卒業論文

題目

CUDAにおけるデータ配置変更による メモリアクセス効率化手法

指導教員

大野 和彦 講師

2014年

三重大学 工学部 情報工学科
コンピュータソフトウェア研究室

山中 準基 (410858)

内容梗概

近年，様々な分野において大規模計算の需要が高まっており，GPU に汎用的な計算を行わせる GPGPU への関心が高まっている．GPGPU 用の SDK である CUDA にはメモリ上のデータの配置によって複数のスレッドの処理を効率的に行うことができる機能がある．しかし，人間は CUDA プログラムを書くとき自分にとって書きやすいコードを書いていることが多く，速度面で最適であるとは限らない．そこで本研究では，構造体の配列を使用して書かれた CUDA プログラムをメモリアクセス効率化機能によって最適化されるように自動変換する手法を提案する．この手法を 2 種類のベンチマークプログラムに適用した結果，両方のプログラムで高速化することに成功した．現段階では実装していないが，本研究室で開発している MESI-CUDA に今後実装することが考えられる．

Abstract

In recent years, the demand of large-scale calculation is increasing in various fields, and concern is increasing in GPGPU which makes general purpose calculation perform to GPU. CUDA SDK for GPGPU has a facility which can process multiple threads effectively by placement of data on the memory. But, we write simple programs for me on many situations and they are not necessarily fast. Then, in this study, I propose the technique to optimize CUDA programs written by using Array of Structures automatically by the memory access optimization facility. I applied this technique to two benchmark programs and both programs ran faster. I has not implemented yet, but it is conceivable to implement this technique in MESI-CUDA which is developed in this laboratory in the future.

目 次

1 はじめに **1**
1.1 背景 1
1.2 研究目的 1
1.3 本文構成 1

2 背景 **2**
2.1 CUDA 2
2.2 MESI-CUDA 2
2.3 コアレッシングアクセス 3
2.4 構造体の配列と配列の構造体 4

3 提案手法 **6**
3.1 概要 6
3.2 最適化手法 6

4 評価 **10**

5 おわりに **13**

謝辞 **13**

参考文献 **13**

A プログラムリスト **17**
A.1 LavaMD 17
A.2 HeartWall 17

図 目 次

2.1 コアレッシングアクセスの例 4
2.2 AoS のメモリ上の配置 5
2.3 SoA のメモリ上の配置 5
3.4 AoS を SoA に変換する例 9
3.5 AoSoA を変換する例 9
3.6 AoS を SoA 用に拡張する例 9

表 目 次

4.1 LavaMD の実行時間 (秒) 12
4.2 HeartWall の実行時間 (秒) 12
4.3 LavaMD の入れ替えに掛かる時間 (秒) 12
4.4 HeartWall の入れ替えに掛かる時間 (秒) 12

1　はじめに

1.1　背景

近年，気象予測や分子動力学といった様々な分野において大規模計算の需要が高まっており，より高性能なコンピュータが求められている．そのため，CPU と比べ性能向上のめざましい GPU に汎用的な計算を行わせる GPGPU への関心が高まっている．GPGPU 用の SDK である CUDA[1][2][3] にはメモリ上のデータの配置によって複数のスレッドの処理を効率的に行うことができる機能がある．

1.2　研究目的

人間は CUDA プログラムを書くとき自分にとって書きやすいコードを書いていることが多く，速度面で最適であるとは限らない．そこで本研究では，メモリ上のデータの配置を自動最適化する手法を提案する．

1.3　本文構成

本文の構成は以下のようになっている．第 2 章で CUDA，メモリアクセス効率化機能の概要について述べ，第 3 章に提案手法の説明，第 4 章に性能の評価を行い，最後に第 5 章でまとめを行う．

