Thrust拡張ライブラリThrustingの開発

Akira Hayakawa
@akiradeveloper
<ruby.wktk@gmail.com>

Thrusting@bitbucket
http://bitbucket.org/akiradeveloper/thrusting/wiki/Home

そもそもThrustって？

- http://code.google.com/p/thrust/
- CUDAによる実装のありふれたしょうもない部分を全部隠蔽.
- STLの<algorithm>をCUDA上でほぼ実装.
  関数型プログラミングによって明瞭な並列コードが書ける.

アルゴリズムに集中出来る.

- ソフトウェア開発において理論的に「出来る」ことと現実的に「出来る」ことには乖離がある.
- また,「出来る」ことと「していい」ことの間にも差がある. ごちゃごちゃ書き散らかされたコードはゴミに等しい（[2]で負の遺産と呼ばれている）.
- 複雑なソフトウェアは理論的に出来ることでも実装は不可能になりうる.
- 並列とかまじでそう. なので抽象化の力（[3]の2章参照）を使う.

ほとんどのCUDAコードはベストからほど遠いはず.

- CUDAさわってせいぜい数年の凡人が高速な並列コードを書けるわけがない（というかアホが1000年やっても無理）.
- 実際にCUDAで実装されてるアルゴリズムは明瞭でしょぼいものばかり.
- じゃないと実装出来ないから.

引用文（[2]より）

言い換えると、プログラマーが計算機の身になって並列性を生かす場所を指定しているため、それが有効に生かされるかどうかは、プログラマーの力量に委ねられている。膨大なデータ量を高速で計算するHPCなど高性能の並列計算機用のプログラム開発を、このような手法で補っている限り抜本的な解決にはならず、かえって負の遺産を増やしかねない。並列性を「書くことができる」のと「分かりやすくきれいに書く」というのは違う。

ならば関数による抽象化だ.

- Thrustによって高レベルのコードを書ける. 高度なアルゴリズムを実装出来る. 高い保守性, 高い生産性を実現出来る（[I]参照）.
- 関数型プログラミングによって, 並列性を隠蔽することが可能になるはず.
- こういう趣旨の研究をさきほどの引用元は言っている. 並列化を意識せずに並列化が出来たら超すごい.

パフォーマンス？

- 汎用ライブラリで心配なのは性能.
- Thrustのドキュメントでも性能について言及されていて高い性能を出せている.
- そもそも我々素人がいくらコードで最適化を行ってもベストからはほど遠い（コードの字面と実際のパフォーマンスが一致するならば苦労はない）.
- Thrustの開発者はNVIDIAの人. そこはプロに任せよう.

まとめると

- Thrustを使うと, 並列性を意識せずに並列コードを書くことが出来る可能性がある.
- 並列性を意識せずというのは人類にとってとても重要な課題. 未来のソフトウェアの課題.
- しかもこのCUDAでの取り組みはまだメジャーではないのでやる価値がある.
- 実アプリでパフォーマンスが出るかどうかもこれから物理シミュレーションでThrustが活用されていくかどうかで試される.

ここからが本題.

- なぜ,Thrustではいけないのか. なぜ拡張する必要があるのか.
- 一体, 何を提供しているのか.
- はたして, 信頼出来るものなのか（信頼出来るものでなければ使うことは出来ない）.

Thrustじゃ不満なの？

- ええ. 発狂するくらい不満です.
- データ型, それらを生成する関数, 他ライブラリとの連携インターフェイス, 物理シミュレーションで必要な関数, 関数型プログラミングに必要な関数合成など全部ないです.
- ふつうに使うとコードが腐敗する.
- たぶん最小限の部分にしか開発が出来ないほど時間に切迫してるのだと思う. あるいはめんどくさいか.

関数が足りないについては

- ThrustのRoadmapで言われている（http://code.google.com/p/thrust/wiki/Roadmap）.
- 「関数が足りないから実装しないと」.
- tr1/hashがどうとか言われてるけどこれは実装すべきものの一つという意味だと思う.

