チュートリアル

科学技術計算の分野で始まったGPU（Graphic Processing Unit）による汎用計算は、製造業、医療、金融など様々な分野へ広まり、モバイル、組み込みの分野でも使われ始めています。当初は専用言語を使う必要がありGPUによる汎用計算は気軽に始められるものではありませんでしたが、用途が広がると同時に、従来の言語でGPU向けにプログラミングする環境が整ってきました。OpenACCはその中の一つで、既存のプログラムにディレクティブと呼ばれるヒント情報を追加するだけで、コンパイラが自動でGPUコードを生成する開発手法です。今回のチュートリアルでは、全4回のシリーズとして、このOpenACCを使ってアプリケーションを効率良くGPUで加速する方法を解説します。

OpenACCで始めるGPUコンピューティング：データ転送の最適化

成瀬 彰

1　はじめに

前回は、我々が推奨するOpenACCによるアプリケーション並列化サイクル（図1）の前半、「1）ホットスポットの特定」と「2）ループの並列化」、この2ステップに関して説明した。例題としたヤコビ反復法のサンプルコードは、この2ステップに従って、計算負荷の重い部分をGPU実行できるようになった。CUDA等のアクセラレータ言語に頼ることなく、既存のC / Fortranソースコードにディレクティブを挿入するだけでGPUを使用できる。これがOpenACCのパワーである。

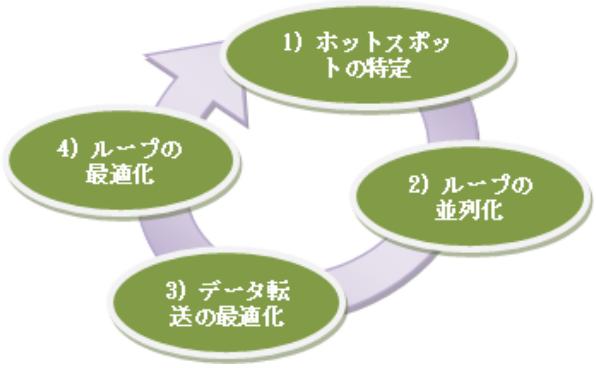

図1　OpenACCによるアプリケーション並列化のサイクル

しかし、前半の2ステップを適用した状態では、GPU実行がCPU実行よりも遅いことも珍しくない。多くのGPU搭載システムは、CPUとGPUでメモリが分離している（図2）。GPUでプログラムを実行するには、実行に必要なデータがGPUメモリ上に載っている必要がある。そのため、GPU実行の前後で、CPUメモリとGPUメモリの間でデータ転送が必要となる。CPUとGPUは、多くの場合、PCIe[1]で接続されているが、そのバンド幅は、GPUメモリのバンド幅と比べて一桁以上遅い。CPUとGPU間のデータ転送量が多いと、その転送にかかる時間が性能のボトルネックとなる。CPU・GPU間のデータ転送は可能な限り減らさねばならない。

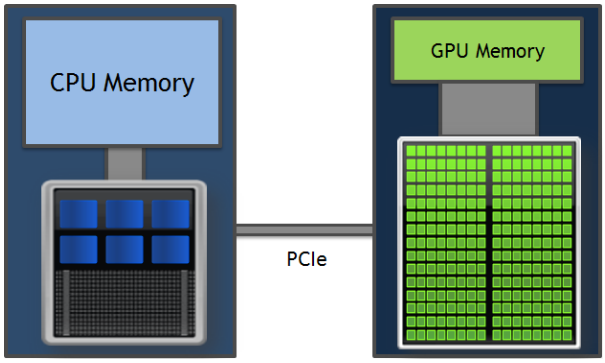

図2　GPUシステムの構成

続きはWebで
日本計算工学会誌「計算工学（Vol.21, No.4）」HP:
http://www.jsces.org/Issue/Journal/

筆者紹介

なるせ あきら
1996年、名古屋大学大学院工学系研究科修士課程修了（情報工学専攻）。同年、富士通研究所に入社、大規模サーバ開発、HPCシステム開発など、様々なプロジェクトに参加、関連するハードウェアやソフトウェアの研究開発に従事。2009年、山下記念研究賞受賞。2010年、SACSIS2010最優秀論文賞受賞。2013年、NVIDIAに参加、シニアデベロッパーテクノロジーエンジニアとして様々なアルゴリズムとアプリケーションのGPU向け並列化に従事、GPUコンピューティングの普及に努めている。