2 背景

2.1 CUDA

CUDA は NVIDIA 社より提供されている GPGPU 用の SDK であり，ユーザーは C 言語を拡張した文法とライブラリ関数を用いて GPGPU プログラムを開発することができる．CUDA プログラムでは，まず CPU 側 (ホスト) で必要なデータを作成し，GPU 側 (デバイス) に転送する．次にホストが GPU に実行させる関数 (カーネル) を起動させ，デバイスがカーネルを実行し，処理を開始する．最後に結果をホストに転送し，デバイスから受け取ったデータをホストが処理することで終了する．また，CUDA にはビルトイン変数というものが用意されている．各スレッドは固有の ID を持ち，配列の添え字にその ID を表すビルトイン変数を利用することで個々の要素を並列に処理することができる．

2.2 MESI-CUDA

本研究室では CUDA プログラミングの負担を軽減させるフレームワーク MESI-CUDA[4][5][6][7][8] を開発している．MESI-CUDA はホスト・デバイス間のデータ転送コード，メモリ確保・解放コード，ストリーム生成・破棄のコードを自動的に生成することでユーザーの CUDA プロ

グラミング開発の負担を軽減させる．ホストとデバイスへの処理の振り分けや GPU 上で実行するカーネルの記述はユーザ自身が従来の CUDA に準じる形でコーディングを行う．現状の MESI-CUDA はメモリ上のデータの配置が最適化された CUDA プログラムを生成できないという問題がある．

2.3 コアレッシングアクセス

GPU には性能の違う複数のメモリが階層的に存在しており，本研究ではグローバルメモリを対象とする．グローバルメモリはレイテンシが大きいため，このメモリへの転送の回数が多いほど処理速度は大きく低下する．さらにアドレスを 128byte 単位のブロックで転送するという特徴を持つ．従って効率的なメモリアクセスを行うには，ブロック単位にアクセスするアドレスを収めて転送処理の回数を少なくすることが必要である．このメモリアクセスをコアレッシングアクセスと呼び，CUDA の性能最適化において非常に重要である．fig. 2.1 にコアレッシングアクセスの例を示す．斜線部が各スレッドがアクセスするアドレスである．このアドレスが 128byte 以内の範囲に収まっていると転送するブロックは少なくて済むが，収まっていないと転送するブロックが増え，転送処理

の回数が増加する．

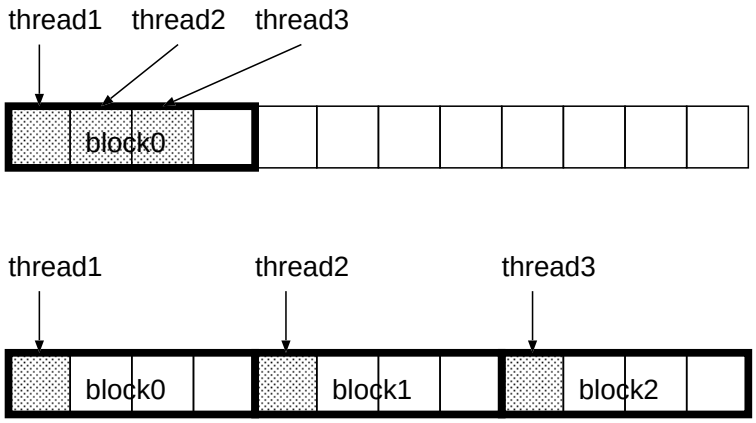

fig. 2.1: コアレッシングアクセスの例

2.4 構造体の配列と配列の構造体

構造体の配列 (Array of Structures:AoS) とは，異なる型の変数を組み合わせたデータ構造である構造体を配列にしたものである．ある一つの物事の情報を一つの構造体にまとめることができるため人間にとって理解しやすく自然な表現である．しかし，例えば x,y,z メンバを持つ構造体の配列はメモリ上で fig. 2.2 のように配置され，各スレッドが個々の x メンバにアクセスする場合，それらの領域は不連続である．従ってコアレッシングの効果が薄いためアクセス性能が下がってしまう．配列の構造体 (Struct of Arrays:SoA) とは，複数の配列を構造体にしたものであり，メモリ上の配置は異なるが AoS と同じ情報を格納できる．この型は上記

