

Pflichtenheft  
Helmholtz-Zentrum Dresden-Rossendorf

## Digital Signal Processing

---

**Eine Cuda Implementierung des Levenberg-Marquardt  
Algorithmus zu Auswertung von Messdaten für die Positron  
Annihilations Spektroskopie.**

---

Fabian Jung  
Mat.Nr.: 3755341  
Diplom Informatik

Nico Wehmeyer  
Mat.Nr.: 3658043  
Diplom Informatik

Richard Pfeifer  
Mat.Nr.: 2922220  
Promotion Physik

21. Januar 2014

## Inhaltsverzeichnis

<b>1</b>	<b>Optimierungsalgorithmus</b>	<b>3</b>
<b>2</b>	<b>Speichertransfer zwischen Host und Device</b>	<b>4</b>
2.1	Grobgliederung . . . . .	4
2.2	Aufgaben . . . . .	4
<b>3</b>	<b>Host</b>	<b>6</b>
3.1	Anforderung . . . . .	6
3.2	Hauptbestandteile . . . . .	6
3.2.1	Ringpuffer-Klasse . . . . .	6
3.2.2	Erzeuger-Klasse . . . . .	7
3.2.3	Verbraucher . . . . .	7
3.3	Tasks . . . . .	7

## 1 Optimierungsalgorithmus

- Aufgabe

Der Algorithmus soll die gemessenen Daten näherungsweise als Funktion interpretieren. (Die Daten sind gleichmäßig über die Zeit verteilt und repräsentieren die Funktionswerte.)

Anschließend sollen der Anfangs-, End- und Maximalwert der Funktion bestimmt werden.

- Implementierungsschritte

- Levenberg-Marquardt-Algorithmus
- double zu short int konvertieren (betrifft Messdaten)
- kernel-Methode (statt main)
- Speicher der Grafikkarte nutzen (statt malloc)
- Cuda Array nutzen (statt Array)
- Berechnung von Anfangs- und Endfunktionswerten (ggf. mitteln)
- Rückgabe der Ergebnisdaten
- ggf. kernel zu sub-kernel parallelisieren
- ggf. andere Fit-Funktion (z. B.  $e^{-x^2}$  mit entsprechenden Parametern) verwenden oder Daten vor Bestimmung der quadratischen Funktion trimmen

## 2 Speichertransfer zwischen Host und Device

### 2.1 Grobgliederung

Die Eingangsdaten werden aus einem Ringpuffer gelesen. Die Ergebnisse werden zunächst in einem Ringpuffer zwischengespeichert und dann in eine Datei geschrieben. Jeder Node erhält seinen eigenen Thread, der die Daten aus dem Eingangspuffer auf die Grafikkarte kopieren. Das Schreiben in die Datei wird ebenfalls von einem eigenen Thread erledigt. Damit es nicht zu Lese- oder Schreibkonflikten kommt, werden die Puffer durch Semaphoren geschützt. Um die einzelnen Nodes zu verwalten wird eine Klasse implementiert, deren Objekte jeweils einen der 4 Nodes der GPU verwalten.

### 2.2 Aufgaben

- Initialisierungscode schreiben
  - Auslesen der Systemdetails mit `cudaDeviceProps`
  - Objekte für jeden Node erstellen
  - die einzelnen Threads starten
- Node Klasse
  - Attribute
    - \* Device Pointer
    - \* Referenz auf Ringpuffer für Eingang
    - \* Referenz auf Ringpuffer für Ausgang
    - \* finish
      - Variable die vom Host gesetzt werden kann um anzuzeigen, dass keine neuen Daten mehr folgen
  - Konstruktor/Destruktor
  - Methoden
    - \* `stop` (public)
      - Host beendet den Thread evt. auch im Destruktor implementiert
    - \* `run` (public)
      - Wird als eigener Thread gestartet
    - \* `copyToDevice` (private)
    - \* `getResults` (private)
- Speichertransfer Host -> Device
  - In Methode `copyToDevice`

- Textur Referenz anlegen (im File scope)
  - Texturobjekt initialisieren
  - Daten in Textur kopieren
  - Anschließend Kernel starten
- Speichertransfer Device -> Host
  - Ende des Kernels abwarten
  - cudaMemcpy der Daten auf den Stack
  - Einfügen in Ausgangspuffer
- Ausgangspuffer
  - Attribute
    - \* ofstream outputFile
    - \* vector finished
      - Zeigt für jeden Node an, ob dieser noch weitere Daten liefert
  - Initialisierung
    - \* Datei öffnen
    - \* finished mit false füllen
  - Destruktor
    - \* Datei schließen
  - Empty
    - \* Wird in eigenen Thread gestartet
    - \* Läuft, solange mindestens einer der Werte aus dem vector finished false
    - \* Schreibt die Ergebnisse unsortiert in Datei
    - \* Sortierte Ausgabe mögliche Erweiterung
  - Write
    - \* Id und Datenstruct in Queue schreiben
  - finished
    - \* setzt ein Bit für einen Node in finished auf true

## 3 Host

### 3.1 Anforderung

- Auf externem Speicher vorliegende Daten sollen in einen Puffer geladen werden. Zwecks Weiterverarbeitung muss eine Schnittstelle bereitgestellt werden, um diese Daten effektiv aus dem Puffer in den Grafikkartenspeicher zu transportieren.
- Das Auslesen aus dem Puffer erfolgt durch mehrere Threads. Alle Operationen auf dem Puffer müssen demnach threadsafe gestaltet sein.