。

参考までに引用文

（Thrust Roadmapより）

Flesh out functional.h with more functions

- tr1/hash: http://www.boost.org/doc/libs/1_39_0/doc/html/boost_tr1/subject_list.html#boost_tr1.subject_list.hash
- Challenge: 2/5

じゃあおれが実装しよう！（以下はThrustingのソースコードより引用）

```
// (a,b)->c -> a->b->c
template<typename F>
struct _curry :public thrust::binary_function<
typename thrust::tuple_element<0, typename F::argument_type>::type,
typename thrust::tuple_element<1, typename F::argument_type>::type,
typename F::result_type> {
  F _f;
  _curry(F f)
  :_f(f){}
  __host__ __device__
  typename F::result_type operator()(
  const typename thrust::tuple_element<0, typename F::argument_type>::type &a,
  const typename thrust::tuple_element<1, typename F::argument_type>::type &b) const {
    return _f(thrust::make_tuple(a, b));
  }
};

template<typename F>
_curry<F> curry(F f){
  return _curry<F>(f);
}
```

curryはHaskellの関数！

インドの食べ物ではない！

- http://zvon.org/other/haskell/Outputprelude/curry_f.html
- uncurryも実装した. 意味的には微妙だけどｗ
- 他には, bind1st, bind2ndなどSTLでおなじみの関数や関数合成compose, 二項演算子の引数順序を変えるflipなども実装した.
- でもたぶん実装は穴だらけだと思う（どこが穴なのかも分からないので直せない）.

SoA VS AoS

- Structure of Arrays or Array of Structure（[1]にも記載）.
- SoAの方が高速. 理由はコアレッシング.
  - 例えば, CUDAのサンプルプログラムではfloat3のかわりにfloat4を使うことで対処している.
  - しかし結局バンクコンフリクトを起こす（http://blogs.yahoo.co.jp/natto_heaven/8534195.html参照）.

ということは,Thrustを使った実装は

- SoAに制約される（じゃないとパフォーマンス出ない）.インターフェイス設計などすべてがSoAに制約されることになる.
- CUDAが提供する.floatNとか使った時点で負け.

おれの黒歴史（float3がダメならfloat4にすればいいと思ってた時期がありました）.

しかもスペル間違ってるじゃんorz

```
typedef float3 Real3 //
conceal Real3 is float3 or
float4 for coarescing
```

つうわけでTupleを使う.

- Thrustのzip_iteratorはコアレッシングをエレガントに解決.
- だからfloat3とかtuple<float, float, float>で置き直して使う方がいい.
- thrusting::typeNを定義. これはタプルのエイリアス.

コード（Thrustingより）

```
#pragma once
#include <thrusting/tuple.h>
namespace thrusting {

typedef typename tuple2<float>::type float2;

typedef typename tuple3<float>::type float3;

typedef typename tuple4<float>::type float4;

typedef typename tuple5<float>::type float5;

typedef typename tuple6<float>::type float6;

typedef typename tuple7<float>::type float7;

typedef typename tuple8<float>::type float8;

typedef typename tuple9<float>::type float9;

}
```

ついでに演算子も定義.

- 物理シミュレーションでは必要. タプルの四則演算とか出来るといい.

- tuple + tuple, x * tupleなどを定義.
- 等価判定, 標準出力なども定義（後述するように, googletestを使うために）.

（禁止）車輪の再開発. 才能の無駄遣い

（http://code.google.com/p/astro-attic/より）

```
///vector addition
template<class t>
  vector2<t> operator+(const vector2<t> & a, const vector2<t> & b){
  return vector2<t>(a.x + b.x,a.y + b.y);
}

///vector subtraction
template<class t>
  vector2<t> operator-(const vector2<t> & a, const vector2<t> & b){
  return vector2<t>(a.x - b.x,a.y - b.y);
}

///vector negate
template<class t>
  vector2<t> operator-(const vector2<t> & a){
  return vector2<t>(-a.x,-a.y);
}

///scalar multiplication
template<class t>
  vector2<t> operator*(const t &m, const vector2<t> & a){
  return vector2<t>(m*a.x,m*a.y);
}
template<class t>
  vector2<t> operator*(const vector2<t> & a, const t &m){
  return vector2<t>(m*a.x,m*a.y);
}

///scalar multiplication by reciprocal of \a m
template<class t>
  vector2<t> operator/(const vector2<t> & a, const t &m){
  return vector2<t>(a.x/m,a.y/m);
}
```

おつかれさまです.

これらはほとんど自動生成.