の例で fig. 2.3 のように配置され，個々の x メンバに当たる部分がメモリ上で連続しておりコアレッシングの効果により AoS で書かれたコードよりアクセス性能が良い．しかし，SoA は AoS と比べて不自然な構造であり，プログラマにとって理解しにくい．そのため，多体問題などの一つの粒子に多くの情報を持たせる構造体を使う CUDA プログラムでは一般的に AoS が良く使用されている．

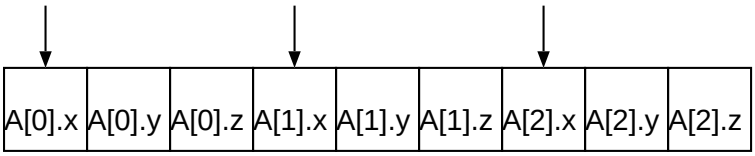

fig. 2.2: AoS のメモリ上の配置

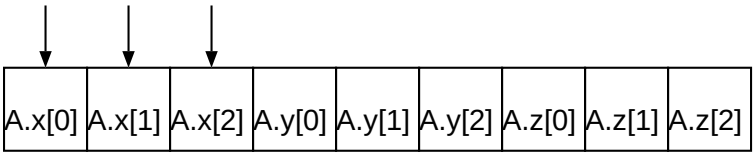

fig. 2.3: SoA のメモリ上の配置

3　提案手法

3.1　概要

2.2節の問題に対処するために，AoSを使用して書かれたCUDAプログラムを最適化する必要がある．コアレッシングアクセスはGPU側で行われる処理なので，カーネル内のAoSをSoAに変換する．しかし，CPU側ではGPUにおけるこの効率的なアクセス方法による影響が無いためAoSとSoAのどちらが効率的であるかは場合による．そのため本手法ではカーネル部分とカーネルに関わる部分だけを変換する．処理の流れはホスト側でAoSをSoAに変換し，このSoAをカーネルに転送する．カーネル実行し，計算が終わったSoAをホストに返す．最後にSoAをAoSに再変換する．

3.2　最適化手法

最初にカーネル内のAoSの式を検出する．どのような形がAoSであるか判定するのかは構造体宣言から構造体の名前を記憶し，カーネル内で構造体 [添え字]. 変数となるような式を検出する．次にどのAoSをSoAに変換するか判定する．これは並列実行するAoSだけを変換させるためである．配列の添え字に一個でもビルトイン変数が使用されていれば変

換の対象とする．この変数が使用されていない配列は並列実行しないため，本手法による最適化の効果が無いので対象から除外する．また，配列の添え字が定数倍であったりランダムになっているとコアレッシングの効果が薄い可能性がある．しかし SoA に変換することで同時実行するアドレスの間隔が狭まることはあっても離れることはないのでアクセス性能が下がることは無いと考えられる．従って，カーネル内の並列実行する AoS を全て SoA に変換する．

AoS の式を変換するために当然角括弧の中身も移動する必要があるので，構造体名直後の添え字を構造体メンバ名の直後に移動する．例えば fig. 3.4 では，構造体 [添え字]. 変数となっているのでこれを構造体. 変数 [添え字] と変更する．また，元の構造体中にまた配列が含まれていることがある．これを配列構造体配列 (AoSoA) と呼ぶ．この場合は変更後の SoA の配列を二次元構造にすることで変換が実現できる．例えば fig. 3.5 では，構造体 [添え字]. 変数 [添え字] となっているので，これを構造体. 変数 [添え字][添え字] と変更する．

ここで，AoS は構造体を配列と同じようにみなすことができるが，SoA はいくつかの配列をまとめただけなため，変換しないとエラーが出る式がある．従って，構造体の状態で計算している式を配列で計算するように拡

