巡回セールスマン問題を題材としたOpenCLによる並列化

村上 隆俊[1]　林田 隆則[1]　佐藤 寿倫[1]

概要：今日, 処理性能を改善するためには並列処理が必須である. そのためには並列プログラムを実装する必要があるが, プラットフォーム毎に異る実装が必要な現状では, その敷居は高い.OpenCL はこの課題を解決するための一つの試みである. そこで今回, 巡回セールスマン問題を題材にして,OpenCL による並列化の効果を評価した. コード変更の容易な SIMD 演算によるベクトル化と局所探索アルゴリズムのデータ並列化では十分な性能改善は望めない. 並列化による効果を得るためには依然としてプログラマの負担が大きいことが確認された.

OpenCL Programming for Parallelizing Traveling Salesman Problem Solver

Abstract: In order to improve performance, parallel processing is an inevitable technique. Currently, dedicated implementation is required for different parallel platforms. It is a very tedious task for programmers. OpenCL is a possible solution to the problem. In this study, we implement a parallel program that solves travelling salesman problem using OpenCL environment. We exploited SIMD parallelism to improve performance and modified local search algorism to data parallelism. Unfortunately, performance is still poor after the parallelization.

1. はじめに

今日では演算処理速度を向上させるための手法として処理の並列化が主流になっている. 例えば,CPU のマルチコア化に伴ってプログラムの並列化が重要になっている. また, 演算に特化したコアを多数持つ GPU の採用が盛んになっている.

GPU を用いた開発には,NVIDIA 社が提供している CUDA[1] や AMD 社が提供している APP[2] など, SDK を用いてそれぞれの GPU の性能を生かしたプログラムを作成することが一般的である. しかしベンダー毎に異なる開発言語を用いなければならず, その開発環境を学習しなければならなかったり, それぞれの環境に合わせた開発を行わなければならなかったりなど, 開発コストが高く平均的なプログラマが利用するには敷居が高い.

OpenCL[3] はこの問題の一つの解決策となるオープンプラットフォームで, 共通の環境で開発されたプログラムが異なるベンダーのデバイス上で動作可能になる. OpenCL によってプログラムの可搬性を確保できる. 本論文では,OpenCL を用いて巡回セールスマン問題の最適解を求めるプログラムの並列化を行い, それらが実行時間の短縮に貢献するかを評価する.

本論文は以下の構成となる.2 章で GPU コンピューティングと OpenCL について説明し、3 章で巡回セールスマン問題について説明する. 4 章で OpenCL による巡回セールスマン問題の最適解プログラムの実装について説明する. 5 章でプログラムの評価と結果を報告し,6 章で本論文をまとめる.

2. GPU コンピューティング

2.1 GPU コンピューティングとは

元々 GPU(Graphics Processing Unit) は画面表示を高速に行うためのユニットで, 描画のための単純な機能だけを搭載していた. しかし現在では,3D グラフィックスや動画再生などの処理を行うために, 汎用の演算器を持っているものが多い. GPU は大量データの並列処理に適しており, 汎用的な計算が出来るようになったことと GPU の性能が向上したことに伴って, グラフィックス以外の用途での活用が注目されるようになった. これを GPGPU(General-

[1] 福岡大学工学部電子情報工学科
Department of Electronics Engineering and Computer Science, Faculty of Engineering, Fukuoka Uniersity

purpose computing on GPU) と呼ぶ. 今日の HPC(High Performance Computing) 分野においては, 物理シミュレーションや金融予測, 解析などの科学技術計算 [4] に GPU が利用されている.

2.1.1 Fermi アーキテクチャ

本研究で使用する Fermi アーキテクチャを, 汎用計算向けの拡張が施されている GPU の代表として説明する. Fermi アーキテクチャの特徴として以下があげられる [5].

CUDA コア

Fermi アーキテクチャでは最大 512 個の CUDA コアを有しており,32 個の CUDA コアで 1 つの SM(Streaming Multiprocessor) を構成している. この SM には 64KB の L1 キャッシュと, チップ上に全 SM で共有される 768KB の L2 キャッシュが搭載されており, データへのアクセスレイテンシの削減とパフォーマンス向上に寄与している.

演算性能の強化

より高速に汎用計算ができるように, Fermi アーキテクチャでは倍精度浮動小数点演算性能が強化されている. 前世代のアーキテクチャと比較して 4 倍高速化している. さらに SFU(Special Function Unit) と呼ばれる, 三角関数や指数関数, 対数関数などの超越関数を処理するためのユニットが倍増され, 汎用計算に必要な機能が強化されている.

