Grid Computing

08M37344 渡辺祐也

取り上げる論文

- タイトル
  - Mars: A MapReduce Framework on Graphics Processor
- 著者
  - Bingsheng He, Wenbin Fang , Naga K. Govindaraju , Qiong Luo , Tuyong Wang
- 発表
  - PACT 2008

背景 MapReduce

- 大量のデータを処理したい
  - サーチエンジンなどのWebアプリケーション
- 複雑な環境
  - タスクが多様
  - ヘテロジニアスなコンピュータリソース
- 次のようなフレームワークが求められる
  - 開発者が簡単かつ効率的にプログラムを作れるフレームワーク
- MapReduce
  - Googleによって提案されたフレームワーク
  - 並列環境でのスケーラビリティがある程度保障される
- MapReduceのプリミティブ
  - map key/value → 中間のkey/valueに変換
  - reduce 同じkeyのペア全てをマージする

背景 GPU

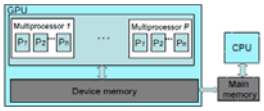
- GPUの特徴
  - 大雑把に言うとCPUの約10倍のピーク性能とメモリバンド幅をもっている
  - 大量のデータを扱う並列計算に向いている
- GPGPUプログラミング環境
  - ベンダーが提供　NVIDIA CUDA, AMD CTM
  - グラフィックAPI OpenGL, DirectX
- GPGPUの問題点
  - 非グラフィックAPIの環境を使用しても、GPUのアーキテクチャを知らなければ性能を発揮できない
  - CPUとGPUのアーキテクチャは大きく異なる
  - GPUアーキテクチャ自身もベンダーにより異なる
- Marsではプログラマーが API を実装することで動作

GPGPUのプログラム

- アプリケーション
  - 行列演算
  - 高速フーリエ変換(FFT)
  - バイオインフォマティクス
  - データベース
- プリミティブ
  - Gather/Scatter
  - Scan (Prefix Sum)
- 分散環境
  - Folding@home
  - Seti@home

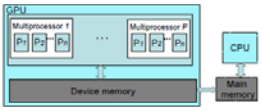
GPUのアーキテクチャ

- Deviceメモリー
  - バンド幅 86GB/s(GeForce 8800 GTX)
  - レイテンシ 約200cycle
- ディスクに直接アクセスできない
- 複数のMulti-processors
  - GPUはSIMD型のMulti-processorを複数もつ



GPUのアーキテクチャ

- Multi-processorsの内部構成
  - それぞれのMulti-processorは内部に複数の演算ユニット(SP)をもち、同じ命令を異なるデータに対して実行する(例えば、足し算を実行するときは全てのALUが足し算を行う)
  - ひとつのMulti-processorで実行されるスレッドはスレッドグループ(ブロック)にまとめられる
  - スレッドグループ内の全てのスレッドでマルチプロセッサーの資源(レジスタやMulti-processor内のメモリ,SP)を共有する



MapReduce

Map: $(k_1, v_1) \rightarrow (k_2, v_2)^*$.
Reduce: $(k_2, v_2^*) \rightarrow v_3^*$.

- Map
  - 全ての入力(k1, v1)に対してMapを適用して出力(k2, v2)を作る
- Reduce
  - 同じkeyのvalueに対してReduceを適用する
- 一般的なMapReduceフレームワークでは、プログラマはMapとReduceを定義する
- MapとReduceの多数のタスクは並列に実行される

MapReduceの例

- ドキュメント内の単語の数を数えるプログラム
- Mapでは、単語を区切り(word, 1)というペアを作る
- Reduceでは単語の数を数える
- EmitIntermediateとEmitはライブラリが提供する関数

```
// the input is a document.
// the intermediate output: key=word; value=1.
Map(void *input) {
for each word w in input;
    EmitIntermediate(w, 1);
}
// input: key=word, the value list consisting of 1s.
// output: key=word; value=occurrences.
Reduce(String key, Iterator values) {
int result = 0;
for each v in values
    result += v;
Emit(w, result);
}
```

プログラムの設計 Map

- Map stage
  - Input Dataをいくつかのチャンクに分割する
    - チャンクの数とGPUスレッドの数を等しくする
  - スレッドがチャンクに対してMapを適用する
  - Mapが終わった後、データをソートする
  - Map stageが終わった後、ソートされたkey/valueのペアが生成される