今回は、OpenACCによるアプリケーション並列化のサイクルの3番目、「3）データ転送の最適化」を説明してゆく。

2　データ転送の最適化

ループの並列化（GPU化）がある程度進んだら、次に取り組むのはデータ転送の最適化である。ループの並列化のステップでは、Kernels / Parallelコンストラクトを使ってプログラム内の並列化可能領域をコンパイラに指示した。このとき、コンパイラは指示領域の計算を並列化するだけでなく、計算に必要なデータをCPU・GPU間で転送する処理を追加する。並列領域に入る前に、計算に必要な入力データをCPUからGPUに送り、並列領域から出た後に、CPUで使われる計算結果をGPUからCPUに戻す。コンパイラはソースコードを解析して、これらの処理を自動で追加する。

しかし、コンパイラが選択・追加したデータ転送処理が、常に最適（実行時間が短い）とは限らない。計算の処理内容が複雑な場合は、不要なデータをCPU・GPU間で転送しているかもしれない。場合によっては、必要なデータ転送が行われていないかもしれない。また、データ転送に関するコンパイラの解析範囲は、各並列領域に閉じているので、異なる並列領域で同じデータを使用している場合でも、それを冗長に重複して転送しているかもしれない。

OpenACCでは、CPU・GPU間のデータ転送を制御するディレクティブとして、Dataコンストラクト、Dataクローズ、Updateディレクティブ等が用意されている。以下、それぞれの特徴や使い方を説明する。

2.1　データ領域（Dataコンストラクト）

Dataコンストラクトは、複数の並列領域にまたがるデータ共有を促進するディレクティブである。Dataコンストラクトで括られた部分はデータ領域と呼ばれる。以下、FortranとC / C++それぞれの、Dataコンストラクトの使用例をコードサンプル（以下CS）1と2に示す。

```
[Fortran]
   !$acc data
   !$acc parallel loop
   do i=1,N
      x(i)=0
      y(i)=i
   enddo
   !$acc end parallel
   !$acc parallel loop
   do i=1,N
      y(i)=y(i)+2.0*x(i)
   enddo
   !$acc end parallel
   !$acc end data
```

CS 1：dataコンストラクト使用例（Fortran）

```
[C/C++]
  #pragma acc data
  {
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      x[i] = 0.0;
      y[i] = i;
    }
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      y[i] += 2.0*x[i];
    }
  }
```

CS 2：dataコンストラクト使用例（C / C++）

この例では、dataコンストラクトを追加することで、2つの並列領域間で、配列xとyを共有することが可能となる。Dataコンストラクトが無い場合、2つの並列領域の間で、CPU・GPU間で配列xとyのデータ転送が行われるが、CS 1、CS 2のようにDataコンストラクトを追加してデータ領域を指示することで、この冗長なデータ転送を取り除くことができる。ただし、この状態では、どのデータをどのように転送するのかは、依然としてコンパイラのコード解析能力に依存しており、まだ、最適なデータ転送方法が選択されることが保証された訳ではない。

なお、データ領域の開始点と終了点は、同一スコープ、つまり、同一関数内、同一サブルーチン内になければいけない。データ領域内から関数・サブルーチンを呼び出すことは可能であり、その場合は、呼び出し先の関数・サブルーチンもデータ領域内にいる関数・サブルーチンとみなされる。また、複数のデータ領域をDataコンストラクトで纏めて、上位層のデータ領域を作ること、つまり、階層的なデータ領域の作成も可能である。

