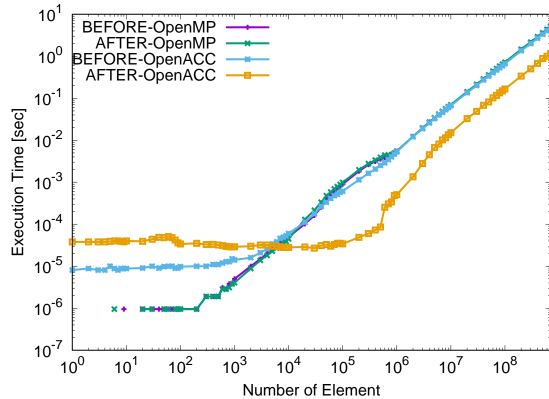

図 14 CPU における ELL 形式の疎行列ベクトル積のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

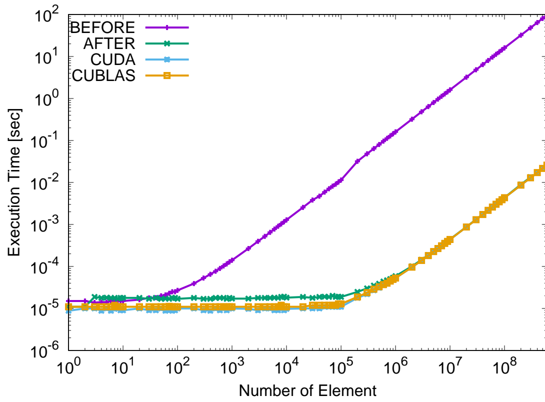

図 15　GPU におけるベクトル同士の加算演算のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

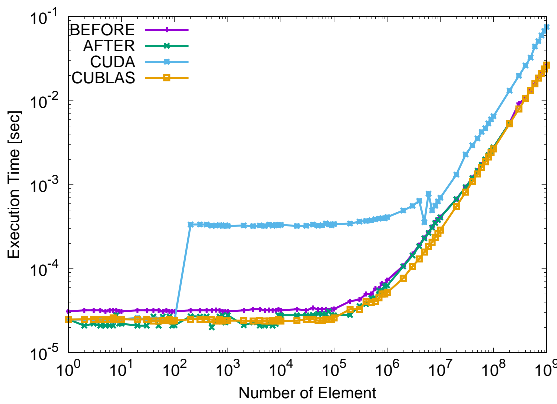

図 16　GPU における内積のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

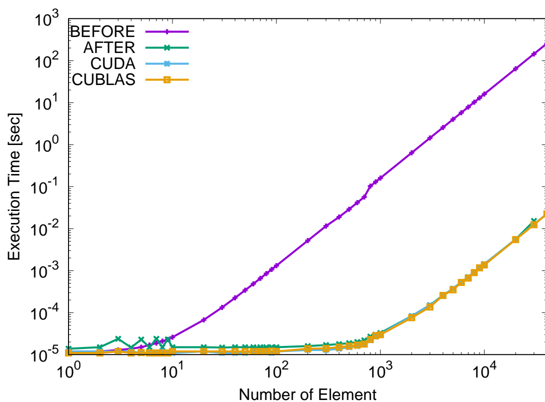

図 17　GPU における密行列ベクトル積のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

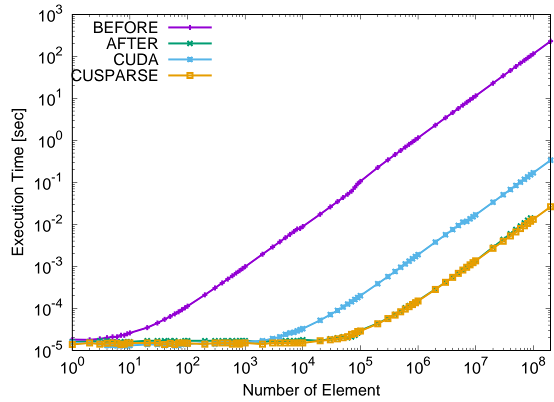

図 18　GPU における CRS 形式の疎行列ベクトル積のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

