Vol.2015-HPC-148 No.14
2015/3/2

自動車のモデルベース開発におけるシミュレーションのGPGPUによる高速化

窪田 昌史[1,a)]　國光 修司[2]　寺岡 陽一[2]　矢野 康英[2]　北村 俊明[1]

概要：複雑化，高度化した自動車を短期間かつ低コストで開発するために，部品の挙動をモデル化したシミュレーションが多用されるようになってきている．本発表では，エンジンに接続される吸排気管内のシミュレーションを行うプログラムを，GPU を用いて高速化する事例について報告する．

Performance Improvement of Simulations for Model Based Development of Automotives by GPGPU

1. はじめに

自動車の開発においては，モデルベース開発 (MBD; Model Based Development) が極めて重要になりつつある [1][2][3]．これは，複雑化，高度化した自動車を短期間かつ低コストで開発するために，開発そのものを机上で効率良く行うことが求められてきているためである．MBD では，部品の挙動をモデル化したシミュレーションが多用されるようになってきている．本発表では，エンジンに接続される吸排気管内のシミュレーションを行うプログラムを取り上げ，その高速化について報告する．

本シミュレーションプログラムの実行時間の短縮のため，我々は，プログラムの実行時間，所要データサイズ，モデルに内在する並列性などを検討し，汎用 PC のマルチコアプロセッサ上での並列化と，GPU 上での並列化を進めることとした．

本報告では，GPU での性能チューニングにおいて必要となったメモリ階層を考慮したデータ配置の最適化を中心に述べる．以下，2 章で吸排気管シミュレーションとその並列性を説明し，3 章では GPU 上の実装方法について述べる．4 章で GPU での実行結果を汎用の CPU での実行結果と比較しながら示し，最後に 5 章でまとめとする．

[1] 広島市立大学
Hiroshima City University
[2] マツダ株式会社
Mazda Motor Corporation
[a)] kubota@hiroshima-cu.ac.jp

2. 吸排気管シミュレーション

本章では，まず，エンジン吸排気管のシミュレーションプログラムが採用している吸排気管のモデルについて説明し，次にモデルに内在する並列性の抽出について議論する．

2.1 モデル

4 ストロークエンジンでは，ピストンがシリンダ内を 2 往復してクランク軸が 2 回転するまでが 1 サイクルとされる．サイクル毎に，エンジンでは吸入，圧縮，燃焼，排気の 4 行程が繰り返される．本シミュレーションでは，微小時間のタイムステップに区切って，エンジンへの吸気管および排気管内の圧力，温度などを求める処理を繰り返す．

吸気管および排気管は複雑な形状で構成され，複数の管の合流などもある．本シミュレータでは，吸気管，排気管それぞれを多数に分割された**管** *(pipe)* が結合されたものとして扱っている．分割された各管は，さらに同じ長さの微小の区間に分割される．本報告では，この微小区間を**メッシュ**と呼ぶ．シミュレーションを高速に行うため，メッシュ内では，圧力や温度などは等しいとみなす 1 次元モデルを採用している [3]．このモデルの数値シミュレーションを行うプログラムは C++言語で記述され，Weighted Aferage Flux (WAF)[4] と呼ばれるスキームを採用している．