スケジューラの強化

GPU コアへのタスクスケジューラが強化され, プログラムの切り替えが前世代アーキテクチャより 10 倍高速化されている. さらにこれまでの GPU では一つのカーネル (プログラム) しか実行できなかったが, Fermi アーキテクチャでは複数の GPU プログラムを同時に実行できるようになり, GPU コアを無駄なく利用できるようになっている.

2.2 OpenCL とは

GPU を汎用計算で利用するためには,NVIDIA 社の CUDA[1] や AMD 社の APP SDK[2] などの開発環境を用いて並列処理プログラムを開発することが必要になる. 例えば CUDA では,CPU(ホスト) が並列処理を行う GPU(デバイス) を制御して利用する. ホストはデバイスの制御や命令コード生成, 処理対象データの転送などの役割を受け持つ. デバイスは CUDA 拡張言語で開発された並列プログラムを実行する.

しかし,CUDA で開発されたプログラムは NVIDIA 社の GPU 上でしか利用できず, 他社の GPU で実行することができない. このような特定のベンダーに依存した開発環境では, 各開発環境をそれぞれ習得しプログラムを作成しなれけばならない. このことは開発効率を低下させ, 作成されたプログラムの可搬性も低下させる.

この問題を解決するために, どのベンダーの GPU でも共通の手続きで扱うことをめざし OpenCL(Open Computing Language) と呼ばれる規格が策定された. OpenCL とは特定のベンダーやプロセッサに依存しない統一的な並列プログラミングフレームワークであり, OpenCL をサポートしていればどのようなプロセッサでも同一のコードを実行できる. これらには,NVIDIA 社や AMD 社,Intel 社のマルチコア CPU と GPU, IBM/東芝/ソニーの Cell/B.E. そして様々な DSP などがある. これらを OpenCL デバイスと呼ぶ.

2.2.1 OpenCL の構成

OpenCL は以下の 2 つで構成されている [3][6].

OpenCL ランタイム API 仕様

OpenCL ランタイム API は, OpenCL デバイスを制御するための API, コマンドキューと呼ばれる演算用プロセッサで実行するタスクをスケジューリングするための API, デバイスへのメモリ制御のための API, そして OpenCL デバイスで実行するためのカーネルを生成するための OpenCL コンパイラなどで構成されている.

OpenCL C 言語仕様

OpenCL C 言語は C 言語 (C99) をベースに並列処理に必要な機能を拡張した言語であり, 並列処理用の関数や算術関数ライブラリ, OpenCL のメモリモデルに対応するためのアドレス空間修飾子, そして OpenCL デバイスでの SIMD 演算を容易にするためのベクタ型の追加などの拡張がされている. この言語で開発されたプログラムを OpenCL ランタイム API でビルドし, 生成されたカーネルと呼ばれるバイナリを演算用プロセッサで実行する.

2.2.2 OpenCL の利点

OpenCL のメリットは以下の 3 点である [3]. 1 つ目は, 全く異なるデバイスを同じ手順で利用できることである. デバイス毎に異なる API を習得しなくても良いため, 開発コストを低減しコードの可搬性を高めることができる.

2 つ目は, 利用するデバイスの特徴に合わせた最適化ができることである. OpenCL はハードウェアに近いレベルの API であるため, それぞれのプロセッサのメモリ構成や特性に合わせて最適化を行うことができる.

3 つ目は,C 言語に似た文法を採用しているために OpenCL の学習コストを低く抑えられることである. OpenCL C 言語の記述方法は C 言語と大きく変わらないため, 容易に開発を行うことができる.

2.2.3 OpenCL のメモリモデル

OpenCL のメモリモデルでは,OpenCL デバイスのメモリ空間をグローバルメモリ, コンスタントメモリ, ローカルメモリ, プライベートメモリの 4 つに分類している. それぞれ, デバイスのメインメモリ, 演算ユニットのキャッシュメ

モリ, 演算ユニット上の共有メモリ, 演算ユニットのレジスタを想定している. どの空間をどのメモリに割り当てるかははデバイス毎に異なる.

3. 巡回セールスマン問題

3.1 巡回セールスマン問題とは

巡回セールスマン問題 [7][8] とは, 地点の集合とその地点間の距離が与えられたときに, 各地点を一度ずつ通って最初の地点に戻ってくる巡回路のなかで, 最も短い距離で移動できるものを求める組み合わせ最適化問題である. この問題は, 運搬経路計画や基盤配線,VLSI の設計などに応用されている. 巡回セールスマン問題を効率的に解くことで, これらの応用において時間短縮や設計の効率化などが可能になる.