張する必要がある．この例を fig. 3.6 に示す．次にグローバル変数として SoA 用の構造体を宣言する．この構造体の要素は変換する AoS の要素を配列としたものである．カーネルを起動する前に SoA 構造体のメモリ領域を確保し，カーネルに引数として渡す AoS の値をコピーする．カーネル終了後 SoA 構造体の値を元の AoS にコピーし，メモリ領域を解放する．

現状，AoS と AoSoA で構成された構造体しか想定していないが，構造体中に別の構造体が含まれている場合もある．今回対応していないが，構造体中の構造体を個々の配列に変換することが考えられる．

```
A[i].x; → A.x[i];
```

fig. 3.4: AoS を SoA に変換する例

```
A[i].x[j]; → A.x[j][i];
```

fig. 3.5: AoSoA を変換する例

```
  B = &A[i];
      ↓
B.v = &A.v[i];
B.x = &A.x[i];
B.y = &A.y[i];
B.z = &A.z[i];
```

fig. 3.6: AoS を SoA 用に拡張する例

4　評価

本手法を用いた最適化の有無による CUDA プログラムの実行時間の比較を行った．評価は二つの計算機で行った．評価環境は Core i7-3820 3.60GHz，メモリ 16GB，GeForce GTX 680 を搭載した計算機と Core i7-4820K 3.70GHz，メモリ 16GB，Tesla K20 を搭載した計算機である．評価に用いたプログラムは Rodinia[9] ベンチマークである LavaMD と HeartWall である．LavaMD は三次元空間内の粒子間の相互作用を計算し，各々の粒子の新しい座標と力を求める．各々の粒子の座標と力は一つの構造体に含まれており，AoS で書かれている．この三次元空間は並列処理するためにいくつかの空間に分割される．一辺の分割する数が 10, 20, … 70，つまり空間を 10^3，20^3，… 70^3 に分割した場合の実行時間を計測した．この結果を Table 4.1 に示す．HeartWall は超音波映像に対し，一つのフレームに並列処理を行い，フレーム毎の情報を AoS に格納する．このプログラムはフレーム毎にカーネルを呼び出す．104 フレームの映像に対し，20，40，… 100 フレーム回数分，カーネルを実行した場合の実行時間を計測した．この結果を表 4.2 に示す．

また，本最適化手法によって実行時間が短縮しても SoA 構造体のメモリ領域確保と値コピーの処理時間が多大にかかる場合，有用であるとは

いえない．このため，この処理の実行時間を計測した．これらのプログラムでの入れ替えは一回のみである．これらの結果を表 4.3，表 4.4 に示す．

表 4.3，表 4.4 から処理時間はほとんどかからないことが分かる．また表 4.1 から，表 4.3 の実行時間が増加していても非最適化に比べ最適化されたプログラムの実行時間が短縮されていることが分かる．これはカーネル内の AoS を SoA に変換したことでコアレッシングの効果により，処理速度が上昇したためである．また分割数が少ないと実行時間の差があまり無いのは，最適化したカーネル内の処理時間に比べ，ホストでの処理時間が大きいためである．

表 4.1: LavaMD の実行時間 (秒)

		10	20	30	40	50	60	70
GTX680	非最適化	1.073	2.176	5.311	11.540	22.044	38.079	60.710
	最適化	1.038	2.064	4.956	10.542	19.965	34.029	53.776
K20	非最適化	1.707	3.281	7.761	16.599	31.318	53.309	84.203
	最適化	1.577	2.313	4.327	8.349	15.037	25.077	39.057

表 4.2: HeartWall の実行時間 (秒)

		20	40	60	80	100
GTX680	非最適化	1.402	1.970	2.474	2.982	3.514
	最適化	1.000	1.091	1.117	1.205	1.281
K20	非最適化	2.158	2.855	3.467	4.134	4.979
	最適化	1.569	1.605	1.649	1.688	1.718

表 4.3: LavaMD の入れ替えに掛かる時間 (秒)

