2009年11月9日 情報システム基盤学専攻合同輪講発表資料

Debugging GPU Stream Programs Through Automatic Dataflow Recording and Visualization

著者： Qiming Hou, Kun Zhou, Baining Guo
出典： *ACM Transactions on Graphics (SIGGRAPH Asia 2009), pp.1-12, 2009*
発表者： 本多研究室 0953009 佐々木 信

1 はじめに

近年GPUは汎用計算にも広く用いられている。中でもCUDAのようなC-Likeな言語によってストリームプログラムの難しさは減少した。[1]

しかし、ストリームプログラムのデバッグには問題が残っている。

まず、複雑なストリームプログラムは普通複数のカーネルと大量のストリームが必要なので、データフローを通して、一つのカーネル(GPUで実行される関数)のエラーが、一見無関係で、完全に正しいカーネルに伝搬してしまう可能性がある。そのような場合、出力を可視化しただけでは元のエラーソースを見つけることができない。そのためプログラマーは、すべての可能性のあるカーネルを手作業で分析しなくてはならない。

また、レースコンディションのようなデータフローエラーは、ストリームプログラムにとっては脅威となる。レースコンディションと関連したエラーは非決定的で再現することが難しいので、手作業で見つけるのは難しい。

本研究では、CUDAにおいて、データフローを自動的に記録し、可視化することでGPUストリームプログラムをデバッグする新しいフレームワークを提案する。このデバッグのフレームワークは、既存のツールでは発見が難しいエラーを見つけるのを支援する。

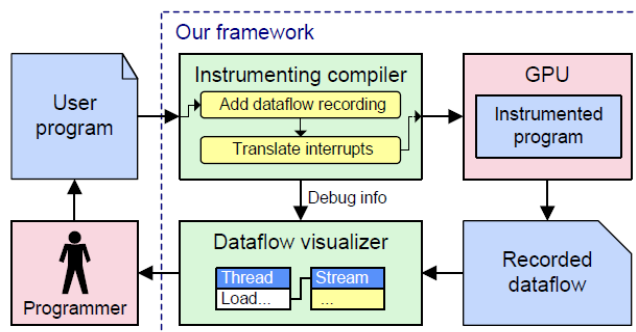

図1: 本研究のデバッグフレームワーク概略図

[1] 本研究において、ストリームプログラムとは、依存関係のない多数のスレッドに対して、メモリや命令のレイテンシを隠蔽するために、任意の順番でスレッドのスケジューリングを行うマルチスレッドプログラムのこと。

2 GPU割り込み

本研究では、GPUのデータフローをCPU側で記録したいので、これをGPUからCPUの機能を呼び出す(割り込み)ことで実現する。CUDAでは構文として、CPUからGPUの機能を呼び出す`__global__`、GPUからGPUの機能を呼び出す`__device__`、CPUからCPU関数を呼び出す`__host__`が提供されている。しかし、GPUからはCPUの機能を呼び出すことができない。そこで本研究では、GPUからCPUの機能を呼び出す構文として`__interrupt__`を新たに加える。

2.1 コード生成における課題

同じカーネル内でも、それぞれのストリームを実行するスレッドの状態は異なる可能性がある。しかし、現在GPUにおいて、GPUからCPUにコントロールを移す唯一の方法は、カーネルを終了させることである。それゆえ、コンパイルアルゴリズムは全てのスレッドを一緒に終了さる必要がある。これは次の2つの課題に帰着する。

1. 割り込み処理メカニズムはオーバーヘッドを最小にするように設計しなくてはならない。
   そのために、ＧＰＵ上で同じ割り込み状態のスレッドをグループ化し、同じグループの全てのスレッドからのパラメータを統合、それらをＧＰＵ一時ストリームとして割り込みハンドラに渡す
2. 割り込まれたスレッドの再開する。
   そのために、割り込みに達成するか、通常終了するとき、各スレッドはグループＩＤを保存する。スレッドが再起動するとき、カーネルの初めにスイッチ文を実行し、対応するアドレスにジャンプする。

2.2 コード生成のアルゴリズム

前述のような課題を考慮したコード生成のアルゴリズムは次のようになる。

1. 割り込み呼び出しを変換
2. カーネル再起動時に割り込み地点にジャンプするスイッチ文を加える
3. 割り込み状態が同じスレッドをグループ化するコードを加える

4. 割り込みハンドラを CPU の関数として呼び出すコードに変換
5. カーネル内のすべての実行が終わるまで、カーネルを繰り返し起動しすようなループを加える

3 Debugging System

デバッグの目標は誤ったコードフラグメントを見つけ、その誤りにより生じた問題を解決することである。そのためには、見えるエラーから元のエラー源までトレースするメカニズムが必要である。この目的のために、データフローの記録と可視化を実装する。