巡回セールスマン問題は NP 困難な問題のひとつとされている. 全ての巡回路を列挙して最適解を求めることを考える. 地点の数が n で任意の 2 点間の距離が対称の場合には, 巡回路の総数は $\frac{(n-1)!}{2}$ となり, 都市数が増えるにしたがって組み合わせが指数関数的に増える. 例えば n=50 で $4.66631077 \times 10^{155}$ 通りという膨大な組み合わせになる. つまり巡回セールスマン問題を列挙法で解くことは不可能である.

3.2 近似解法

総当りにより巡回セールスマン問題の厳密解を求めることは不可能であるため, 近似解を求めることになる. 様々な近似解を求める方法が考案されている. ここでは「局所探索法」について述べる.

局所探索法は, 巡回路の局所的な部分に変形を加えることで解を改善する方法である. 代表的な方法は k-opt 法 ($k > 2$) である. k は選択する枝の数であり, 多くの場合 2-opt 法あるいは 3-opt 法が利用される. また k を可変にするように拡張された Lin-Kernighan 法がある. 2-opt 法の手順は以下のとおりである.

2-opt 法

(1) 適当な巡回路を生成する.
(2) 生成した巡回路から都市間を繋ぐ 2 本の枝 (a, b) と (c, d) を選択して,
$d(a,c)+d(b,d) < d(a,b)+d(c,d)$ となる枝を探す.
(3) そのような枝があれば, 枝 (a, b) と (c, d) を枝 (a, c) と (b, d) に繋ぎ変えて 2. に戻る.
無ければ終了する.

2-opt 法では調べる組合せは $\frac{(n-1)\times n}{2}$ になるため, 列挙法と比べて計算時間を大幅に短縮できる.

4. OpenCL による TSP 最適解プログラムの並列化

今回, 局所探索法の 2opt 法を実装した. まず逐次プログラムを実装し, アルゴリズム実装の正しさを検証した. 続いて OpenCL により並列プログラムとして書き直した. 開発言語には C++を用いた. 続いて OpenCL による実装で行った並列化手法を説明する.

4.1 ベクトル化

都市間距離の計算において並列性があることに着目した. ここではベクタ型を活用し,SIMD 型演算として並列化を行った. 図 1 と 2 に並列化前と後のコードの例を示す. 並列化前に float 型として表されている都市の座標値 x と y を float2 型にすることでベクトル化し, 都市間距離の計算を高速化する. float2 型はベクタ型のひとつで,2 つの float 型を 1 つの型として同時に扱うことできる. ベクタ型では 1 命令で複数の値に対して同時に演算操作を行える. また,SIMD 型演算は 1 命令で複数のデータを処理することが出来る. ベクタ型を活用することで,OpenCL が SIMD 型演算を活用するプログラムを生成する.

```
1 float Distance::euclidean(Node const &a, Node const
      &b)
2 {
3         float xd = a.x − b.x;
4         float yd = a.y − b.y;
5         return round(sqrt(xd ∗ xd + yd ∗ yd));
6 }
```

図 1 都市間距離計算関数 (並列化前)

```
1 float distSum(const float2 a, const float2 b)
2 {
3         float2 d = a − b;
4         return round(sqrt(d.x ∗ d.x + d.y ∗ d.y));
5 }
```

図 2 都市間距離計算関数 (並列化後)

4.2 データ並列化

局所探索法においてデータ並列化ができる部分を抽出し, プログラムの処理方法を変更して並列処理化を行った. 次に局所探索では図 3 に示すように都市間を繋ぐ枝を 2 重の for ループで都市を繋ぐ枝を選択し, 入れ替えた場合に距離が短くなるか否かの評価を行なっている. 都市の入れ替えを行わなければこのループにはデータ依存が無いため, これを並列化することにした. **図 4** に示すように, 評価すべき枝の組合せは右下の部分となる縦方向がインデックス i

```
1  int st = 0;
2  RESTART:
3  for (int i = st; i < st + toursize; i++) {
4          for (int j = i + 2; j < i + toursize − 1; j++) {
5                  // 距離短縮の評価
6
7                  if 短い場合 () {
8                          // 入れ替え処理
9
10                         // 探索開始点を変更
11                         st = (i + 1) % toursize;
12                         goto RESTART;
13                 }
14         }
15 }
```