本シミュレータでは，吸気管と排気管をあわせて管を最大 500 まで，各管は最大 31 個のメッシュまで扱えるようになっている．

2.2 並列性の抽出

各管が N 個のメッシュに分割されているとする．この N 個のメッシュの各タイムステップ毎の処理は以下に述べるように，並列実行可能である．

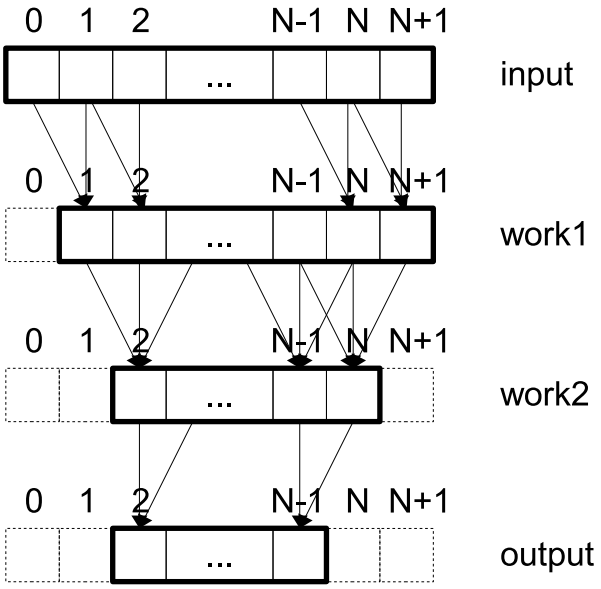

図 1 吸排気管のシミュレーション
Fig. 1 Simulation of Intake and Exhaust Pipes

```
if (fabs(work1[i]) < val) {
  work2[i] = (work1[i] + work1[i-1])/2.0;
} else {
  work2[i] = (work1[i] + work1[i+1])/2.0;
}
```

図 2 条件分岐
Fig. 2 Conditional Branch

メッシュ数 N の管のシミュレーションは，各タイムステップで**図 1** に示すような処理が行われる．入力データは，着目している管の両端に隣接して接続されている管のメッシュ 1 個づつのデータを含め，N+2 個のメッシュのデータである．入力データの i-1 番目と i 番目の要素を用いて work1 データの i 番目の要素を求める処理を行う．i=1 から N+1 のこれらの処理は並列に実行可能である．同様に，work1 の i-1,i,i+1 番目の要素を用いて work2 の i 番目の要素を求める処理も並列に実行できる．なお，この処理は**図 2** のように，条件分岐によって i-1 番目と i+1 番目の要素が選択的に使用される処理を含んでいる．最終的に，出力データとして i=2 から N-1 までの要素を work2 から求める処理も並列に実行可能である．

各管の i=2 から N-1 までの要素のデータや，エンジンなどの吸排気管に接続されている他の部品のシミュレーションから得られるデータを用いて，各管の次のタイムステップの i=0 から N+1 までの要素の入力データが用意される．

先に述べたように，本シミュレーションでは，吸排気管は 500 本, メッシュは 31 個まで扱え，各メッシュを並列に処理できることから，シミュレーション全体では最大 15,500 程度の並列度を持つことになる．

GPU に搭載されている多数のプロセッサコアの並列度を活用するため，GPU 上では吸気管，排気管ごとに，全メッシュを同時に処理するようなモデルとした．つまり，吸気管，排気管ともに，分割された管をすべて一列に連結し，その管にすべてのメッシュが並んでいるとした仮想的な吸気管，排気管に対してシミュレーションを行うように，GPU 向けにアルゴリズムを変更した．

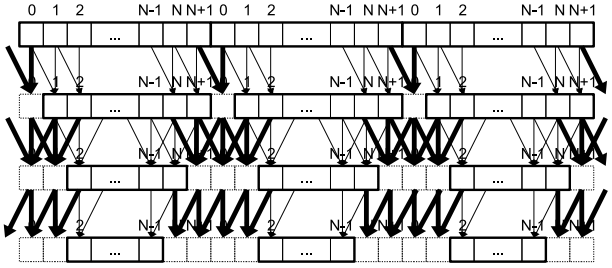

図 3 連結された管における冗長計算
Fig. 3 Redundant Processing on Connected Pipes

ただし，図 1 に示すように，処理が進むにつれて i=0,1 や i=N,N+1 の要素に対する処理は不要となるが，GPU 向けに変更したアルゴリズムでは，これらの要素に対する冗長な処理が行われてしまう．**図 3** は、仮想的に連結された管の一部分を取り出し，隣接メッシュデータの参照関係を表したものである．この図中，太い矢印でデータの流れを表しているのが冗長な処理である．冗長な処理を行っても，各管の管端以外の i=2 から N-1 までの要素の処理結果には影響しない．

ここで，これらの冗長な処理を行わないように分岐処理で導入することが考えられる．しかし，GPU では一般に分岐処理によるオーバヘッドが大きいことが知られている．そのため，この冗長な処理を含めて実行することとした．