### 3.2 Hauptbestandteile

Die Anforderungen sind typisch für ein Erzeuger-Verbraucher-Problem. Dieses wird durch folgende Komponenten gelöst:

#### 3.2.1 Ringpuffer-Klasse

Der Ringpuffer wird durch Erzeuger und Verbraucher gefüllt/geleert. Die Puffergröße ist fix. Threadsicherheit wird mittels Semaphoren in die Ringpufferklasse implementiert. Da momentan POSIX-Plattformen als Ziel genannt sind, wird `semaphore.h` dafür genutzt.

```
1 template <class Type>
2 class Ringbuffer() {
3 private:
4     sem_t mtx;
5     sem_t full, empty;
6 public:
7     Ringbuffer(unsigned int bSize);
8     ~Ringbuffer();
9     int writeFromHost(Type* inputOnHost);
10    int copyToHost(Type* outputOnHost);
11    Type* reserveHead();
12    int freeHead();
13    Type* reserveTail();
14    int freeTail();
15    bool doQuit();
16 }
```

- `int writeFromHost(Type* inputOnHost)` ermöglicht das Schreiben der Daten `inputOnHost` in den Puffer. Dazu müssen die Daten im Speicher des Host liegen. Der Rückgabewert ist 0 wenn das Schreiben erfolgreich war, sonst eine positive Zahl. Bei vollem Puffer blockiert der Aufruf bis wieder Platz verfügbar ist.
- `int copyToHost(Type* outputOnHost)` liest Daten aus dem Puffer nach `outputOnHost`. Bei leerem Puffer blockiert der Aufruf bis Daten im Puffer verfügbar sind. Der Zeiger `outputOnHost` muss auf eine Hostadresse zeigen.

- Um Daten von der Grafikkarte in den Puffer zu schreiben, muss der Kopiervorgang vom Aufrufer selbst durchgeführt werden. Dazu kann vom Puffer mit `Type* reserveHead()` eine verfügbare Speicheradresse angefordert werden. Sollte kein Platz im Puffer sein, blockiert der Aufruf bis dies der Fall ist. Bis zum Aufruf von `int freeHead()` ist der Puffer für andere Aktivitäten blockiert.
- `Type* reserveTail()` ermöglicht Daten aus dem Puffer auf die Grafikkarte zu laden. Die Rückgabeadresse ist die reservierte Adresse im Puffer aus der die Daten vom Aufrufer selbst kopiert werden können. Die Adresse und der Puffer müssen mit `int freeTail()` wieder freigegeben werden.
- `freeHead()` und `freeTail()` geben im Erfolgsfall 0 zurück. Bei einem Fehler ist der Rückgabewert positiv.
- Ist der Puffer leer und hat der Erzeuger gemeldet, dass keine Daten mehr geliefert werden, so gibt der Puffer auf `bool doQuit()` `true` zurück. Andernfalls `false`.

### 3.2.2 Erzeuger-Klasse

Verantwortlich für das Einlesen der Daten aus einer Verzeichnisstruktur in den Puffer. Die Waveforms aus den beiden Kanälen werden aufgetrennt in zwei Waveforms. Dazu wird ein kleiner interner Puffer verwendet. Der Erzeuger versucht zu schreiben, solange er Daten liefern kann. Der tatsächliche Schreibvorgang wird über das Semaphore `sem_w` innerhalb der Ringpufferklasse gesteuert.

Die sample-Daten sind laut Code vom Typ `short int`. Folgende Typen werden verwendet

```
typedef short int sample_t           // Samplingpunkte

typedef std::vector<sample_t> wform_t
wform_t.reserve(SAMPLE_COUNT)       // eine waveform

typedef std::vector<wform_t> chunk_t
chunk_t.reserve(CHUNK_COUNT)        // Ein Chunk waveforms
```

### 3.2.3 Verbraucher

Der Verbraucher wird im Teil “Speichertransfer” beschrieben. Das Semaphore `sem_r` regelt den Lesezugriff innerhalb der Ringpufferklasse.

Der Verbraucher stellt über entsprechendes mapping sicher, dass die Datenreihenfolge bekannt bleibt, um die Ergebnisse den Eingangsdaten zuordnen zu können.

## 3.3 Tasks

- Erzeugerklassen: Lesen der Messdaten mit root-Verzeichnis der Daten als Eingabewert

- erwartete Ordnung der Messdaten nach Einlesen im Puffer beschreiben
- Ringpuffer initialisieren
  - Speicherreservierung
  - Semaphoren: `sem_mtx`, `sem_w`, `sem_r`
- Ringpuffer: `write`
- Ringpuffer: `readToHost`
- Ringpuffer: `readToCUDA`
- Vergleich der Ergebnisse CPU vs GPU-Auswertung
- Skalierung
- Template-version des Ringpuffers