Entwicklung algorithmischer Skelette für CUDA am Beispiel von Affintiy Propagation

Christoph Winter

Fakultät für Informatik und Mathematik
Ostbayerische Technische Hochschule Regensburg
93049 Regensburg
christoph.winter@st.oth-regensburg.de

25. September 2014

Inhaltsverzeichnis

Projektziele

- Portiere Clusteralgorithmus **Affinity Propagation** auf **GPUs**
- Programmierung auf hohem Abstraktionsniveau
  - Compute Unified Device Architecture (CUDA)
  - Separation of Concerns: Kenntnis von Hardwarespezifika (**wie**) von Wissen um Algorithmus (**was**) trennen.
  - Thrust: A Productivity-oriented Library for CUDA[1]
    - Nachbau der C++ Standard Template Library (STL)
    - Funktionen höherer Ordnung (Higher-Order Function, HOF)[2]
    - werden mit anderen Funktionen/Funktionsobjekten parametrisiert
    - z. B.: Map-Funktion: *thrust::transform(first,last,result, UnaryFunction)*

[1] Nathan Bell and Jared Hoberock
[2] Murray I. Cole

Nachteile Thrust

Für Affinity Propagation werden v. a. **zeilen- und spaltenweise** Transformationen und Reduktionen von **Matrizen** benötigt.

- Thrust auf Verarbeitung von Vektoren ausgelegt
- keine direkte Unterstützung von Matrizen
  - lediglich thrust::reduce_by_key()
    - **zusätzlicher Speicherverbrauch** um *keys* zu speichern
    - Nichtnutzung impliziter Speicherstruktur Matrix $\Rightarrow$ **langsam**
- verschiedene Backends (CUDA, OpenMP, TBB)
  - **komplexe** Codebasis
  - **schwer erweiterbar**

$\Rightarrow$ eigene HOFs als Erweiterung zu Thrust $\Rightarrow$ **Pulse**

Pulse: Vorteile im Gegensatz zu monolithischen Kerneln

```
pulse::reduce_column( A.begin(), A.end(), R.begin(), 100, plus<float>());
```

- Datenparallele HOFs der Pulse Bibliothek sind
  - ... **einfach** zu benutzen
  - ... **sehr flexibel** durch Iteratoren und Funktionsobjekte

Pulse: Vorteile im Gegensatz zu monolithischen Kerneln

```
pulse::reduce_column(  (double*)&A[0], (double*)&A[100*100],R.begin(), 100, minus<float>())
```

- Datenparallele HOFs der Pulse Bibliothek sind
  - ... **einfach** zu benutzen
  - ... **sehr flexibel** durch Iteratoren und Funktionsobjekte

Pulse: Vorteile im Gegensatz zu monolithischen Kerneln

```
pulse::reduce_column(
   thrust::make_zip_iterator(make_tuple( A.begin(), B.begin() )
   thrust::make_zip_iterator(make_tuple( A.end(), B.end() )
   R.begin(), 100, minus<float>());
```

- Datenparallele HOFs der Pulse Bibliothek sind
  - ... **einfach** zu benutzen
  - ... **sehr flexibel** durch Iteratoren und Funktionsobjekte
  - ... **unabhängig von Datentypen** durch Templates

Pulse: Vorteile im Gegensatz zu monolithischen Kerneln

```
pulse::reduce_column(
   thrust::make_zip_iterator(make_tuple( A.begin(), B.begin() )
   thrust::make_zip_iterator(make_tuple( A.end(), B.end() )
   R.begin(), 100, minus<float, int>());
```

- Datenparallele HOFs der Pulse Bibliothek sind
  - ... **einfach** zu benutzen
  - ... **sehr flexibel** durch Iteratoren und Funktionsobjekte
  - ... **unabhängig von Datentypen** durch Templates
  - ... somit **wiederverwendbar**: Abstraktion von Berechnung und Mapping auf Hardware

Pulse: Vorteile im Gegensatz zu monolithischen Kerneln

```
pulse::reduce_column(
   thrust::make_zip_iterator(make_tuple( A.begin(), B.begin() )
   thrust::make_zip_iterator(make_tuple( A.end(), B.end() )
   R.begin(), 100, minus<float, int>());
```