3. GPU への実装

前節述べた並列化の方針に従い，C++言語で記述されたシミュレーションプログラムを NVIDIA 社製の GPU 向けに CUDA[5] を用いて記述している．また，性能を比較するため，同様の並列化の方針に従って OpenMP[6] を用いた実装も用意している．

3.1 GPU の記憶階層

シミュレーションプログラムを CUDA で実装する際に，GPU の記憶階層を意識した最適化を行っている．GPU の記憶階層は，GPU のプロセッサコアに近い方から順に以下のようなものがあり，GPU に搭載される大量のプロセッ

サコアを並列に実行させるには，これらのメモリ階層へのアクセスを工夫することが重要になる．

- レジスタ
- キャッシュ/シェアードメモリ
- グローバルメモリ
- ホスト PC のメインメモリ

GPU から直接アクセスできるメモリは，大容量のグローバルメモリやシェアードメモリである．一方，ホスト PC のメインメモリとの間で転送可能なのは，グローバルメモリである．3.2 節で転送が必要なデータの分類について説明する．

この転送のレイテンシを隠蔽して高速化を図るために，**ストリーム** *(stream)* を用いて転送と GPU の処理をオーバーラップさせることが可能である．この最適化については 3.3 節で述べる．

また，GPU からグローバルメモリへのアクセスには，同時に実行されているスレッドが連続したアドレスのデータをアクセスする**コアレッシング** *(coalescing)* による高速化が重要となる．こちらの最適化については，3.4 節で述べる．

3.2 GPU とホスト PC 間のデータ転送

データ転送とカーネル関数をオーバラップさせて実行するために，カーネル関数でアクセスする配列変数を以下のように分類する．

- 定数からなる配列変数
- 各タイムステップ毎に，カーネル関数に対する入力として与えられる配列変数
- 各タイムステップ毎に，カーネル関数からの出力となる配列変数
- カーネル関数内でワーキングエリアとして使用される配列変数

これら 4 種類の配列変数に対し，GPU のグローバル変数領域をすべて割当てておく．シミュレーションの各タイムステップ毎では，

(1) 入力配列のホスト PC から GPU への転送
(2) カーネル関数の実行
(3) 出力配列の GPU からホスト PC への転送

が繰り返される．定数配列は，この繰り返しを行うループに入る前に，ホスト PC から GPU へ先に転送しておく．

3.3 ストリーム

各タイムステップ毎の繰り返しで，データ転送とカーネル関数の実行をオーバラップさせて同時に実行することで，データ転送の遅延を隠蔽することができる．吸気管と排気管の処理毎に，データ転送とカーネル関数の実行を分割する．以下のように，CUDA のストリーム機能を利用し，吸気管に関する処理を stream1, 排気管に関する処理を stream2 として実行すると，同じストリーム内の実行順序は保たれ，他のストリームとの間では，資源の競合が起こらない限り，データ転送，カーネル関数実行などが，同時に行われる．

(1) 吸気管入力データ転送 (stream1)
(2) 排気管入力データ転送 (stream2)
(3) 吸気管カーネル関数 (stream1)
(4) 排気管カーネル関数 (stream2)
(5) 吸気管出力データ転送 (stream1)
(6) 排気管出力データ転送 (stream2)

ゆえに，排気管の入力データ転送と吸気管のカーネル関数の実行が同時に行われることが期待できる．排気管のカーネル関数の実行と吸気管の出力データ転送の同時実行も同様である．

本シミュレーションでは，各タイムステップ毎に，管端にあたる配列要素や他の部品のシミュレータとの値の交換が行われる．このため，タイムステップ毎の処理を超えた実行のオーバラップは行わない．たとえば，排気管のカーネル関数実行中に，次のタイムステップの吸気管の入力データの転送を行ったり，排気管の出力データの転送中に，次のタイムステップの吸気管のカーネル関数の実行を行ったりすることはしない．

3.4 ループ交換と配列の次元の交換

シミュレーションで扱う配列変数の中には，**図 4** の例の `work11` や `input11` のように，吸排気管のメッシュに加えて，他の次元を持つものもある．

```
for (i=0; i<N_MESH; i++) {
  for (j=0; j<NJ; j++) {
    for (k=0; k<NK; k++) {
      work11[i][j][k]
        = input11[i][j][k] * input12[i];
    }
  }
}
```