図 3　局所探索における枝の選択

で, 縦方向がインデックス j を示している．選択される枝をある程度の大きさのブロックごとに分割する．図では 20 個ずつに分割して色分けして示している．このブロックに含まれる枝の評価をデータ並列で行う．評価後にブロックの中から距離を短く出来るものがあれば, 枝の入れ替えを行う. 無ければ次のブロックを同じように評価する．全てのブロックで距離を短くできる枝が無くなれば, 探索を終了する．

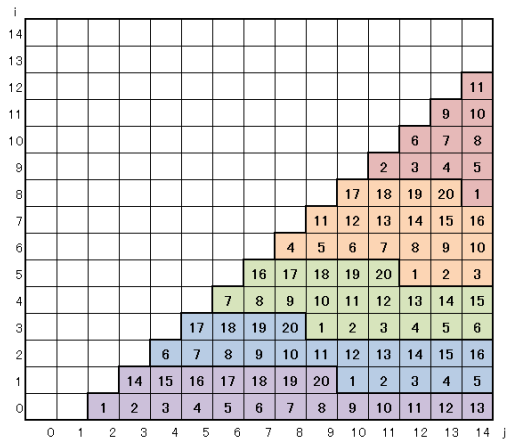


図 4　枝の選択

5.　評価

本節で,4.1 節でのベクタ型を用いたプログラムと, 4.2 節のデータ並列化を加えたプログラムの並列化を評価する．

5.1　評価方法

評価するためのベンチマークには巡回セールスマン問題を集めた tsplib[9] を利用した．このベンチマークでは 2 点間距離の計算方法と都市の座標データが与えられる．

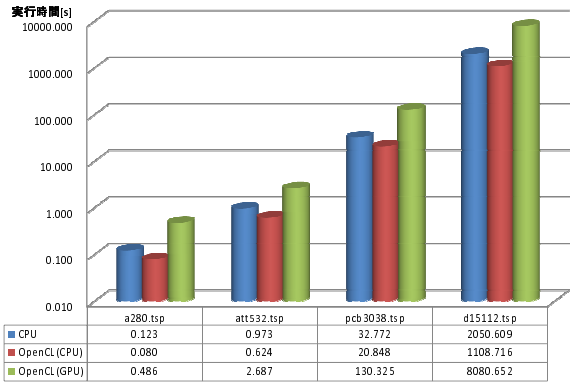
今回は a280,att532,pcb3038,d15112 の 4 つの問題を選択して使用した．それぞれの都市数は 280,532,3038,15112 である．前節で紹介した逐次プログラムと OpenCL で実装した並列プログラムの実行時間を比較した．逐次版は CPU 上で実行した．OpenCL 版は, 異なるデバイスで特徴に違いが出るか否かを確認するために,CPU 上と GPU 上とで実行した．実行時間は巡回路の局所探索の開始から最適解の導出までの時間とし, 初期化に要する時間を含めない. 但し,CPU と GPU との間の通信時間は含まれる．それぞれ 6 回測定し, 最大値と最小値を除いた 4 回の測定結果を用いて平均値を求めた．評価に使用した環境を**表 1** に示す．

表 1　評価環境

	逐次実行	OpenCL(CPU)	OpenCL(GPU)
CPU	Intel Xeon W3580 3.2GHz		
GPU	N/A		NVIDIA Tesla C2075
OS	Ubuntu 12.04 64bit		

5.2　結果と考察

ベクタ化の評価結果を**図 5** に示す．グラフは問題ごとに左から CPU 上で逐次プログラムを実行したとき, 並列プログラムを CPU 上で実行したとき, 並列プログラムを GPU 上で実行した時の実行時間である．対数グラフであることに注意されたい．表には局所探索にかかった時間 (秒) を示す．

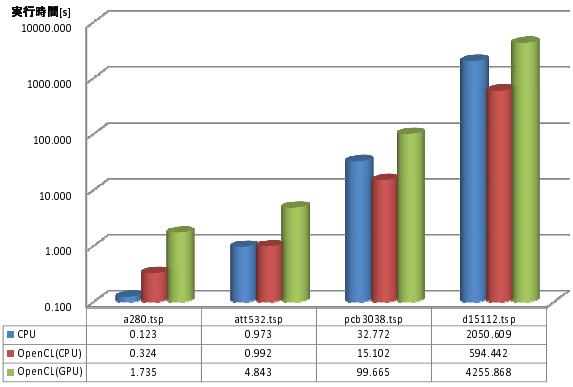


	a280.tsp	att532.tsp	pcb3038.tsp	d15112.tsp
CPU	0.123	0.973	32.772	2050.609
OpenCL(CPU)	0.080	0.624	20.848	1108.716
OpenCL(GPU)	0.486	2.687	130.325	8080.652

