技術・研究報告

再帰関数を含むアプリケーションに対する OpenCLのハイブリッド並列

若谷彰良, 中谷秋栄

甲南大学 知能情報学部
神戸市東灘区岡本８－９－１, 658-8501

(受理日 2023 年 5 月 9 日)

概 要

近年, GPU をアクセラータをして活用されることが一般化し, ノートパソコンにも GPU が搭載されており, 単体のプロセッサにも CPU と GPU が統合されていることが多くなっている. よって, GPU の性能を活かした高速計算が可能となっており, CPU と GPU の両方を用いた高性能計算が容易に実現できる. しかし, 一つのアプリケーションを両方のハードウェアを利用して実行する場合, 計算のタスクをどのように負荷分散すればいいかは必ずしも予想が容易ではない. 本研究では, 再帰関数を含むアプリケーションの例として巡回セールスマン問題を対象とし, マイクロタスクの実行による CPU と GPU の最適な負荷分散を事前に決定し, 性能を最適化する手法の提案とその評価を行なう.

キーワード: 高速計算, 負荷分散, チューニング, 並列化

1 はじめに

コンピュータの性能は CPU に比例することが多いが，CPU 単体による性能向上の限界は近づきつつある．そこで昨今の AI アプリケーションでは CPU に加え，アクセラレータとして GPU を用いられることがあるが，このシステムは異なるプロセッサの特性を活かした処理を必要とする [1]．本研究では GPU を有効活用するための一つの手法として, CPU と GPU を同時に動作させるハイブリッドな並列処理方法を提案している．そこで，本研究は巡回セールスマン問題を対象とし，OpenCL 環境において CPU 内蔵 CPU における CPU 及び GPU によるハイブリッド並列の最適な負荷分散を求め，性能向上の評価を行う．また，全ての経路を探索して最適な負荷分散を求めるのは冗長で実用性に乏しいことから，その負荷分散を実行前に推定する自動チューニングプログラムを実装し，その性能評価を行う．

2 対象問題

2.1 巡回セールスマン問題

巡回セールスマン問題（Traveling Salesman Problem）は「セールスマンがいくつかの都市を 1 度ずつすべて訪問して出発点に戻ってくるときに，移動距離が最小になる経路」を求める問題である [3]. 総当り法による単純な解法では都市数の階乗に比例した計算時間がかかるが, 計算量を抑えて, 実用的に解く手法は様々なものが知られている [4]. また, 巡回セールスマン問題の総当り法は, 再帰関数を含む整数アプリケーションの一例としても有名な問題でもある. 本研究では再帰関数を含む大規模計算を対象とし，ハイブリッド並列実装の評価を行う．また，今回はハイブリッド並列の実行速度と CPU 単独，GPU 単独の実行速度を比較した際の性能差を評価することが目的なので，再帰関数を用いた総当たり法を用いる.

2.2 OpenCL

OpenCL は，GPU などのヘテロジニアスなアーキテクチャ向けの並列化環境の一つである．OpenMP と同様に C/C++言語での開発が可能だが，ホスト側とデバイス側から成る 2 種類のプログラムを用意しなければならない点や，それらのプログラムを連携する際に多くの前処理と後処理を要する点が特徴的な並列プログラミングである．他の並列化環境と比べ，OpenCL は画像処理などの粒度の小さな並列化に適しているため，タスク並列よりもデータ並列においてその威力を発揮すると言われている [1]．また，特徴の一つとして OpenCL では再帰関数を扱うことができない.

2.3 実装方法

今回の実験では，再帰関数を利用できない OpenCL 環境において疑似的に再帰関数を実装するために，再帰処理におけるスタックの動作を配列等で再現する手法を用いて評価を行った.

まず, 総当たり法の巡回セールスマン問題は. 図 1 に示すような関数を再帰的に呼び出すことで実装するのが容易な方法の一つである.

つまり, 関数 `tsp01` を再帰呼び出しして未探索の都市を順に辿り, 探索の深さ (`nloc`) が都市数になるまで続けて, その時のコストを計算することを行なう.