	10	20	30	40	50	60	70
GTX680	0.005	0.028	0.085	0.200	0.384	0.655	1.044
K20	0.006	0.027	0.083	0.192	0.374	0.640	1.020

表 4.4: HeartWall の入れ替えに掛かる時間 (秒)

GTX680	K20
0.00727	0.00756

5　おわりに

本研究では，CUDA プログラムにおけるメモリ上のデータの配置を自動最適化する手法を提案し，評価を行った．その結果，本手法を用いることでメモリアクセスを効率的に行うことができ，実行時間の高速化を行うことができた．

今後の課題として，実際に MESI-CUDA により生成されるプログラムを解析し，この自動最適化を実装することが必要である．

謝辞

本研究を行うにあたり，ご指導，ご助言いただきました下さいました大野和彦講師に深く感謝いたします．また，様々な局面にてお世話になりましたコンピュータソフトウェア研究室の皆様にも心より感謝いたします．

参考文献

[1] NVIDIA Developer Zone，https://developer.nvidia.com/category/zone/cuda-zone. NVIDIA Developer Zone.

[2] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, April 2012.

[3] NVIDIA Corporation. CUDA C Best Practices Guide, January 2012.

[4] 道浦悌, 大野和彦, 松本真樹, 佐々木敬泰, 近藤利夫. GPGPU におけるデータ転送を自動化する MESI-CUDA の提案. In 先進的計算基盤システムシンポジウム SACSIS2012, pp. 201–209, 2012.

[5] K. Ohno, M. Matsumoto, T. Kamiya, and T. Maruyama. Supporting dynamic data structures in a shared-memory based GPGPU programming framework. In Proc. 24th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems, pages 122—131, 2012.

[6] K. Ohno, D. Michiura, M. Matsumoto, T. Sasaki, and T. Kondo. A GPGPU programming framework based on a shared-memory model. Parallel and Distributed Computing and Networks, 3:1—14, 2013.

[7] Tomoharu Kamiya, Takanori Maruyama, Kazuhiko Ohno, and Masaki Matsumoto. Compiler-level explicit cache for a GPGPU programming framework. In Proc. The 2014 Intl. Conf. on Parallel and

Distributed Processing Techniques and Applications, pp. 632-638, July, 2014.

[8] Kazuhiko Ohno, Tomoharu Kamiya, Takanori Maruyama, Masaki Matsumoto, Automatic Optimization of Thread Mapping for a GPGPU Programming Framework, 2014 Second International Symposium on Computing and Networking (CANDAR 2014), pp. 198-204, December, 2014.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, Sang-Ha Lee and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. IEEE International Symposium on Workload Characterization, Oct 2009.

[10] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In CCGRID, pages 136—143. IEEE Computer Society, 2013.

[11] J. A. Stratton, N. Anssari, C. Rodrigues, I. Sung, N. Obeid, L. Chang, G. D. Liu, and W. Hwu. Optimization and architecture ef-

fects on GPU computing workload performance. In Proc. Innovative Parallel Computing 2012, InPar 2012, pages 1—10, 2012.

A　プログラムリスト

評価に用いたベンチマークプログラムの実行方法を列記する.

A.1　LavaMD

- 改良前: lavamd_cuda_code_original/a.out
- 改良後: lavamd_cuda_code/a.out

実行方法

- a.out -boxes1d 10
- 数字を 1〜76 に変更できる.

GPU 環境によって makefile 内の CUDA_FLAG = -arch sm_35 を -arch sm_13 や -arch sm_30 に変更しなければならない.

A.2　HeartWall

- 改良前: hw_tracking_cuda_code_original/a.out
- 改良後: hw_tracking_cuda_code/a.out

実行方法

- a.out 10

- 数字を 1〜104 に変更できる.

input フォルダ内に Rodinia ホームページ,

http://www.cs.virginia.edu/škadron/wiki/rodinia/index.php/Main_Page

からダウンロードできる input.avi と input.txt が必要である.