2.2　Dataクローズ

Dataクローズは、どの配列を、どのタイミングで、どのように転送するのかを、プログラマーがコンパイラに指示するときに使用するディレクティブである。Dataクローズは単独ディレクティブではなく、Data / Kernels / Parallelコンストラクトに追加する形で使用する（Kernel / Parallelコンストラクトが形成する並列領域は、暗黙的なデータ領域なので、dataクローズを追加できる）。以下に、OpenACCでサポートされているDataクローズの名前と意味を示す。

・**copy**：Read / Write配列に使用する。
データ領域に入るときに、GPUメモリ上に配列スペースを確保し（**ALLOC**）、配列データをGPUに送る（**DATA-IN**）。データ領域から出るときに、配列データをHostに戻し（**DATA-OUT**）、GPUメモリから配列スペースを削除する（**FREE**）。

・**copyin**：Read-Only配列に使用する。
データ領域に入るときに、GPUメモリ上に配列スペースを確保し(**ALLOC**)、配列データをGPUに送る(**DATA-IN**)。データ領域から出るときに、GPUメモリから配列スペースを削除する(**FREE**)。
・**copyout**：Write-Only配列に使用する。
データ領域に入るときに、GPUメモリ上に配列スペースを確保する(**ALLOC**)。データ領域から出るときに、配列データをHostに戻し(**DATA-OUT**)、GPUメモリから配列スペースを削除する(**FREE**)。
・**create**：Temporal配列に使用する。
データ領域に入るときに、GPUメモリ上に配列スペースを確保する(**ALLOC**)。データ領域から出るときに、GPUメモリから配列スペースを削除する(**FREE**)。
・**present**：既にGPUメモリに載っている配列、主に、上位のデータ領域で既にデータ転送方法が指定済みの配列に使用する。動作は発生しないが、その配列がGPUメモリ上に存在しないとエラーになる。
・**deviceptr**：OpenACCのメモリ管理の枠外で、GPUメモリ上に確保された配列に対して使用する。OpenACCを、CUDA等の他プログラミングモデルと併用するときに使用することがある。

Dataクローズの一つ、copyの使用例をCS 3と4に示す。この例では、2つの並列領域にまとめたデータ領域に対して、配列xとyの「copy」制御を指示している。従って、このデータ領域に入るときに配列xとyのデータがGPUに送られ、データ領域から出るときに配列xとyのデータがCPUに戻される。

```
[Fortran]
   !$acc data copy(x, y)
   !$acc parallel loop
   do i=1,N
      x(i)=0
      y(i)=I
   enddo
   !$acc end parallel
   !$acc parallel loop
   do i=1,N
      y(i)=y(i)+2.0*x(i)
   enddo
   !$acc end parallel
   !$acc end data
```

CS 3：dataクローズの使用例(Fortran)

```
[C/C++]
  #pragma acc data copy(x, y)
  {
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      x[i] = 0.0;
      y[i] = i;
    }
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      y[i] += 2.0*x[i];
    }
  }
```

CS 4：dataクローズの使用例(C / C++)

上説のdataクローズに加え、OpenACC 1.0と2.0には、present_or_*(*の部分にはcopy、copyin、copyout、createが入る)というdataクローズが存在する。例えばpresent_or_copyの場合、まず、その配列がGPUメモリ上に存在するかどうかのチェックが行われる。もし配列が存在すれば何も行われないが、もし配列がなければcopy動作が行われる。ただし、OpenACC 2.5から、presentチェックが全てのdataクローズの基本動作となったため、OpenACC 2.5対応のコンパイラならpresent_or_* クローズを使用する必要はない(互換性維持のため、OpenACC 2.5以降でもpresent_or_*クローズは使用できる)。