しかし, 前述の通り, OpenCL では再帰関数は利用できないので, 図 1 のプログラムのままでは実行できない. そこで, 再帰呼び出しの呼び出し深さが最大で都市数であることを利用し, 呼び出しに必要な情報を都市数と同じ大きさの配列として宣言したデータ構造を利用する.

また, 最初およびその次に訪問する都市の組を固定し, それを並列の単位とする. 例えば, 都市数が 13 とすると, 最初の訪問都市が 13 通りで次に訪問する都市が 12 通りとなるので, 合計 156 通りの組合せがある. これらを並列実行可能なハードウェアに分散することで負荷分散を図る. CPU と GPU に 50%ずつの負荷分散の場合であれば, CPU が 78 通り, GPU が 78 通りを担当し, さらに CPU に 4 コアあれば, 各コアは約 19 通りずつ担当し, GPU に 24 個の execution unit あれば, それぞれが約 3 通りず

つ担当することになる. なお, 本研究の目的は, CPU と GPU の負荷分散の最適値を実行時に動的に決定することである.

```
1  int tsp01(int tmpc, int A, int B, int nloc, int size,int *visit, int *res){
2   ...
3    if(nloc==NCITY-1){
4      tmpc2=tmpc+cost[A][B]+cost[B][0];
5      return tmpc2;
6    }
7  ...
8    for(i=0;i<NCITY;i++){
9      if(visit[i]!=1){
10       ctmp=tsp01(tmpc2,B,i,nloc+1,size,visit,tres);
11       ...
12     }
13   }
14   return cmin;
15 }
```

図 1: 再帰関数による実装 (一部を省略)

2.4 自動チューニング

CPU と GPU の性能がアプリケーション毎に変わる可能性があるので, 実行前に最適な負荷を決定することは困難である. そこで, 本研究では計算の一部を事前実行し, その時の実行時間の結果により最適な負荷分散を決定することとする.

例えば, あるタスクを CPU で実行すると実行時間が t_{CPU} となり, GPU で実行すると実行時間が t_{GPU} となったとする. この場合, CPU が担当する負荷と GPU が担当する負荷を $t_{\mathrm{GPU}} : t_{\mathrm{CPU}}$ に分散することにより, 同じタスクが $\frac{t_{\mathrm{CPU}} \cdot t_{\mathrm{GPU}}}{t_{\mathrm{CPU}} + t_{\mathrm{GPU}}}$ の時間で実行できる. すなわち, この負荷分散に従えば, CPU の実行時間に対するスピードアップは $\frac{t_{\mathrm{GPU}}}{t_{\mathrm{CPU}} + t_{\mathrm{GPU}}}$ となることが期待される. ただし, 事前実行のタスクが大きくなれば, 最適な負荷分散によるハイブリッド並列実行の割合が減ることになるので不利になり, できるだけ事前実行のタスクは小さくすることが望ましい.

3 実験及び考察

3.1 実験環境

本研究では 2 種類の実験環境を用い, その仕様を表 1 に示す.

PC1 のプロセッサである i7-10700 は動作周波数が 2.9 GHz のインテルの第 10 世代のアーキテクチャ (Comet Lake) のプロセッサで 8 コア 16 スレッドであるのに対し, PC2 のプロセッサである i5-11400 は

動作周波数が 2.6 GHz のインテルの第 11 世代のアーキテクチャ(Rocket Lake) のプロセッサで 6 コア 12 スレッドである. また, UHD 630 も 730 も execution unit は 24 個であるが, 最大動作周波数が前者は 1.15 GHz であるのに対し, 後者は 1.3 GHz である. このように, プロセッサ毎に CPU と GPU の性能比が異なり, またアプリケーションの動作も異なるので, 負荷分散の最適値も異なることが予想される.

