2010 年 12 月 13 日 情報システム基盤学専攻合同輪講発表資料

OpenMPC:Extended OpenMP Programming and Tuning for GPUs

著者： Seyong Lee and Rudolf Eigenmann

出典： *Proceeding of the 2009 ACM/IEEE conference on SuperComputing 2010 PP.1-11*

発表者： 高性能コンピューティング学講座　1053018 長塚　郁

1　はじめに

GPU(Graphics Processing Unit) は高い演算性能と広いメモリバンド幅を持つため，N-body 問題や流体の数値計算など，様々な分野で活用されている．近年，Nvidia 社が C 言語ベースの統合開発環境 CUDA(Compute Unified Device Architecture) を公開し，専門的な知識を持たなくとも GPU のアプリケーション開発が行えるようになった．しかし，GPU の高い性能を引き出すにはメモリモデルなどの特有のアーキテクチャを意識したプログラミングを行う必要があるため，プログラマにとって敷居が高い．そのため，複雑な GPU プログラミングの簡易化が求められる．

そこで本研究では簡易な並列プログラミングが可能である OpenMP に注目した．OpenMP のソースコードを CUDA GPU のソースコードへ変換，また自動的に最適化を行う機構を先行研究 [1] で提案した．この論文ではその機構を基に，最適化の追加や OpenMP の拡張，動的な自動最適化機構を追加し，更なる性能向上を目指した．本研究ではその性能評価をベンチマークを用いて行う．

2　OpenMP と CUDA GPU

OpenMP と CUDA GPU では以下のような共通点を持つ．

- 繰り返し処理の並列化に強い．
- 並列性の追加によるアプリケーションの高速化が期待される．
- 親 OpenMP スレッドと子 OpenMP スレッドによる OpenMP の fork-join モデルは CUDA GPU における Host CPU と Device GPU による実行モデルに置き換える事が出来る．

一方で相違する点も存在する．その大きな点として OpenMP は単純であり，CUDA GPU は複雑である点が挙げられる．例えば For-Loop 処理を並列化することを考えることにする．

OpenMP の場合，For-Loop 処理の前に指示子文を置くことで，コンパイラ側が自律的に並列化を行う．そのため，プログラマは並列処理を意識することなく並列化ができ，また並列環境と非並列環境とのソースコードの違いは最小限となる．

それに対し CUDA の場合は，For-Loop 処理のイタレーションを分解し，GPU 上で実行する専用の関数 (以下カーネル関数) に置き換える．更に逐次処理部分にカーネル関数呼び出しを挿入し，その際に実行するスレッド数を指定する．また GPU と CPU ではメモリアドレスが独立しているため，GPU 上で実行するデータの転送が必要になる．このため，並列環境と非並列環境とのソースコードの違いは大きくなり，またプログラマ自身が明示しなければいけない部分が多く，負担になる．

3　OpenMPC

OpenMP のソースコードを CUDA GPU のソースコードに変換する．この時，OpenMP から CUDA GPU に変換する際には繰り返し処理のイタレーションの分解と分配，そしてデータのマッピングの手順が必要になる．OpenMPC ではそれらの必要な手順を以下のように行う．

1. OpenMP の処理の解釈
2. GPU 上で実行する領域 (以下カーネル関数領域) の引き出しとカーネル関数へ変換
3. GPU でアクセスされるデータの分析と転送命令の挿入

3.1　解釈

変換を行うために，OpenMP の処理が CUDA GPU 上で如何に動くのか解釈する必要がある．OpenMP の指示子を使い，以下のように解釈を行う

- omp parallel
  並列領域．コンパイラはこの領域内より，カーネル関数領域候補を発見する
- omp for, omp sections
  ワークシェアリング領域．並列性が強い処理が存在すると解釈し，カーネル関数領域の候補とする．
- omp barrier, omp critical, omp flush, etc...
  同期構造．並列領域の分割点と解釈され，分割された領域はカーネル関数領域になる

- omp shared, omp private, omp threadprivate
  データ特性の明示命令．GPU のデータ領域へのマッピングに使用され，その種類によってマッピングのされ方は区別される．

