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	Nico Wehmeyer	Richard Pfeifer
Mat.Nr.: 3755341	Mat.Nr.: 3658043	Mat.Nr.: 2922220
Diplom Informatik	Diplom Informatik	Promotion Physik

14. Januar 2014

Inhaltsverzeichnis

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
  - cudaMemcopy 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 Graphikkartenspeicher 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(size_t bSize);
8      ~Ringbuffer();
9      type* reserveHead(unsigned int count);
10     int freeHead(type* data, unsigned int count);
11     type* reserveTail(unsigned int count);
12     int freeTail(type* gpu_data, unsigned int count);
13 }
```

- Der Zustand voll / leer wird über die Semaphore `sem_w` (Anfangszustand `bSize`), `sem_r` (Anfangszustand `0`) festgestellt.
- Die Funktion `write` blockiert bei vollem Puffer um ein Verlust an Messdaten entgegenzuwirken. Für späteres Lesen eines Datenstroms direkt vom Messgerät kann auch nichtblockierendes Schreiben in den Puffer mit entsprechendem Überlauf implementiert werden.
- `readToCUDA` kopiert die Daten vom Tail des Puffers nach `*gpu_data`. Die Verantwortlichkeit für `cudaSetDevice(deviceId)` ist noch zu klären. Der Kopiervorgang kann nicht asynchron ablaufen, um den Speicher im Puffer sicher freigeben zu können.
- `readToHost` kopiert die Daten vom Tail des Puffers an eine Hostspeicheradresse. Dies dient zum Testen der CPU-Variante des Auswertealgorithmus.

- `size` in `write`, `readToHost` und `readToCUDA` gibt jeweils die Anzahl der auszutauschenden Waveforms vom Datentyp `wform` an.

3 Semaphoren:

- `sem_mtx` für exklusiven Zugriff auf Speicher
- `sem_r` um Lesezugriff bei leerem Puffer zu regeln
- `sem_w` um Schreibzugriff bei vollem Puffer zu regeln

Der Datentyp `wform` ist zunächst fix. In einer zweiten Version wird er per Template implementiert.

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`. Damit ergibt sich für `wform`

```
typedef short int wform[nSample]
```

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

- Erzeugerklasse: 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