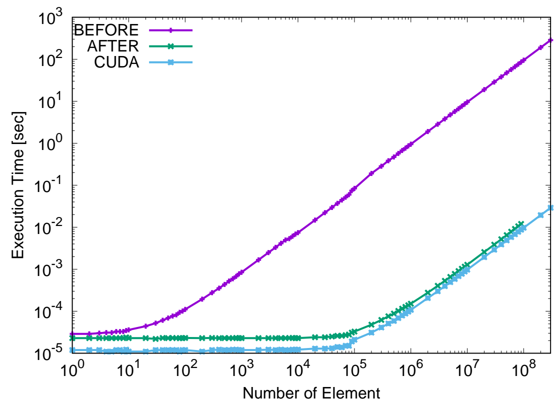

図 19　GPU における ELL 形式の疎行列ベクトル積のソースコード最適化有無による OpenMP と OpenACC の計算時間の比較．ただし，利用するスレッド数は 72 とする．

表 2 FX100 (SORA-MA) の計算機環境．ただし，OpenMP で 32 スレッドを用いて並列化する．

OS	XTC OS 2.1.1
Kernel	2.6.32
CPU	SPARC64 XIfx 2.2 GHz
CPU Memory	HMC 32 GB
Compiler	Fujitsu GM-2.0.0-06 beta
Compile Option	`-std=c++11 -Xg -Kocl -Kfast -Kopenmp -Kprefetch_sequential=soft`

表 3 Ivy Bridge (SORA-PPi) の計算機環境．ただし，OpenMP で 12 スレッドを用いて並列化する．

OS	RHEL 6.7
Kernel	2.6.32
CPU	Intel Xeon E5-2643 x 2
CPU Memory	DDR3 1600MHz 64 GB
Compiler	GCC 7.2.0
Compiler Option	`-std=c++11 -fopenmp -Ofast -march=native`

4.2 OpenACC 環境と JSS2 の実行性能比較

本節では前述の OpenACC 環境の結果と JAXA で運用・稼働中のスーパーコンピュータ JSS2 の計算機である FX100 (SORA-MA) と Ivy Bridge (SORA-PPi) との実行性能の比較を行う．FX100 の計算機環境を表 2, Ivy Bridge の計算機環境を表 3 に示す．各計算機環境におけるソースコード最適化後のベクトル同士の加算演算の比較結果を図 20, 内積の比較結果を図 21, 密行列ベクトル積の比較結果を図 22 に示す．また, CRS 形式の疎行列ベクトル積の比較結果を図 23, ELL 形式の疎行列ベクトル積の比較結果を図 24 に示す．

図 20 より, ベクトル同士の加算演算では問題を大規模化させた際に Ivy Bridge と Skylake よりも FX100 のほうが 2.3 倍以上高速に計算が可能である．また, FX100 よりも OpenACC を用いて P100 を利用することで 4.1 倍高速である．

図 21 より, 内積では問題を大規模化させた際に FX100 が Ivy Bridge よりも 1.02 倍, Skylake よりも 1.2 倍高速である．また, FX100 よりも OpenACC を用いて P100 を利用することで 5.4 倍高速である．

図 22 より, 密行列ベクトル積では問題を大規模化させた際に Ivy Bridge が FX100 よりも 1.04 倍, Skylake よりも 1.2 倍高速である．また, Ivy Bridge よりも OpenACC を用いて P100 を利用することで 4.4 倍高速である．

図 23 より, CRS 形式の疎行列ベクトル積においては問題を大規模化させた際に FX100 が Ivy Bridge よりも 2.1 倍, Skylake よりも 1.6 倍高速である．また, FX100 よりも OpenACC を用いて P100 を利用することで 3.6 倍高速である．

図 24 より, ELL 形式の疎行列ベクトル積においては問題を大規模化させた際に FX100 が Ivy Bridge よりも 1.8 倍, Skylake よりも 1.9 倍高速である．また, FX100 よりも OpenACC を用いて P100 を利用することで 6.7 倍高速である．