3.1 Dataflow Recording

コンパイラは以下の情報を自動的に記録するコードを挿入する。

- スレッドメモリのアクセス履歴
- ストリーム情報
- ストリーム初期化ステータス
- ソースコードの位置

3.2 Dataflow Visualization

プログラムが終了したら、データフローを解析するために、データフロービジュアライザーが起動する。もしエラーが検出されたら、ビジュアライザー起動直後にエラーを引き起こしているスレッドに対応するウィンドウが開く。表示されているデータフローを確認することで、エラーを見つけることができる。

ここで実際のアプリケーションのデバッグプロセスの例を示す。図 2 は mergesort のソースコードである。これをデバッグすると、アプリケーション終了時、データフロービジュアライザーが起動し図 3 のようなウィンドウが表示される。これを見るとこのアプリケーションはレースコンディションが発生していることが分かる。このエラーは、スレッド 4012 と 4013 の両方が、結果を b[4013] に書き込むことから起きている。その原因は、a[3012] に 0x80000000、つまり int 型の最小値が入っているため、図 2 の 9 行目の引き算で整数オーバーフローが発生しているからである。このようにデータフローを可視化することで、原因の分かりにくいエラーを見つけることができる。

```
1  __global__ void bsmerge(int* b,int* a,int n,int sz){
2      //@Binary search merge pass
3      int id=blockIdx.x*blockDim.x+threadIdx.x;
4      if(id>=n)return;
5      //examine input
6      int k0=a[id];
7      debug::watch(k0);
8      //enforce left<right order on equal numbers
9      int k=k0-!(id&sz);
10     //binary search in neighboring sz elements
11     int l0,l,r;
12     l0=l=((id&-sz)^sz);r=min(l+sz,n)-1;
13     while(l<=r){
14         int m=(l+r)>>1;
15         int rk=a[m];
16         if(rk<=k)l=m+1;else r=m-1;
17     }
18     //(l-l0) elements less than k0 in the neighbor
19     b[(id&~sz)+(l-l0)]=k0;
20 }
21
22 void msort(int* b,int* a,int n){
23     int nb=((n+255)>>8);
24     int* tar=b;
25     for(int sz=1;sz<n;sz+=sz){
26         int* tmp;
27         bsmerge<<<nb,256>>>(b,a,n,sz);
28         tmp=a;a=b;b=tmp;
29     }
30     if(tar!=a){cuMemcpyDtoD(tar,a,n*sizeof(int));}
31 }
```

図 2: mergesort のソースコード

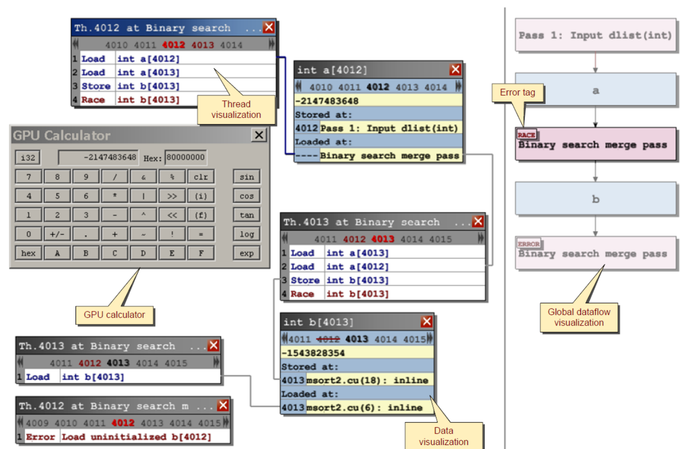

図 3: dataflow visualizer

しかしながら、データフローを考慮に入れておらず、データフローによって伝搬するエラーはデバッグできない。

[Boyer et al.2008] の研究では、自動的に CUDA 共有メモリの競合状態とバンクコンクリフトを検出できるが、1 つのカーネル内での共有メモリ関連の分析に限られている。また、グローバルデータフローは無視される。加えて、エミュレーションモードで実行する必要があるので、実際のＧＰＵ上での実行より最大 800 倍の時間がかかることが分かっている。

4 関連研究

CUDA framework[NVIDIA 2008] は、デバイスエミュレーションに基づく組み込みデバッガを提供している。しかし、CPU エミュレーションでは GPU を用いて実行するよりも数百倍遅く、結果としてリアルタイムアプリケーションはオフラインでしかデバッグできない。また、GPU 上で起こるエラーのいくつかは CPU エミュレーションでは再現できない可能性がある。

CUDA-GDB[NVIDIA 2008] は、GPU 上で直接デバッグできるので、エミュレーション関連の問題を解決した。

5 まとめ

本研究のシステムは汎用計算ストリームプログラムのデバッグや、既存のツールでは見つけるのが難しいエラーを見つけるのに非常に効率的である。