配列形状の指定：

Dataクローズに配列名のみを記述すると、その配列の全データが制御の対象となる。Fortranでは、配列形状がプログラムに明記されるのが一般的なので、多くの場合、コンパイラは普通に配列形状が分かる。しかし、C / C++では、配列はポインターとして渡されることが多く、コンパイラは配列形状が分からないことが多い。配列形状・サイズが分からないと、データ転送量を決定できない。また、配列の全データではなく、その一部分しか使わないプログラムの場合、配列の全データをCPU・GPU間で転送するのは、リソースの浪費である。

OpenACCでは、配列のサイズや、Dataクローズの制御対象とする部分配列を記述することができる。FortranとC / C++、それぞれの配列形状の記述方法を以下に示すが、FortranとC / C++では記述方法が異なるので、注意してほしい。

・Fortran：**array(start:end)**
startは配列の最初の要素番号で、endは最後の要素番号を示す。従って、要素数はend – start + 1となる。
・C / C++：**array[start:count]**
startは配列の最初の要素番号で、countは要素数を示す。従って、配列の最後の要素番号はstart + count – 1となる。

```
[Fortran]
   !$acc data create(x(1:N)) copyout(y(1:N))
   !$acc parallel loop
   do i=1,N
      x(i)=0
      y(i)=I
   enddo
   !$acc end parallel
   !$acc parallel loop
   do i=1,N
      y(i)=y(i)+2.0*x(i)
   enddo
   !$acc end parallel
   !$acc end data
```

CS 5：最適なdataクローズの使用例（Fortran）

ここまでの情報を踏まえて、CS 3と4のケースを再考する。前提として、配列xとyのサイズはNより大きく、配列xはtemporal配列（GPU上でしか使われない配列）とする。Temporal配列は、CPU・GPU間でデータ転送する必要がないので、Dataクローズはcreateを選択するのが良い。配列yは、元の値を参照していない、つまり、CPUからGPUにデータを送る必要がないので、copyoutを使えば良い。配列xとyは、どちらもN個の要素しか使用されないので、配列形状を指定すると、転送量を削減できる。この方針に基づいてdataクローズを最適化した結果をCS 5と6に示す。

```
[C/C++]
  #pragma acc data create(x[0:N]) copyout(y[0:N])
  {
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      x[i] = 0.0;
      y[i] = i;
    }
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      y[i] += 2.0*x[i];
    }
  }
```

CS 6：最適なdataクローズの使用例（C / C++）

2.3　非構造データ区間（enter / exit dataディレクティブ）

Dataコンストラクトで作成できるデータ領域は、その開始点と終了点が同一スコープ内になければならない。しかし、プログラム・配列によっては、データをGPUに転送したいタイミング（開始点）と、データをGPUから回収したいタイミング（終了点）が、同一スコープにないこともあるだろう。そのような場合に便利なのが、OpenACC 2.0から採用されたenter / exit dataディレクティブである。CS 7と8に、その使用例を示す。

```
[Fortran]
  subroutine init_array
    ...
    allocate(array(N))
    read(fn,*) array
!$acc enter data copyin(array(1:N))
    ...
  end subroutine

  subroutine fin_array
    ...
!$acc exit data copyout(array(1:N))
    write(fn,*) array
    deallocate(array)
    ...
  end subroutine
```

CS 7：enter / exit dataディレクティブの使用例（Fortran）

Enter dataディレクティブで指定された配列は、そのタイミングでGPU上に必要なメモリスペースが確保される。使えるdataクローズは、createとcopyinである。Copyinが使われた場合は、同時にデータがCPUからGPUに送られる。

Exit dataディレクティブで指定された配列は、そのタイミングで、GPU上からその配列のメモリスペースが削除される。使えるdataクローズは、copyoutとdelete（exit dataディレクティブ専用）である。Copyoutが使われた場合は、同時にデータがGPUからCPUに戻される。

```
[C/C++]
  void init_array()
  {
    ...
    array = (char*)malloc(N);
    read(fd, array, N);
#pragma acc enter data copyin(array[0:N])
    ...
  }

  void fin_array()
  {
    ...
#pragma acc exit data copyout(array[0:N])
    write(fd, array, N);
    free(array);
    ...
  }
```