プログラムの設計 Reduce

- Reduce stage
  - Map stageで生成されたkey/valueのペアをいくつかのチャンクに分割する
    - 同じkeyを持つペアは同じチャンクにする
    - チャンクの数とGPUスレッドの数を等しくする
  - スレッドがチャンクに対してReduceを適用する

プログラムの設計　オプション

- プログラムの動作を細かく制御するためのオプションが設定されている
- Reduce stageやSortを行わないプログラムを作れる
- 下の4つはGPUの動作を制御するためのオプション
  - スレッドグループ(block)の数を指定するオプション
  - スレッドグループ(block)内のスレッドの数を指定するオプション

Parameter	Description	Default
noReduce	Whether a reduce stage is required (If it is required, *noReduce*=F; otherwise, *noReduce*=T).	F
noSort	Whether a sort stage is required (If it is required, *noSort*=F; otherwise, *noSort*=T).	F
tgMap	Number of thread groups in the Map stage.	128
tMap	Number of thread per thread group in the Map stage.	128
tgReduce	Number of thread groups in the Reduce stage.	128
tReduce	Number of thread per thread group in the Reduce stage.	128

出力データのサイズと書き込み位置

- GPUで実装するときの問題点
  - Map、Reduceそれぞれの出力結果のサイズをkernel実行前に知っておく必要がある
    - GPU実行時にデータを動的に確保できない
    - 各スレッドごとの書き込み位置を正しく指定する必要がある
- 解決方法
  - Map、Reduceそれぞれに結果のサイズを計算できる関数をユーザに定義させる
  - 出力を2stepで実行する
    - スレッドごとに結果のサイズを計算する
    - データをGPU上に確保する

Mapでの詳しい動作

1. 各Map関数ごとに、それぞれの出力するkeyとvalueのサイズを出力する
   1. Keyとvalueごとにサイズの和を計算しKeyとvalueの出力用の配列をGPU上に確保する
   2. Keyとvalueごとにprefix sumを求める（この値が書き込みスタート位置になる

   (例えば、Map関数の出力サイズが3,5,6,1, 3になる場合、prefix sumは[0, 3, 8, 14, 15]になる）
2. Mapを適用し、結果を書き込む

API

- ユーザは最大で5個の関数を実装する必要がある
  - オプションによっては2つ
- MAP_COUNTとRECUDE_COUTN関数はCPUヴァージョンのMapReduceフレームワークでは必要ない関数
  - 前のスライドで説明した結果のサイズを返す関数

APIs of user-defined functions	*Optional*
`void MAP_COUNT(void *key, void *val, size_t keySize, size_t valSize)` Counts the size of output results (in bytes) generated by the map function.	no
`void MAP(void *key, void val, size_t keySize, size_t valSize)` The map function. Each map task executes this function on the key/value pair.	no
`void REDUCE_COUNT(void* key, void* vals, size_t keySize, size_t valCount)` Counts the size of output results (in bytes) generated by the reduce function.	yes
`void REDUCE(void* key, void* vals, size_t keySize, size_t valCount)` The reduce function. Each reduce task executes this function on the key/value pairs with the same key.	yes
`int compare(const void *d_a, int len_a, const void *d_b, int len_b)` User-defined function comparing two keys.	yes

実装テクニック(1/2)

- Thread parallelism
  - スレッドグループとスレッドの数はプログラムによって最適な値が異なる
    - Memory-intensiveなアプリケーションでは多いほうが良い
  - 直接オプションで指定できる
  - 評価用のプログラムでは、コンパイル後のGPUコードを分析して、レジスタの使われ方からスレッドグループとスレッドの数を決定
- Coalesced accesses
  - スレッドがある条件を満たすメモリ上の連続領域をアクセスすると、転送速度が速くなる
  - この条件を満たしやすいように、入力を1列に並べ、スレッド数でタスクを分割した
- Build-inタイプの使用
  - コンパイラがCoalesced accessesに変換してくれる可能性がある

実装テクニック(2/2)