5. 結論

本研究では PGI コンパイラ 17.10 の C++ OpenACC 向け最適化手法を確立し, 最適化の有無による実行性能の評価を実施した．また, CPU 向け OpenACC は OpenMP と実行性能を比較し, GPU 向け OpenACC は CUDA 環境と実行性能を比較した．さらに, 同最適化結果を用いて JSS2 の計算機である FX100 と Ivy Bridge と実行性能の比較を行った．本研究で得られた結論を以下に示す．

- 共役勾配法のカーネルコードに対して前述の OpenACC 向け最適化を用いることで CPU は全ケースにおいて 4 倍以上, GPU は 7000 倍以上の高速化に成功した．よって, クラスのメンバ変数の取り出しや`__restrict__` 修飾子の付与などのソースコード最適化が有効な手法であると考えられる．しかしながら, CUDA 環境の実行性能と比較すると OpenACC は 20% 程度実行性能が劣るということがわかった．
- FX100 環境と OpenACC を用いた Skylake 環境と比較すると FX100 のほうが 1.2 倍程度高速であることがわかった．しかしながら, P100 環境の実行性能を比較すると 3.6 倍以上の高速化が可能であることがわかった．

以上より, OpenACC はソースコード最適化が必要ではあるが, 共役勾配法のカーネルコードに対し有効な手法であると云える．

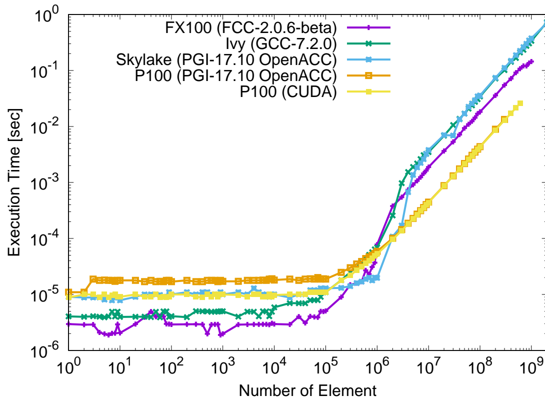

図 20 各計算機環境におけるソースコード最適化後のベクトル同士の加算の計算時間の比較．ただし，OpenMP で FX100 は 32 スレッド，Ivy Bridge は 12 スレッド使用する．

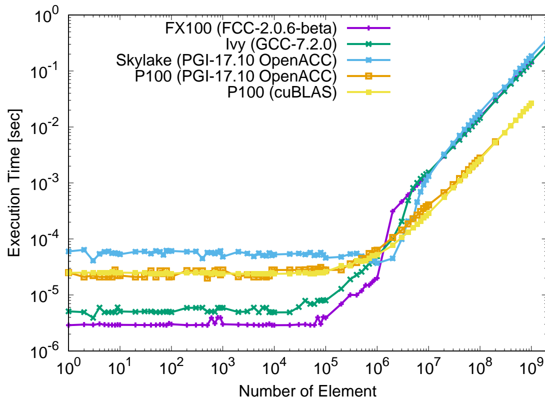

図 21 各計算機環境におけるソースコード最適化後の内積の計算時間の比較．ただし，OpenMP で FX100 は 32 スレッド，Ivy Bridge は 12 スレッド使用する．

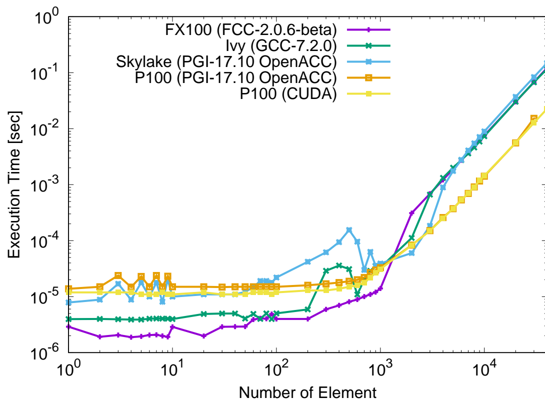

図 22 各計算機環境におけるソースコード最適化後の密行列ベクトル積の計算時間の比較．ただし，OpenMP で FX100 は 32 スレッド，Ivy Bridge は 12 スレッド使用する．