CS 8：enter / exit dataディレクティブの使用例（C / C++）

2.4　Updateディレクティブ

GPUからのみ使われる配列（CPUで使われない配列）を、プログラムの上位層の関数・サブルーチンから、GPUメモリに常駐させるのは、性能的には良いアプローチである。しかし、基本的にはGPUからのみ使われる配列でも、例えば、ファイルIOやノード間通信のために、時々CPUから参照・更新せざるを得ないこともあるだろう。このような場合に有用なのがUpdate

ディレクティブである。

Updateディレクティブは、GPUメモリ上にある配列の値を、データ領域内のどこからでも、CPU上の該当部分へ転送することができる。反対に、CPU上で更新した配列の値を、GPU上の該当部分に転送することもできる。

Updateディレクティブは、Selfクローズ、もしくは、Deviceクローズと併用する。Update selfディレクティブでは、指定した配列の値がGPUからCPUへ転送され、Update deviceディレクティブでは、反対に、CPUからGPUへ転送される。文字通り、Update selfではセルフ(Host、CPU)が更新され、Update deviceではデバイス(GPU)が更新される、と覚えれば良いだろう。

Updateディレクティブの使用例を、CS 9と10に示す。この例では、データ領域内に3つのループがあるが、最初と最後のループはGPUで実行、2番目のループはCPUで実行するようになっている。最初のループの実行が終わると、GPU上の配列xは更新されるが、その更新はHost上の配列xには反映されない。このまま2番目のループをCPU上で実行すると、計算結果がおかしくなる。この例では、ここにUpdate selfディレクティブが入っており、配列xの値がGPUからCPUへと転送されるので、2番目のループをCPU上で実行しても問題ない。同様に、2番目のループの後には、Update deviceディレクティブが入っており、配列xの値がCPUからGPUへと転送されるので、最後のループも問題なくGPUで実行できる。

```
[Fortran]
  !$acc data copy(x(1:N))
  !$acc parallel loop
  do i=0,N
    x(i) = x(i) + ...  # GPU
  end do
  !$acc end parallel
  !$acc update self(x(1:N))

  do i=0,N
    x(i) = x(i) + ...  # CPU
  end do
  !$acc update device(x(1:N))

 !$acc parallel loop
  do i=0,N
    x(i) = x(i) + ...  # GPU
  end do
  !$acc end parallel
  !$acc end data
```

CS 9：updateディレクティブの使用例(Fortran)

```
[C/C++]
  #pragma acc data copy(x[0:N])
  {
    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      x[i] += ...;  /* GPU */
    }
    #pragma acc update self(x[0:N])

    for (i=0; i<N; i++) {
      x[i] += ...;  /* CPU */
    }
    #pragma acc update device(x[0:N])

    #pragma acc parallel loop
    for (i=0; i<N; i++) {
      x[i] += ...;  /* GPU */
    }
  }
```

CS 10：updateディレクティブの使用例(C / C++)

なお、Updateディレクティブは、Enter dataディレクティブを使ってGPUメモリ上に生成した配列に対しても適用可能である。もちろん、配列形状を指定して、配列の一部をUpdateディレクティブで転送させることもできる。

2.5　ヤコビ反復法のデータ転送の最適化

ここまで説明したデータ転送に関連するディレクティブを使って、ヤコビ反復法のデータ転送を最適化してみよう。

現状確認：

まずは現状の把握である。前回のチュートリアルで、2)ループの並列化まで適用したヤコビ反復法のコードを、pgprof[2]でプロファイリングした結果を以下に示す(関連部分のみ抽出)。

```
$ pgprof ./jacobi
...
==13890== Profiling application: ./jacobi
==13890== Profiling result:
Time(%)      Time   Calls .. Name
 49.73%  4.42237s     600 .. [CUDA memcpy HtoD]
 45.73%  4.06678s     600 .. [CUDA memcpy DtoH]
  2.57%  228.45ms     200 .. jacobi_37_gpu
  1.94%  172.80ms     200 .. jacobi_47_gpu
  0.03%  2.3767ms     200 .. jacobi_37_gpu_red
...
```