図 4 3 重ループの例
Fig. 4 An Example of Triple Nested Loop

この例を GPU で実行する際に，各メッシュ毎の処理を行うループ i をスレッドに割り当てることが考えられる．しかし，配列要素へのアクセスが coalescing にはならないために性能低下が起こる．そこで，**図 5** のようにループ交換と配列の次元の交換を行って，アクセスが coalescing になるような最適化が望ましい．

なお，グローバルメモリ上に多次元配列を確保する場合には，コンパイル時に静的に配列のサイズが確定されていなければならない．ホスト PC から GPU のグローバルメモリ上に多次元配列を動的に確保することはできない．これは，CUDA のベースとなる C, C++, Fortran 言語の動

```
for (j=0; j<NJ; j++) {
  for (k=0; k<NK; k++) {
    for (i=0; i<N_MESH; i++) {
      work11[j][k][i]
        = input11[j][k][i] * input12[i];
    }
  }
}
```

図 5 ループ交換と配列の次元の交換を行った 3 重ループ

Fig. 5 The Triple Nested Loop where Lopps and Array Dimensions are Interchanged

的領域確保の制限によるものである．ゆえに，配列を動的確保する場合には，3 次元配列は 1 次元化され，添字の記述が複雑になる．**図 6** にカーネル関数の記述例を示す．このカーネル関数での配列のアクセスを coalescing にするためには，**図 7** のように書き換える必要がある．

```
i = blockDim.x * blockIdx.x + threadIdx.x;
for (j=0; j<NJ; j++) {
  for (k=0; k<NK; k++) {
    work11[i*NJ*NK+j*NK+k]
      = input11[i*NJ*NK+j*NK+k] * input12[i];
  }
}
```

図 6 カーネル関数中の多重ループの例

Fig. 6 An Example of Nested Loop in a Kernel Function

```
i = blockDim.x * blockIdx.x + threadIdx.x;
for (j=0; j<NJ; j++) {
  for (k=0; k<NK; k++) {
    work11[j*NK*N_MESH+k*N_MESH+i]
      = input11[j*NK*N_MESH+k*N_MESH+i]
        * input12[i];
  }
}
```

図 7 ループ交換と配列の次元の交換を行ったカーネル関数中の多重ループ

Fig. 7 The Nested Loop in the Kernel Function where Loops and Array Dimensions are Interchanged

4. 性能評価

本章では，**表 1** に示した吸排気管の総数が多いモデル 1 と，やや少ないモデル 2 の 2 つのモデルを用いてシミュレーションを実行した結果について述べる．モデル 1 と 2 では，吸気管と排気管を分割した総数はそれぞれ 500 と 300 であり，その半数が吸気管，残りの半数が排気管である．分割された各管のメッシュ数は，乱数によって 3 個から 31 個としている．すべての管の管端の重複メッシュを含めたメッシュ数は，モデル 1 と 2 でそれぞれ 9,758 個と 5,589 個となっており，吸排気管にほぼ均等な個数のメッシュが存在する．

オリジナルのシミュレーションプログラムの行数は C++ 言語で約 1,000 行である．2 章で述べたように，並列性を抽出するために，分割された管を連結し，冗長な処理を行うようにプログラムを変更している．

表 1 吸排気管の仕様

Table 1 Specification of Intake and Exhaust Pipes

	モデル 1	モデル 2
管数	500	300
メッシュ/管	3-31	3-31
総メッシュ数	9,758	5,589
（吸気管）	4,951	2,669
（排気管）	4,807	2,920

このプログラムを，OpenMP を用いて並列化し，通常のマルチコアプロセッサ上で実行したものと，CUDA を用いて記述し，GPU 上で実行したものとを比較する．使用した GPU と，比較のために用いたマルチコアプロセッサ搭載 PC の仕様は**表 2** のとおりである．