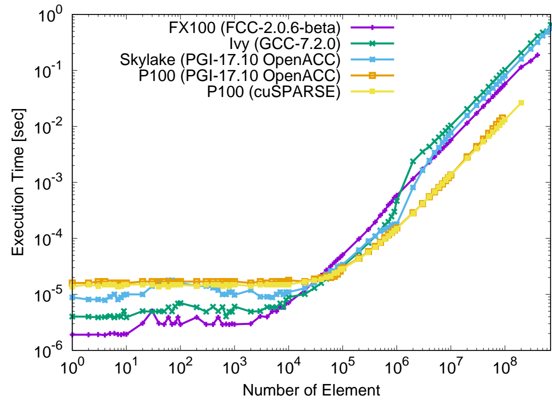

図 23 各計算機環境におけるソースコード最適化後の CRS 形式の疎行列ベクトル積の計算時間の比較．ただし，OpenMP で FX100 は 32 スレッド，Ivy Bridge は 12 スレッド使用する．

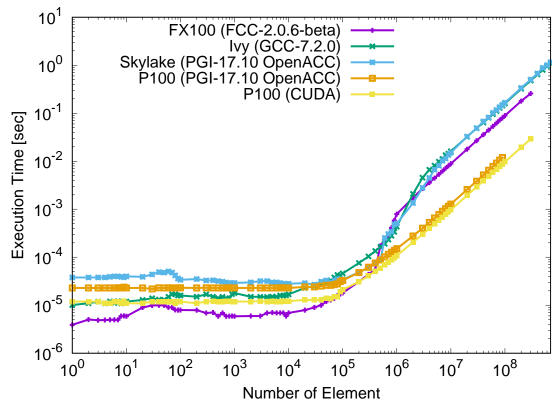

図 24 各計算機環境におけるソースコード最適化後の ELL 形式の疎行列ベクトル積の計算時間の比較．ただし，OpenMP で FX100 は 32 スレッド，Ivy Bridge は 12 スレッド使用する．

謝辞 数値計算の実行に当たっては宇宙航空研究開発機構スーパーコンピュータ JAXA Supercomputer System Generation 2 (JSS2) を用いた [9].

参考文献

[1] 中島研吾, 大島聡史, 塙敏博, 星野哲也, 伊田明弘: ICCG 法ソルバーの Intel Xeon Phi 向け最適化, 情報処理学会研究報告 Vol.2016-HPC-157 No.16.

[2] 星野哲也, 大島聡史, 塙敏博, 中島研吾, 伊田明弘: OpenACC を用いた ICCG 法ソルバーの Pascal GPU における性能評価, 情報処理学会研究報告 Vol.2017-HPC-158 No.18.

[3] 星屋哲也, 丸山直也, 松岡聡: 大規模流体アプリケーションの CUDA・OpenACC への移植性の評価, 情報処理学会研究報告 Vol.2012-HPC-135 No.42.

[4] 藤野清次, 張紹良: 反復法の数理, 株式会社朝倉書店, pp23-82, 1996.

[5] Nathan Bell, Michaerl Garland: Effective Sparse Matrix-Vector Multiplication on CUDA, NVIDIA Technical Report NVR-2008-004.

[6] : OpenACC Programming ディレクティブによるプログラミング, https://www.softek.co.jp/SPG/Pgi/OpenACC/.

[7] 生野壮一郎, 川口優樹, 上田信行: GPU プログラミングのツボと GPGPU の実用例, 可視化情報学会誌 巻:31 号.

[8] : NVIDIA DEVELOPER ZONE CUDA TOOLKIT DOCUMENTATION, http://docs.nvidia.com/cuda/index.html.

[9] : JAXA Supercomputer System Generation 2 (JSS2), https://www.jss.jaxa.jp.