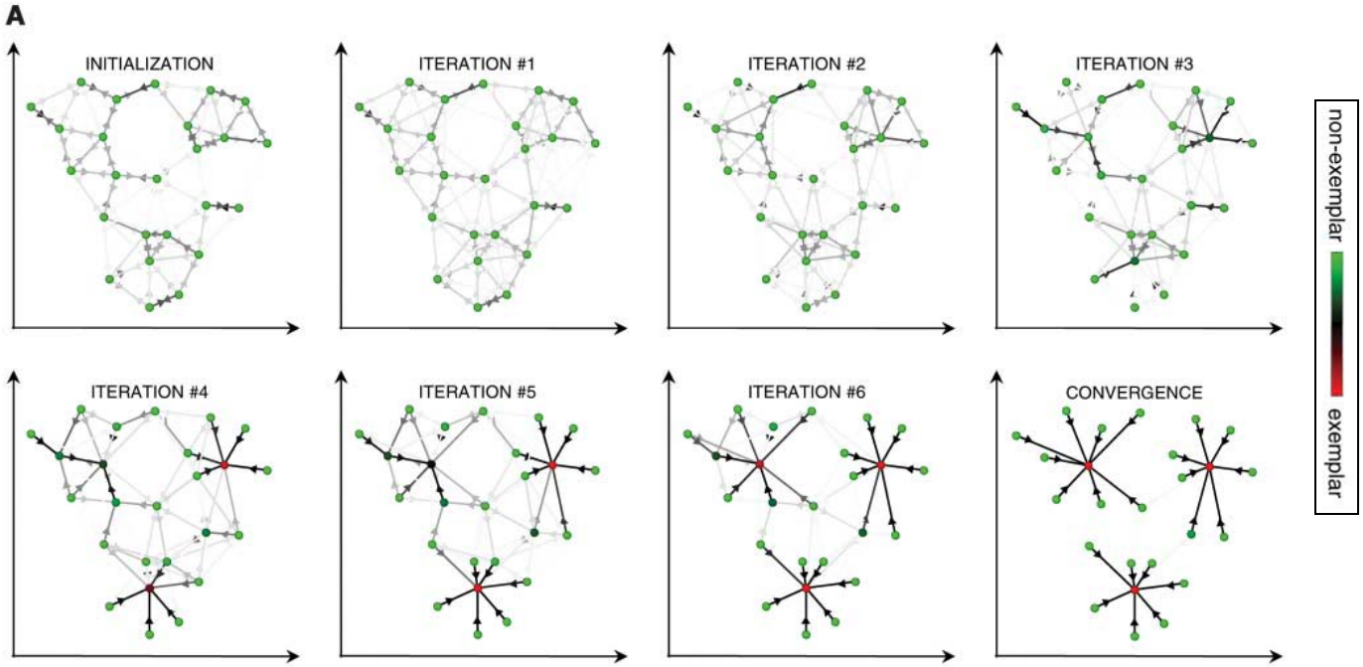
- Datenparallele HOFs der Pulse Bibliothek sind
  - ... **einfach** zu benutzen
  - ... **sehr flexibel** durch Iteratoren und Funktionsobjekte
  - ... **unabhängig von Datentypen** durch Templates
  - ... somit **wiederverwendbar**: Abstraktion von Berechnung und Mapping auf Hardware
  - *... selbst als Templates zu betrachten, die bei Entwicklung neuer Kernel dabei helfen analytisch Vorzugehen, da Reduktion auf Nötigste (im Gegensatz zu Thrust)*

Pulse: Vorteile im Gegensatz zu monolithischen Kerneln

```
pulse::reduce_column(
  thrust::make_zip_iterator(make_tuple( A.begin(), B.begin() )
  thrust::make_zip_iterator(make_tuple( A.end(), B.end() )
  R.begin(), 100, minus<float, int>());
```

- Datenparallele HOFs der Pulse Bibliothek sind
  - ... **einfach** zu benutzen
  - ... **sehr flexibel** durch Iteratoren und Funktionsobjekte
  - ... **unabhängig von Datentypen** durch Templates
  - ... somit **wiederverwendbar**: Abstraktion von Berechnung und Mapping auf Hardware
  - ... *selbst als Templates zu betrachten, die bei Entwicklung neuer Kernel dabei helfen analytisch Vorzugehen, da Reduktion auf Nötigste (im Gegensatz zu Thrust)*

Affinity Propagation[3]: Grundidee



- $n \times n$ Distanzmatrix (sog. **S**imilarities)
- Iterativ: Nachrichtenaustausch zwischen Datenpunkten
- Kumuliere Nachrichten in zwei $n \times n$ Matrizen
  - **R**esponsibilities $r(i,k) \in \mathbf{R}$: Verlangen von Datenpunkt i, Cluster anzugehören, dessen Exemplar Punkt k ist
  - **A**vailabilities $a(i,k) \in \mathbf{A}$: Bereitschaft des Punktes k, als Exemplar für i zu dienen.