表 2 GPU とマルチコアプロセッサ搭載 PC の仕様

Table 2 Specification of GPU and PC equipped with multi-core processor

	GPU	PC
プロセッサ	NVIDIA K20c (Kepler)	Core i7-3770 3.40GHz
コア数	2496	4(8)
メモリサイズ	5GB	16GB
OS	CentOS 6.5(ホスト)	CentOS 7
コンパイラ	CUDA 5.5	gcc 4.8.2

モデル 1 およびモデル 2 のデータで 3600 タイムステップを実行した結果を**図 8**, **図 9** に示す．

OpenMP プログラムの実行には，物理的には 4 コアのプロセッサで HyperThreading の適用により 8 スレッドまでスレッド数を増加させて実行した．2 章で述べたように，並列性を抽出するために冗長な演算を行っている．そのため，逐次プログラムの実行時間に比べ，1 スレッドの OpenMP プログラムの実行時間が長くなっている．

CUDA のプログラムは，3.2 節で述べたオリジナルのプログラムと，3.3 節で述べたストリームを利用してデータ転送とカーネル関数の実行のオーバラップを行ったプログラム (+stream)，ストリーム化に加えて 3.4 節で述べたループ交換と配列の次元の交換によるデータアクセスの coalescing を行ったプログラム (+coalescing) の 3 種類を実行した．

データサイズの大きいモデル 1 では，逐次プログラムの実行に比べ，GPU での実行により，5.79 倍に高速化されている．OpenMP8 スレッドでの実行に比べても，2.41 倍高速である．CUDA の実行では，ストリームの導入により 1.06 倍，ストリームと coalescing の最適化により 2.52 倍

に高速化されている．特に coalescing による効果が大きいことがわかる．

モデル 1 よりもデータサイズの小さいモデル 2 では，逐次プログラムの実行と OpenMP8 スレッドの実行に比べ，GPU での実行によりそれぞれ 4.18 倍と 1.57 倍に高速化されている．CUDA の実行では，ストリームの導入により 1.17 倍，ストリームと coalescing の最適化により 2.22 倍に高速化され，この場合にも coalescing による効果が大きいことがわかる．

モデル 1,2 ともに，総メッシュ数が GPU のコア数よりも大きいため，coalescing による効果はどちらのモデルでも大きくなっていると考えられる．

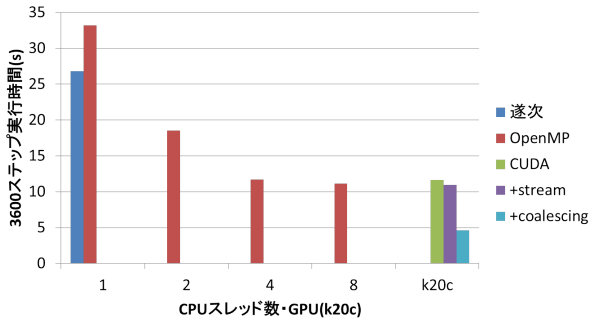

図 8 実行時間 (モデル 1)
Fig. 8 Execution Time for Model 1

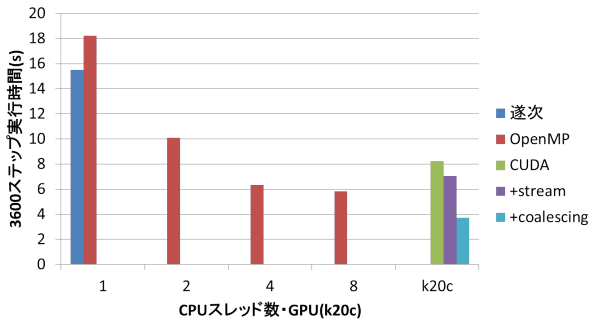

図 9 実行時間 (モデル 2)
Fig. 9 Execution Time for Model 2

CUDA での実行時間について，カーネル関数の実行時間と，それ以外の定数配列の転送などの時間の割合は**図 10**, **図 11** となる．カーネルの実行時間には，タイムステップ毎に変わる入出力データの転送時間も含まれる．

定数の転送時間は，モデル 1,2 によらず 1.35 秒でほぼ一定である．これは，転送時間の増加には，転送データサイズよりも CUDA の転送 API の起動回数による影響が大きいためと考えられる．