jacobi_*がGPU上の計算処理(GPUカーネル)を示しているが、この3つを合わせても、全実行時間に占める割合は5%未満である。実行時間の大半、95%以上を占めているのは、[CUDA memcpy HtoD]と[CUDA memcpy DtoH]である(HtoDはCPUからGPUへのデータ転送、DtoHはGPUからCPUへのデータ転送)。明らかに、GPU上の計算ではなく、CPU・GPU間のデータ転送が性能ボトルネックになっている。データ転送の最適化が必要な状況である。

参考までに、NVIDIA Visual Profiler(nvvp) [3]による解析結果(時系列情報)を図3に示す(nvvpを使うと、統計情報だけでなく、このようにイベント間の時系列関係を確認できる。nvvpには他にも様々な機能が存在するが、ここではその説明は割愛する)。これ見ると、各GPUカーネル実行の前後で、データ転送が行われていることが分かる。

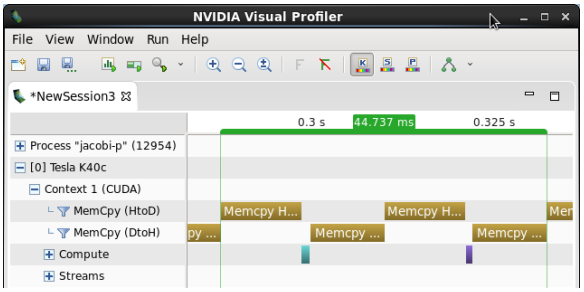

図3　データ転送を最適化する前のヤコビ反復法の時系列プロファイリング結果

ディレクティブ追加：

ヤコビ反復法のサンプルコードのメインループには、2つの並列領域が含まれているが、その間でデータが共有されていないことが、ここでは一番の問題である。その解決には、Dataコンストラクトを使って、この2つの並列領域を含むデータ領域を設定すればよい。このケースでは、whileループの外側にデータ領域を設定するのが適切である。Whileループの内側にデータ領域を設定することもできるが、その場合は、whileループのiteration間でのデータ共有のチャンスを失うことになる。

```
[Fortran]
  !$acc data create(Anew) copy(A)
  do while (error > TOL .and. iter < MAX_ITER)
    error = 0.0
    !$acc parallel loop reduction(max:error)
    do j = 2, N-1
      !$acc loop reduction(max:error)
      do i = 2, M-1
        Anew(i,j) = 0.25 * &
          (A(i,j-1) + A(i,j+1) + A(i-1,j) + A(i+1,j))
        error = max(error, abs(Anew(i,j) - A(i,j)))
      end do
    end do
    !$acc end parallel

    !$acc parallel loop
    do j = 2, N-1
      !$acc loop
      do i = 2, M-1
        A(i,j) = Anew(i,j)
      end do
    end do
    !$acc end parallel

    if (mod(iter,100) == 0) write(*,*) iter,, error
    iter = iter + 1
  end do
  !$acc end data
```

CS 11：データ転送を最適化したヤコビ反復法(Fortran)

```
[C/C++]
  #pragma acc data create(Anew) copy(A)
  while (error > TOL && iter < MAX_ITER) {
    error = 0.0;

    #pragma acc parallel loop reduction(max:error)
    for (j = 1; j < N-1; j++) {
      #pragma acc loop reduction(max:error)
      for (i = 1; i < M-1; i++) {
        Anew[j][i] = 0.25 *
          (A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j+1][i]);
        error = fmaxf(error, fabsf(Anew[j][i] - A[j][i]));
      }
    }

    #pragma acc parallel loop
    for (j = 1; j < N-1; j++) {
      #pragma acc loop
      for (i = 1; i < M-1; i++) {
        A[j][i] = Anew[j][i];
      }
    }

    if (iter % 100 == 0) printf("%d, %f\n", iter, error);
    iter++;
  }
```