- Rake（http://www2s.biglobe.ne.jp/~idesaku/sss/tech/rake/）を使っている.
- Rubyの力を生かしてシンプルに生成コードを書いてRakeでタスクを生成している.
- あとC++のテンプレートメタプログラミングも少しだけ使っている.

コード例（タスクの自動生成がキマってる）.

```
def gen_namespace(dir)
  namespace dir do
    file = "#{dir}/Rakefile"
    task :build =>  file do
      Dir.chdir(dir) do
        sh "rake build"
      end
    end
    task :clobber => file do
      Dir.chdir(dir) do
        sh "rake clobber"
      end
    end
  end
end
```

コード生成スクリプト例（mapとかjoinとか使いまくる）.

```
def operator(n, op)
  input = (0...n).map { |i| "#{get_tuple(i, "x")}#{op}#{get_tuple(i,
"y")}" }
"""
template<#{typename(0...n)}>
__host__ __device__
thrust::tuple<#{type(0...n)}> operator#{op}(const thrust::tuple<#{type
(0...n)}> &x, const thrust::tuple<#{type(0...n)}> &y){
  return thrust::make_tuple(#{input.join(", ")});
}
"""
end

def all()
  ops = ['+', '-']
  code = (TUPLE_MIN..TUPLE_MAX).map { |i| ops.map { |op| operator(i,
op) } }.join
```

実際に生成するところ.

```
Hayakawa-Akiras-MacBook-Pro:detail akira$ rake build
(in /Users/akira/sandbox/thrusting/thrusting/detail)
rake build
(in /Users/akira/sandbox/thrusting/thrusting/detail/dtype)
rake build
(in /Users/akira/sandbox/thrusting/thrusting/detail/list)
rake build
(in /Users/akira/sandbox/thrusting/thrusting/detail/tuple)
ruby /Users/akira/sandbox/thrusting/thrusting/detail/tuple/rb/make_string.rb > /Users/akira/sandbox/thrusting/thrusting/detail/tuple/make_string.h
ruby /Users/akira/sandbox/thrusting/thrusting/detail/tuple/rb/make_tuple_n.rb > /Users/akira/sandbox/thrusting/thrusting/detail/tuple/make_tuple_n.h
ruby /Users/akira/sandbox/thrusting/thrusting/detail/tuple/rb/tuple_equality.rb > /Users/akira/sandbox/thrusting/thrusting/detail/tuple/tuple_equality.h
ruby /Users/akira/sandbox/thrusting/thrusting/detail/tuple/rb/tuple_n_operator.rb > /Users/akira/sandbox/thrusting/thrusting/detail/tuple/tuple_n_operator.h
ruby /Users/akira/sandbox/thrusting/thrusting/detail/tuple/rb/tuple_n_typedef.rb > /Users/akira/sandbox/thrusting/thrusting/detail/tuple/tuple_n_typedef.h
ruby /Users/akira/sandbox/thrusting/thrusting/detail/tuple/rb/tuple_operator.rb > /Users/akira/sandbox/thrusting/thrusting/detail/tuple/tuple_operator.h
ruby /Users/akira/sandbox/thrusting/thrusting/detail/tuple/rb/tuple_ostream.rb > /Users/akira/sandbox/thrusting/thrusting/detail/tuple/tuple_ostream.h
rake build
(in /Users/akira/sandbox/thrusting/thrusting/detail/real)
ruby /Users/akira/sandbox/thrusting/thrusting/detail/real/rb/make_real_n.rb > /Users/akira/sandbox/thrusting/thrusting/detail/real/make_real_n.h
ruby /Users/akira/sandbox/thrusting/thrusting/detail/real/rb/real_n_typedef.rb > /Users/akira/sandbox/thrusting/thrusting/detail/real/real_n_typedef.h
rake build
(in /Users/akira/sandbox/thrusting/thrusting/detail/iterator)
ruby /Users/akira/sandbox/thrusting/thrusting/detail/iterator/rb/zip_iterator.rb > /Users/akira/sandbox/thrusting/thrusting/detail/iterator/zip_iterator.h
```

float/doubleを隠蔽.

```
#pragma once

namespace thrusting {
#ifndef THRUSTING_USE_DOUBLE_FOR_REAL
  typedef float real;
#else
  typedef double real;
#endif
}
```