吸排気管の詳細な動作を解析するためにシミュレーションのタイムステップを増加させる場合には，定数転送の時間は変わらず，カーネル実行時間がタイムステップ数に比例して増加する．ゆえに，タイムステップ数を増加させるとカーネル実行時間が全体の実行時間の大部分を占めるようになる．このとき，ストリームや coalescing の導入の効果も，より大きくなることが期待できる．

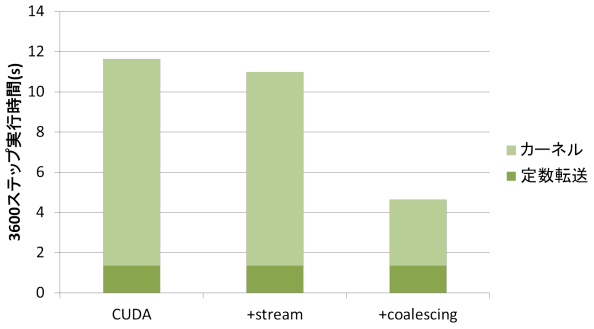

図 10 カーネル実行時間 (モデル 1)
Fig. 10 Execution Time of the Kernel Function for Model 1

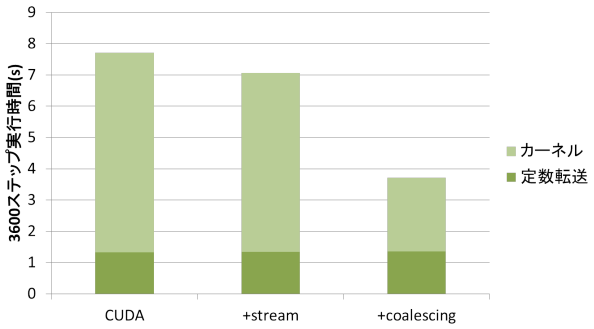

図 11 カーネル実行時間 (モデル 2)
Fig. 11 Execution Time of the Kernel Function for Model 2

5. おわりに

本発表では，自動車のモデルベース開発で用いられるエンジンの吸排気管のシミュレーションプログラムを取り上げ，その GPU への実装において適用したメモリ階層を考慮した性能チューニングについて報告した．性能チューニング技法としては，データ転送とカーネル関数のオーバーラップ実行，ループ交換と配列の次元の交換による coalescing の効果が大きかったことを示した．

今後は，XeonPhi 上への実装，汎用マルチコアプロセッサ上でのさらなる最適化などを検討したい．今回は GPU を主な対象として実装を進めたために，並列度を高めるようにシミュレーションプログラムのチューニングを進めたが，SSE や AVX などを活用する場合は，ループ分割を行い，コンパイラがこれらの SIMD 命令を適用できるように修正する必要があると思われる．これにより，本シミュレーションのプログラムの，各アーキテクチャに対して効果的な最適化手法を明らかにする．

謝辞 本研究を遂行するにあたり，広島市立大学卒業生

（現：株式会社ティラド）の岩本紳吾氏には、OpenMP による実装を中心に多大な貢献をいただいた．また，マツダ株式会社と広島市立大学の共同研究「FPGA/GPGPU/CPU を用いたモデル高速化技術の研究」メンバー諸氏には，活発な議論を通して多数の有益なご意見やコメントをいただいた．ここに感謝する．本研究は一部，JSPS 科研費新学術領域研究 26105013 の助成を受けた．

参考文献

[1] 藤川智士：マツダの目指すモデルベース開発，マツダ技報 31 (2013).
[2] 臼田浩平，小森　賢，三吉拓郎，寺岡陽一，本城　創，久禮晋一：SKYACTIV の MBD 検証環境について，マツダ技報 31 (2013).
[3] 横畑英明，佐藤圭峰，和田好隆，田所　正，小林謙太，植木義治：マツダ技報 31 (2013).
[4] Toro, E. F.: *Riemann Solvers and Numerical Methods for Fluid Dynamics*, Springer, 3 edition (2009).
[5] NVIDIA: CUDA, . https://developer.nvidia.com/cuda-zone.
[6] OpenMP Architecture Review Board: OpenMP Application Program Interface, . http://www.openmp.org.