表 1: 実験環境

	CPU	GPU	memory	OS	framework
PC1	i7-10700	UHD Graphics 630	16 GB	Windows 10	OpenCL 2.1
PC2	i5-11400	UHD Graphics 730	8 GB	Windows 11	OpenCL 3.0

3.2 負荷分散実験

PC1 および PC2 において, CPU と GPU で探索する空間を分割し, 同時並行でハイブリッド並列実行した時のスピードアップを図 2 に示す. 縦軸は都市数 13 における CPU 単独 (OpenMP 環境) の実行速度を 1 とした場合のハイブリッド並列によるスピードアップ，横軸は CPU 及び GPU 全体の実行を 1 とした場合の GPU 側の実行比率を表している．つまり, 0.4 の場合は, GPU が 40%で CPU が 60%の計算を担当している.

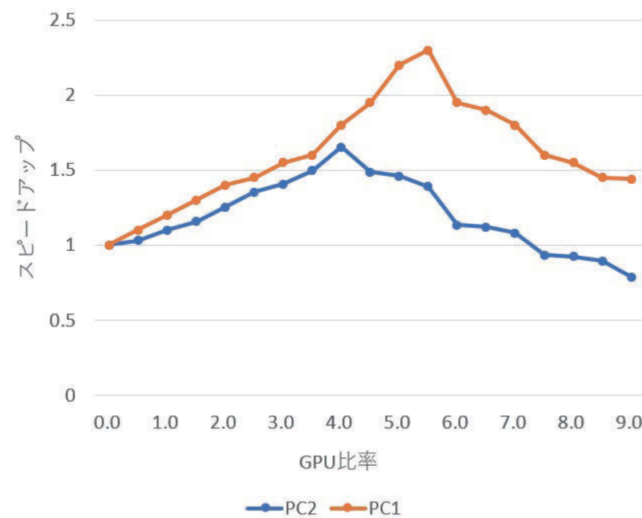

図 2: ハイブリッド実装 (都市数は 13)

PC1 と PC2 では CPU のみでの実行時間と GPU のみでの実行時間の比率が異なる. 実験からは, PC1 では GPU の方が CPU より約 1.5 倍高速で, PC2 では逆に GPU 実行が CPU 実行の約 0.75 倍であった. 従って, PC1 において最適な GPU 比率は 0.55 であり, PC2 において最適な GPU 比率は 0.40 であった. これらの比率は, PC1 と PC2 における CPU のコア数や動作周波数, もしくは, GPU の execution unit

の数や最大動作周波数などから事前に静的に求められるものではなく, マイクロタスクの事前実行によって決定されることが必要である.

3.3 自動チューニング実験

コスト最小な都市訪問順の探索を行うために, 全ての都市訪問順候補のコストを順番に計算し, その最小値を求めるが, 訪問順候補の一部の探索を事前実行とし, それの CPU での実行時間および GPU の実行時間から負荷分散を決定する. このときの事前実行の割合が大きければ正確な負荷分散が決定できるが, オーバーヘッドは大きい. 反対に事前実行の割合を小さくすれば正確な負荷分散がみつけられない可能性がある.

図 3 に, 都市数が 11, 12, 13 の場合の PC1 における事前探索割合と，事前探索の実行時間によって決定された負荷分散によるハイブリッド並列実行のスピードアップの関係を示す.