図 5　ベクトル化の効果

CPU 上で並列プログラムと逐次プログラムを実行した場合を比較すると, 並列化による効果が得られていることが判る．a280,att532,pcb3038 では, どれも逐次プログラムより 35%程度高速化している．d15112 では逐次プログラムより 40%高速化している．これはベクタ型を使用したことで,CPU の SIMD 命令を使うようになったためと考えられる．GPU 上で並列プログラムを実行すると,CPU 上で逐次プログラムを実行するよりも総じて実行時間が大きくなる結果となった．逐次プログラムをほぼそのまま OpenCL に移植したため, GPU の得意とするデータ並列性を利用出来ておらず,GPU の性能を生かしきれなかったからである．

ベクタ化とデータ並列化の両方を施した場合の結果を



	a280.tsp	att532.tsp	pcb3038.tsp	d15112.tsp
CPU	0.123	0.973	32.772	2050.609
OpenCL(CPU)	0.324	0.992	15.102	594.442
OpenCL(GPU)	1.735	4.843	99.665	4255.868

図 6 データ並列化の効果

図 6 に示す. 図の見方は図 5 と同じである.

ベクタ化に加えてデータ並列化を施すと, 都市数が増えるにつれて, CPU 上での並列プログラムの実行時間が逐次プログラムより-163%,-2%,54%,71%となる. a280 と att532 では実行時間が長くなる一方, pcb3038 と d15112 では実行時間が短くなっており, 都市数が増えてくると並列化の効果が出ていることが確認できる. GPU 上で並列プログラムを実行したとき, CPU 上での実行時と同様に pcb3038,d15112 において実行時間が短縮が確認できた. しかし依然, 逐次プログラムより遅い結果となっている. データ並列化によって,OpenCL デバイスとホスト CPU の間で局所探索の制御を行う通信が増えたことによって, 都市が少ない場合にはデータ並列化の効果が打ち消されてしまったと考えられる.

6. まとめ

今回, 巡回セールスマン問題の 2opt 法による近似解法を OpenCL を用いて並列化し, その効果を評価した. SIMD 演算によるベクトル化と局所探索のデータ並列化によって CPU では効果を得ることが出来た. しかし,GPU では期待される並列化の効果を得ることは出来なかった. データ並列化においては GPU は多くコアを持つにも関わらず, 逐次プログラムより遅くなる結果となった.

OpenCL によるプログラム作成において,OpenCL デバイスとホスト CPU とのデータの通信処理や OpenCL デバイスで実行するプログラムのキューイングが必要なる. さらにプログラムの並列化において, アルゴリズムへの理解した上で並列化箇所の検討と実装, デバッグといった作業が必要になるため, 依然プログラマへの負担が大きい. しかし, 同一の並列プログラムで CPU と GPU で動作させることが出来るため, 可搬性においてはプラットフォームごとのフレームワークよりはプログラマの負担は軽減される.

今後の課題として, 並列プログラムのプロファイリングによって GPU で遅くなる原因を特定してプログラムの改良を行う, 今回用いたデータ並列化手法とは異なる並列化手法を検討する, 局所探索法以外に並列化に適した近似解法が無いかを検討するなどが考えられる.

謝辞

本研究の一部は，科研費挑戦的萌芽研究（課題番号：23650026）によるものである.

参考文献

[1] NVIDIA Corp.: CUDA, http://www.nvidia.com/object/cuda_home_new.html (accessed 2012-12-2).

[2] AMD Inc.: Accelerated Parallel Processing (APP) SDK, http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/ (accessed 2012-12-2).

[3] 土山了士, 中村孝史, 飯塚拓郎, 浅原明広, 三木聡：OpenCL 入門，インプレスジャパン (2012).

[4] GPU コンピューティング研究会：GPU を利用した研究事例・成果, http://gpu-computing. gsic.titech.ac.jp/node/52 (参照 2012-12-2).

[5] NVIDIA Corp.: NVIDIA's Next Generation CUDA Compute Architecture: Fermi, Whitepaper (2009).

[6] Khronos Group: The OpenCL Specification Version:1.2 Document Revision:15, Technical report (2012).

[7] 山本芳嗣，久保幹雄：巡回セールスマン問題への招待，朝倉書店 (1997).

[8] B. コルテ，J. フィーゲン：組合せ最適化 (第 2 版)，シュプリンガー・ジャパン (2009).

[9] G.Reinelt: TSPLIB, http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/ (accessed 2012-10-9).