CS 12：データ転送を最適化したヤコビ反復法(C / C++)

次に、各配列をどう制御するのが適切かを考える。ここでは配列Anewと配列Aの2つの配列が存在する。まず、配列Anewについて考える。whileループの各iterationで配列Anewに出力された値は、その次のiterationで使われることがない。また、whileループの最終iterationで出力された値が、whileループを出た後で使われることもない。つまり、配列Anewは、このwhileループ内に閉じたTemporal配列であり、dataクローズにはcreateを適用するのが適切である。次に、配列Aだが、こちらはwhileループに入る前の値が参照されており、whileループ内で配列Aに出力された値は、whileループから出た後も必要である。従って、配列Aにはcopyを適用するのが適切である。この考察に基づいてディレクティブを追加したコードサンプルをCS 11と12に示す。

性能確認：

ソースコードを変更したら、性能確認である。Pgprofのプロファイリング結果は以下の通りで、狙い通り、データ転送時間を大幅に削減できている。

```
$ pgprof ./jacobi
...
==14005== Profiling application: ./jacobi
==14005== Profiling result:
Time(%)      Time   Calls .. Name
 53.72%  228.59ms     200 .. jacobi_38_gpu
 40.60%  172.76ms     200 .. jacobi_48_gpu
  2.65%  11.265ms     204 .. [CUDA memcpy HtoD]
  2.48%  10.552ms     205 .. [CUDA memcpy DtoH]
  0.54%  2.3182ms     200 .. jacobi_38_gpu_red
...
```

Nvvpによる解析結果を図4に示す。この結果からも、GPU上での計算処理間のデータ転送時間が大幅に削減できていることが分かる。転送時間は短くなっているが、依然としてデータ転送は行われている。これは、変数errorに関わるデータ転送である。1つめのループ

では、変数errorの縮約計算（reduction）がGPU上で行われているが、その値はwhileループの各iterationでCPUに回収する必要がある。Nvvpを使うと、このようなプログラムの挙動も比較的簡単に推測・確認できるので、挙動確認にはnvvp使用を推奨する。

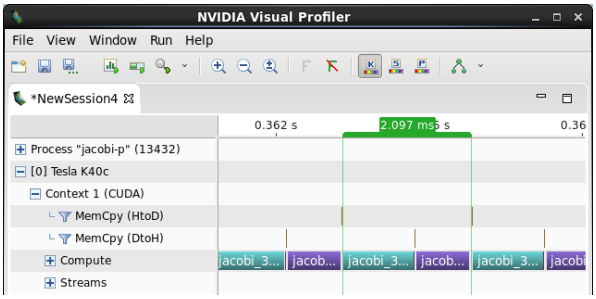

図4　データ転送を最適化した後のヤコビ反復法の時系列プロファイリング結果

3　おわりに

今回は、我々が推奨するOpenACCによるアプリケーション並列化サイクルの3番目のステップ、3）データ転送の最適化を説明した。サンプルとして用いたヤコビ反復法は、このステップの適用後に、GPU使用に相応しい性能が得られるようになった。このように、2）ループの並列化を適用してGPU実行したら、むしろ性能が低下、しかし、3）データ転送の最適化を適用することで、性能が大幅に向上という事例を、我々はこれまでに多数見てきた。同様の状況になっても、あきらめずに、3）データ転送の最適化に取り組んで欲しい。

次回は、OpenACCによるアプリケーション並列化サイクルの最後のステップ、4）ループの最適化について説明する。

■参考文献

[1]　PCIe: https://ja.wikipedia.org/wiki/PCI_Express
[2]　pgprof: http://www.pgroup.com/doc/pgprofug.pdf
[3]　NVIDIA Visual Profiler (nvvp):
https://developer.nvidia.com/nvidia-visual-profiler