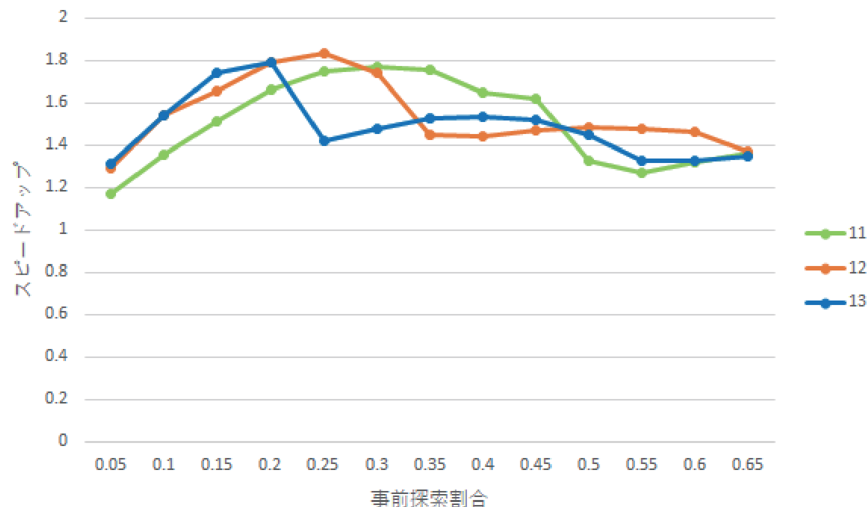

図 3: 自動チューニングによる CPU 単独，GPU 単独との性能比較

都市数が 11 の場合は事前探索割合を 30%の時に約 1.79 倍のスピードアップを達成し, 都市数が 12 の場合は事前探索割合を 25%の時に約 1.82 倍のスピードアップを達成し, 都市数が 13 の場合は事前探索割合を 20%の時に約 1.80 倍のスピードアップを達成している. 巡回セールスマン問題は都市数の階乗に比例した実行時間がかかるので, 都市数が増えるにつれて 10 倍以上の実行時間がかかる. よって, 都市数が少ない場合は実行時間が短いので, 短い事前実行では, GPU の起動時間やその他のオーバーヘッドのために, CPU と GPU の正確な実行時間比が得られにくい可能性がある. 従って, 都市数が多くなるにつれて, 事前実行の割合は少なくても, 最適な負荷分散が求められると考えられる.

また, 図 2 に示したように, 都市数 13 での PC1 におけるスピードアップの最大値は約 2.30 倍であったが, 事前実行を用いた自動チューニングにおけるスピードアップの最大値は約 1.80 倍である. これは, 事前実行部分がハイブリッド並列実行になってないことに起因するものである. よって, 正確な負荷分散の確定と得られるハイブリッド並列の効果はトレードオフの関係にあると考えられる.

4 おわりに

本研究では, GPU を内蔵しているプロセッサにおいて, 巡回セールスマン問題の総当たり法を CPU と GPU のハイブリッド並列実行によって性能向上する手法の評価を行った. 巡回セールスマン問題は再帰関数を用いて実装することが可能であるが, 本研究で用いるプログラミング環境の OpenCL では再帰関数が実装できず, 配列を用いて擬似的に再帰呼び出し動作をエミュレートすることで実現した. これにより, 巡回セールスマン問題をハイブリッド並列によって解く際の最適な負荷分散を求め, それによる性能の向上を確認できた．また, 自動チューニング実験についても，最適な負荷分散とほぼ一致する CPU と GPU の比率を実用性のある形で求めることができるプログラムを実装し, その有効性を示した.

しかし, 都市数が少ない場合は GPU 実行時の様々なオーバーヘッドを無視できず, 最適な負荷分散の推定ができなかったため, 改善する必要がある. また, その他のアプリケーションのハイブリッド並列実装やより簡便なハイブリッド並列のプログラミング方法の提案も今後の課題である.

謝辞

本研究の一部は JSPS 科学研究費 (基盤研究 (C) 18K02920 (2018-2023), 基盤研究 (C) 19K03018 (2019-2023), 基盤研究 (C) 23K02671 (2023-2025)) 及び私立大学等経常費補助金特別補助「大学間連携等による共同研究」による.

参考文献

[1] 北山洋幸, OpenCL2 入門 メニーコア CPU&GPGPU 時代の並列処理, カットシステム, 2017.

[2] 川崎 真之，大島 聡史，八巻 隼人，三輪 忍，本多 弘樹, ”OpenMP/OpenACC ハイブリッド並列化のためのコード変換フレームワークの提案”, 情報処理学会研究報告, Vol. 2022-HPC-187, 7 pages, 2021.

[3] https://jbpress.ismedia.jp/articles/-/47988. (As of 2023/5/18)

[4] https://future-architect.github.io/articles/20211201a/. (As of 2023/5/18)