（禁止！）車輪の再開発. 才能の無駄遣い！

（http://code.google.com/p/astro-attic/より）

もう泣き寝入りするしかないのか.

```
typedef float Real;
```

vectorの型も隠蔽

```
#pragma once

#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

#ifdef THRUSTING_USE_DEVICE_VECTOR
  #define THRUSTING_VECTOR thrust::device_vector
#else
  #define THRUSTING_VECTOR thrust::host_vector
#endif
```

vectorの型を隠蔽する？

- Thrustのアルゴリズムはvectorの型に対して多態的（タグディスパッチという手法を使っている. http://www.issei.org/programming/boost/more/generic_programming.htm#tag_dispatching）.
- だから, クライアントのプログラムでvectorの型を書き換えるだけでhost_vectorとdevice_vectorを切り替えることが出来る.
- それをマクロでやりましょうという試み.

で, なにがいいの？

- GPU/CPUパフォーマンスに対して比較に中立的（プログラムをほとんど書き換えないから. それぞれのオブジェクトに対してThrustは最大限のパフォーマンスを努力する）.
- 現行の研究者たちが言ってるGPU/CPU比較というのはCPUに対する最適化が弱すぎるでしょう. 彼らの興味はGPUなので贔屓しているに決まってる. ほとんど参考にならない（こういう意見の人もいる. http://am112705.blog.so-net.ne.jp/archive/c2300742966-1）. しかしThrustを使った比較は妥当である. だって, ThrustはOpenMPを自動で適用してくれるもの（参考http://code.google.com/p/thrust/wiki/DeviceBackends）.

テストをしましょう.

- googletest（http://code.google.com/p/googletest/）を使用している.
- 単体テストは存在は現代的なソフトウェアにおいて信頼性の最低条件でしょう（Thrustでは600を超える自動テストを定義してる. 彼らは独自のテスティングフレームワークを開発している）.
- 公開インターフェイスに対しては網羅的にテストしている（ただし, 自動生成によって相似関係が自明な部分は一つだけテストして他はテストしない方針）.
- 実装を変更したあとに回帰テストを行うことで仕様の保存を確認する. すでに開発の現段階でその恩恵を得ている.

テストのために.

- Thrustだけではgoogletestでのテストは出来ない.
- ほうりこむオブジェクトはoperator==と ostream& <<を定義している必要がある.
- list(head, len)という型を定義（vectorの切り出しのようなもの）して, これらの演算子を定義している.Thrustingを使うことでgoogletestを使ってCUDAのプログラムが容易にテスト可能になる.

Thrustingにおけるgoogletest

使用例.

```
TEST(Functional, UnCurry){
  // 2 + 3 = 5
  int x = thrusting::uncurry
(thrust::plus<int>())
(thrust::make_tuple(2,3));
  EXPECT_EQ(5, x);
}
```

googletestの実行

```
[ RUN      ] Tuple.MakeString
[       OK ] Tuple.MakeString (0 ms)
[ RUN      ] Tuple.OStream
(1, 2)
[       OK ] Tuple.OStream (0 ms)
[ RUN      ] Tuple.ArithmaticOrdering
[       OK ] Tuple.ArithmaticOrdering (0 ms)
[----------] 10 tests from Tuple (0 ms total)

[----------] 11 tests from Functional
[ RUN      ] Functional.Flip
[       OK ] Functional.Flip (0 ms)
[ RUN      ] Functional.Multiplies
[       OK ] Functional.Multiplies (0 ms)
[ RUN      ] Functional.divides
[       OK ] Functional.divides (0 ms)
[ RUN      ] Functional.LeftShift
[       OK ] Functional.LeftShift (0 ms)
```

以前は出来なかった.

- 参考 http://forums.nvidia.com/index.php?showtopic=150025
- nvcc 2.3, gcc 4.3.2ではデバイスコードとホストコードを手動で分離する必要があった.
- nvcc 3.2, gcc 4.2.1では, cuファイルのままテストすることが可能に. よりテストを書きやすくなった（テストは負債になりやすいので保守性も重要）.
  NVIDIAさんありがとう.

引用

- [1] An Introduction to Thrust. http://code.google.com/p/thrust/
- [2] 並列プログラム開発を抜本的に改革. http://www.i.u-tokyo.ac.jp/news/focus/100215_1.shtml
- [3] SICP http://mitpress.mit.edu/sicp/