- 可変長型の取り扱い
  - 独自に可変長の型を取り扱うライブラリを作成
  - 特に、stringを取り扱うライブラリを作成
- Sort
  - Bitonic sortを使用
  - 正しくソートする必要がない場合はhash値でソートする

評価　実験環境

- Mars（この論文の実装）、Phoenix(C言語でMapReduceを実装）で実験
- GPUはG80コアのNVIDIA GeForce 8800 GTX
- CPUはCore2Duo Quad
- OSはFedora 7.0

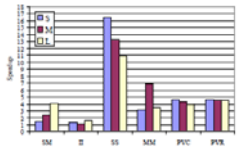
	GPU	CPU
Processors	1350MHz × 8 × 16	2.4 GHz × 4
Data cache (shared memory)	16KB × 16	L1: 32KB × 4, L2: 4096KB × 2
Cache latency (cycle)	2	L1: 2 , L2: 8
DRAM (MB)	768	2048
DRAM latency (cycle)	200	300
Bus width (bit)	384	64
Memory clock (GHz)	1.8	1.3

評価アプリケーション

- Web dataの会席に使われる6つのアプリケーション
- それぞれにS,M,Lの3サイズのデータセット
- Matrix MultiplicationとSimilarity Scanはcompute-intensive
- Page View CountはMap stageが2回ある

App.	Description	Data sets
String Match	Find the position of a string in a file.	S: 32MB, M: 64 MB, L: 128 MB
Inverted Index	Build inverted index for links in HTML files.	S: 16MB, M: 32 MB, L: 64MB
Similarity Score	Compute the pair-wise similarity score for a set of documents.	#doc: S: 512, M: 1024, L: 2048. #feature dimension: 128.
Matrix Multiplication	Multiply two matrices.	#dimension: S: 512, M: 1024, L: 2048
Page View Count	Count the page view number of each URL in the web log.	S: 32MB, M: 64 MB, L: 96 MB
Page View Rank	Find the top-10 hot pages in the web log.	S: 32MB, M: 64 MB, L: 96 MB

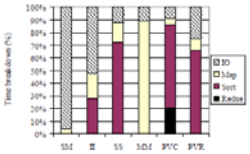
MarsとPhoenixの比較



Phoenixのスピードを1としたときのMarsのスピード

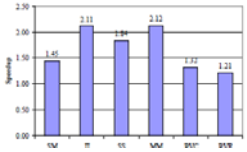
- Compute-intensiveアプリケーション(Matrix Multiplication,Similarity Scan)のとき、特にMarsは高速
- Memory-intensiveアプリケーション(Inveted Index,Sgring Matchでも、わずかにMarsのほうが高速

Marsの各過程でも実行時間の割合



- 実行時間をIO, Map, Sort, Reduceに分割
  - Mapステージは入力データをGPUメモリにコピーする時間を含む
  - Reduceステージは結果をメインメモリに書き戻す時間を含む
- Memory Intensiveなアプリケーションではio時間が多い
- それ以外のアプリケーションではMap, Sort, Reduceの占める時間が多い

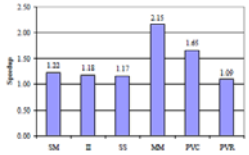
Coalesced accessesによるスピードアップ



Coalesced accessesの最適化を行わなかったときのスピードを1とする

- Build-in vector型は使用しない
- 1.2倍から2.1倍高速化

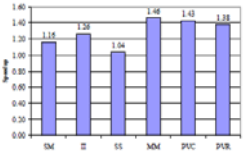
Build-inのベクター型を使用した場合のアクセス速度の向上



- Build-inベクター型を使用しなかった場合を1
- Build-inベクター型を使用するとメモリリクエストの回数が減る
- 1.09-2.15倍の性能向上　データのアクセス頻度によって性能改善の度合いが違う

CPUとGPUの併用

- CPU(Phoenix)とGPU(Mars)を併用
- GPU(Mars)を単独で使用した場合の性能を1とする
- タスクの分割率は単独で実行した場合の比
- SSが遅いのはCPUが遅かったことが原因



まとめ

- MapReduceフレームワークMarsを使うことでGPUの複雑さを隠蔽できる
  - ただし、ユーザー定義関数を実装するときはGPUのコードを書く必要がある
- 今後の課題
  - Hadoopなどの既存のフレームワークとの統合