これらの解釈を基にコードの領域分割を行い，カーネル関数領域を見つける．この時，制御フローを崩壊させないようコンパイラは考慮をしながら領域分割を行い，カーネル関数領域を決定する．

3.2 変換

3.1 節でカーネル関数領域を抽出した．コンパイラは抽出されたカーネル関数領域をカーネル関数に変換し，GPU 上で実行できるようにする．その時，主な変換手順として以下の二つが挙げられる．

- Work partition
- Data mapping

Work partition では，カーネル関数領域内部の並列処理を CUDA スレッドに分割できるように分解する．このとき，それ以外の残ったカーネル関数領域の処理は動作している全てのスレッドで冗長に処理されることになる．またコンパイラは並列処理を分割するためのスレッドの必要数を解析する．この必要数が CUDA スレッドの数となり，初期設定の CUDA ブロックサイズから，CUDA ブロック数が決定される．このとき，コマンドラインや OpenMPC の拡張機能によりブロックサイズは変更できる．

Data mapping では，GPU 上でアクセスされるデータのマッピングを行う．その指標として，3.1 節のデータ特性明示の命令を用いる．この時，omp shared のデータメモリの場合は全てのスレッドで参照されるため，同じ属性を持つ GPU Global Memory に格納される．omp private の場合，一つのスレッドでのみ参照されるため，Register や Local Memory に格納される．omp threadprivate は一つのスレッドでのみ参照されるが，実行中は生存し続けるため，Global Memory に格納される．また必要に応じて，コンパイラは高速なオンチップメモリにデータを分配する。また，GPU 上でこれらのデータを使用するために，必要なメモリ転送や確保の命令を逐次処理部分に挿入する．

3.3 最適化

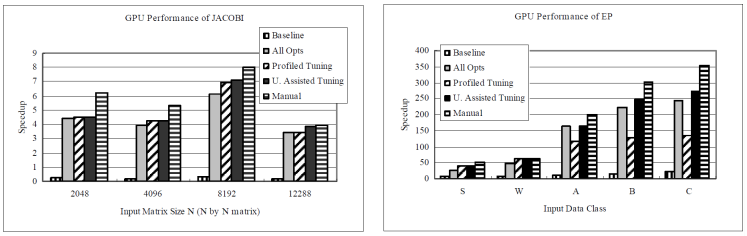
コンパイラでは Global Memory のアクセスの最適化や，メモリ資源の利用，CPU-GPU 間のメモリ転送の最適化を行う．また，OpenMPC では OpenMP の指示子や環境変数を拡張することで，プログラマにより細かな最適化をコンパイラに対して指示することが可能となっている．また，動的な自動最適化処理の追加により，さらに性能向上を図っている．

4 実験

4.1 内容

OpenMPC の性能評価を OpenMP 向けの並列ベンチマークを用いて行う．使用したベンチマークは JACOBI, NAS Parallel Benchmark EP である．また，実験で用いられている動的な自動最適化機構は試作段階である．

4.2 結果



(a) JACOBI Kernel　　(b) NAS Parallel Benchmark EP

Fig. 1: CPU に対する高速化率

Fig. 1 は OpenMPC によって出力された CUDA GPU ソースファイルを実行した際の 1 CPU 実行に対する高速化率である．この時，両方のベンチマークで，最適化を行わない Baseline の場合では性能値が低く出た．これは Global Memory のコアレスアクセスが行われていないためである．コンパイラの最適化によって，コアレスアクセスされるようになり，JACOBI では約 6 倍，EP では約 250 倍の高速化率を得られ，手動チューニングに近い性能値を得られた．

5 結論

本研究では，複雑な GPU プログラミングの簡易化のアプローチとして，OpenMPC を提案した．それにより，OpenMP による並列プログラミングによって CUDA GPU が可能となった．本研究の機構による CUDA GPU において，CPU に対し最大約 250 倍の高速化率を得ることができ，更に手動チューニングに近い性能値が得られた．

参考文献

[1] S. Lee, S.-J. Min, R. Eigenmann, OpenMP to GPGPU: A compiler framework for automatic translation and optimization, in ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPoPP), February 2009, pp.101-110.