[3] Clustering by Passing Messages Between Data Points, Science 315, 972 (2007); DOI: 10.1126/science.1136800

Affinity Propagation: Beispiel

Zeilenweise Reduktion: Suche größte und zweitgrößte Summe aus a^{t-1} und s

```
pulse::transform_reduce_row(
    thrust::make_zip_iterator( make_tuple( similarities.begin(), availabilities.begin() )),
    thrust::make_zip_iterator( make_tuple( similarities.end(), availabilities.end() )),
    thrust::make_zip_iterator( make_tuple( mx1.begin(), mx2.begin() )),
    columns,
    convert_to_tuple< thrust::tuple<value_type,value_type> >(),
    reduce_tuple< thrust::tuple<value_type,value_type> >() );
```

```
template<class T>
struct convert_to_tuple : public std::unary_function<T,T> {
  T operator()(const T& value) const {
    //!< calculate sum a^{t-1} + s and return tuple (a^{t-1} + s, -∞)
    return thrust::make_tuple( thrust::get<0>(value)+thrust::get<1>(value),-MAX_VALUE );
  }
};
```

Affinity Propagation: Beispiel

Zeilenweise Reduktion: Suche größte und zweitgrößte Summe aus a^{t-1} und s

```
pulse::transform_reduce_row(
    thrust::make_zip_iterator( make_tuple( similarities.begin(), availabilities.begin() )),
    thrust::make_zip_iterator( make_tuple( similarities.end(), availabilities.end() )),
    thrust::make_zip_iterator( make_tuple( mx1.begin(), mx2.begin() )),
    columns,
    convert_to_tuple< thrust::tuple<value_type,value_type> >(),
    reduce_tuple< thrust::tuple<value_type,value_type> >() );
```

```
template<class T>
struct convert_to_tuple : public std::unary_function<T,T> {
  T operator()(const T& value) const {
    //!< calculate sum a^{t-1} + s and return tuple (a^{t-1} + s, -∞)
    return thrust::make_tuple( thrust::get<0>(value)+thrust::get<1>(value),-MAX_VALUE );
  }
};
```

```
template<class T>
struct reduce_tuple : public std::binary_function< T, T, T > {
  T operator()(const T& a, const T& b) const {
    typedef typename thrust::tuple_element<0,T>::type value_type;
    //!< invariant: first element of tuple always contains greatest value
    value_type max1 = thrust::max( thrust::get<0>(a), thrust::get<0>(b) );

    //!< lookup second greatest value...
    value_type max2 = thrust::max(
     thrust::min( thrust::get<0>(a), thrust::get<0>(b)),
     thrust::max( thrust::get<1>(a), thrust::get<1>(b)));

    return thrust::make_tuple(max1,max2);
  }
};
```

Affinity Propagation: Beispiel

Zeilenweise Reduktion: Suche größte und zweitgrößte Summe aus a^{t-1} und s

```
pulse::transform_reduce_row(
    thrust::make_zip_iterator( make_tuple( similarities.begin(), availabilities.begin() )),
    thrust::make_zip_iterator( make_tuple( similarities.end(), availabilities.end() )),
    thrust::make_zip_iterator( make_tuple( mx1.begin(), mx2.begin() )),
    columns,
    convert_to_tuple< thrust::tuple<value_type,value_type> >(),
    reduce_tuple< thrust::tuple<value_type,value_type> >() );
```

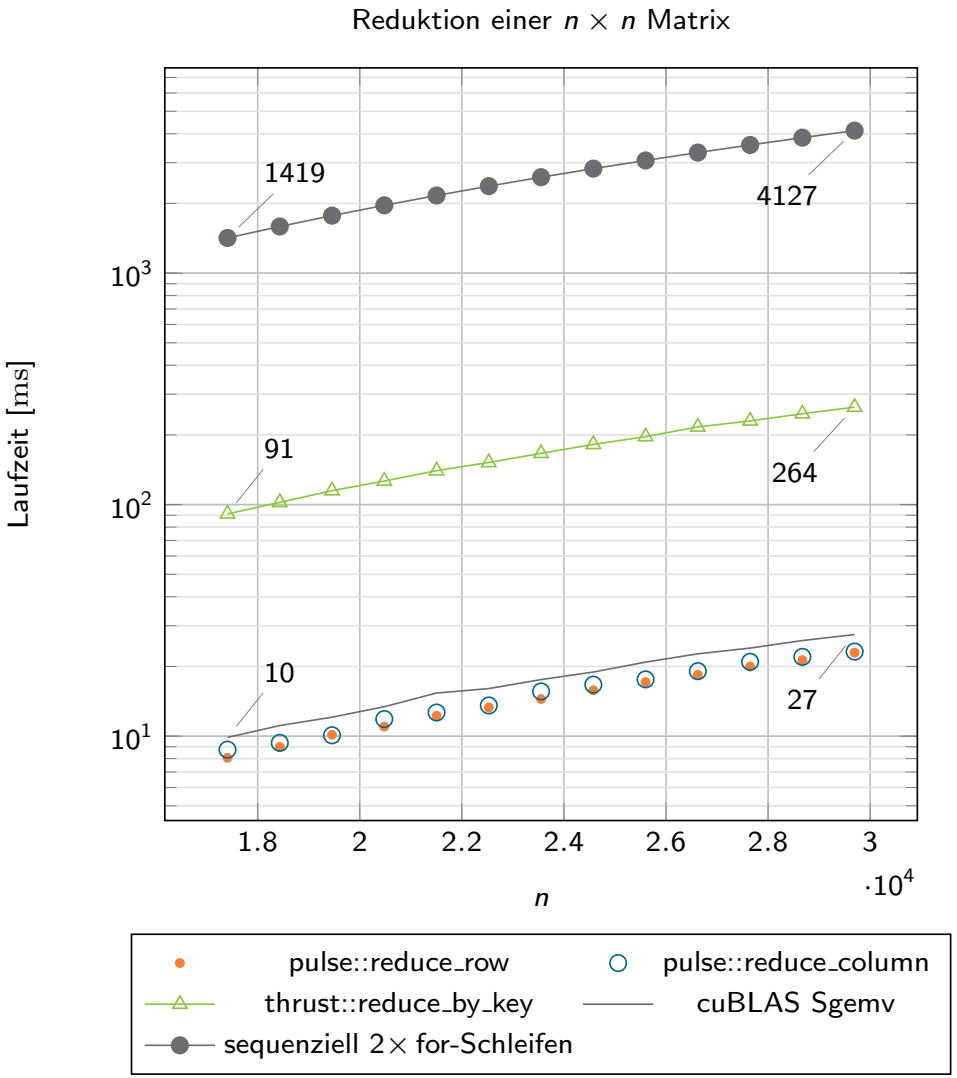
```
template<class T>
struct convert_to_tuple : public std::unary_function<T,T> {
  T operator()(const T& value) const {
    //!< calculate sum a^{t-1} + s and return tuple (a^{t-1} + s, -∞)
    return thrust::make_tuple( thrust::get<0>(value)+thrust::get<1>(value),-MAX_VALUE );
  }
};
```

```
template<class T>
struct reduce_tuple : public std::binary_function< T, T, T > {
  T operator()(const T& a, const T& b) const {
    typedef typename thrust::tuple_element<0,T>::type value_type;
    //!< invariant: first element of tuple always contains greatest value
    value_type max1 = thrust::max( thrust::get<0>(a), thrust::get<0>(b) );

    //!< lookup second greatest value...
    value_type max2 = thrust::max(
     thrust::min( thrust::get<0>(a), thrust::get<0>(b)),
     thrust::max( thrust::get<1>(a), thrust::get<1>(b)));

    return thrust::make_tuple(max1,max2);
  }
};
```

Higher-order Functions im Vergleich



Affinity Propagation: 1× Tesla K20 GPU vs. seq. CPU

Problemgröße	Sequenzielle Ausführung [s]	GPU[4] Laufzeit [s]	Speedup
	Doppelte Genauigkeit		
1.000 × 1.000	21.8	0.8	27.3
2.000 × 2.000	88.5	2.3	38.3
4.000 × 4.000	342.0	8.7	39.3
6.000 × 6.000	759.4	19.1	39.8
8.000 × 8.000	1335.3	33.6	39.7
10.000 × 10.000	2050.0	51.6	39.7
12.000 × 12.000	2972.3	74.4	40
	Einfache Genauigkeit		
1.000 × 1.000	21	0.7	30
2.000 × 2.000	84.3	1.8	46.8
4.000 × 4.000	324.8	6.2	52.4
6.000 × 6.000	721	14.1	51.1
8.000 × 8.000	1273.8	23.8	53.5
10.000 × 10.000	1982	38.9	51
12.000 × 12.000	2857.8	55.2	51.8

Tabelle: Mittlere Laufzeiten von 1000 Iterationen der GPU Variante (ECC aktiviert) im Vergleich zu sequenziellem AP.

[4] CUDA 6.0, gcc 4.7.2, GPU:Tesla K20, nvcc 6.0.1, Driver 331.62