1 taumain.py

Die Klasse "dataThread" dient zum Auslesen der Daten aus dem tauhost-Programm. Weiterhin benutzen wir die "animThread"-Klasse, um die Werte, die wir aus "dataThread" bekommen, zu plotten. Zur Datenverarbeitung haben wir einen Eventhandler zwischen "animThread" und "dataThread" benutzt. Mit den presets übergeben wir der tauhost-Datei, die für die verschiedenen Potentiale spezifischen Werte. Dann kommen die benutzten Variablen:

- n: Anzahl der Punkte
- deltat: Zeitunterschied zwischen zwei Punkten
- deltatau: Entwicklung der Liste über die fiktive Zeit tau
- h: Parisi-h
- parisi: Wird der Parisi-Trick benutzt oder nicht
- entw: Anzahl der Entwicklungsschritte
- c: eine momentan nicht benutzte Konstante
- device: ID des Geräts, das benutzt werden soll
- rpf: Mit rpf=1 werden in jedem Loop die Daten übergeben
- intime: Wie viele Loops laufen, bevor in fsum und fhsum geschrieben wird
- loops: Loops pro Entwicklungsschritt
- inputf: Möglichkeit der Dateieingabe
- outputf: In welche Datei wird geschrieben
- acco: Genauigkeit der geschrieben Daten

Das Unterprogramm mit den zuvor definierten Parametern wird ausgeführt in Zeile 154.

2 tauhost.c

- bis Zeile 41: Parameter werden eingelesen
- Zeilen 47-50: Speicherplatz wird reserviert
- Zeile 56: Wenn deltatau sich ändert, wird deltatau nicht überschrieben, sondern in deltautmp geschrieben
- Zeilen 77-79: es werden mit Hilfe der Box-Müller-Transformation zwei gaußverteilte Zufallszahlen zwischen 0 und 1 erzeugt, damit wird omega zum ersten Mal berechnet

- Zeilen 82-172: Inputverarbeitung, falls Datei als Input genutzt wird
- Zeilen 174-192: Daten werden initialisiert (Seeds für Zufallszahlengeneration werden erstellt)
- Zeilen 205-255: Infos über das benutzte Gerät
- Zeilen 264-319: Speicherbuffer auf dem Gerät wird kreiert
- Zeilen 324-377: Werte werden das erste Mal in den Buffer geschrieben
- Zeile 381: Programm wird kreiert
- Zeile 403: Kernel erstellt
- Zeilen 417-433: Argumente werden an Kernel übergeben
- Zeile 481: Beginn des Hauptloops
- Zeile 483: Kernel ausführen
- Zeile 485: Synchronisierung
- Zeile 487: Beginn zweiter Loop (Wertausgabe)
- Zeile 534: Wert für stable wird ausgelesen und der Kernel meldet, ob die Berechnung stabil verlief oder nicht
- Zeilen 547f: Wert wird fsum und fhsum hinzugefügt
- Zeile 563: Wenn die Berechnung nicht stabil verlief, dann wird deltatau aus dem Kernel ausgelesen, in deltatautmp geschrieben und das neu berechnete deltatau (altes $deltatau \cdot 0.95$) wird wieder an den Kernel zurückgegeben
- Zeile 600f: f-Werte aus dem host-Programm werden zurück in den Kernel geschrieben
- Zeilen 614-630: Möglichkeit in Output-Datei zu schreiben
- Zeilen 636-666: Speicherplatz wird freigegeben

3 tau_kernel.cl

Der eigentliche Kern des Programms liegt hier. Dem Kernel wurden vom Hostprogramm einige Zeiger übergeben. Die wichtigsten hiervon sind `__global double *f`, `__global double *fh`, `__global double *newf`, `__global double *newfh`, `__global double *omega` und `__global ulong *rand1`. Diese sind der Reihenfolge nach: die Liste der Fluktuationen f mit und ohne Parisi-h, Listen für neue Werte dieser Fluktuationen, ein Zeiger auf die kollektive Koordinate ω, sowie eine Liste mit Seeds für die Zufallszahlengeneration. In den Zeilen 41-53 werden einige Konstanten lokal gespeichert und die globale ID `i` des Threads abgefragt.

Dann beginnt der Hauptloop des Kernels, bei dem vorher spezifiziert werden kann, wie oft geloopt werden soll. Gibt es mehrere Work-Groups, können viele Loops aufgrund von nicht-synchronisiertem Ausführen zu Ungenauigkeiten führen. Zunächst wird das Noise `dw` berechnet. Hierzu wird in der Funktion `random` eine Zufallszahl mithilfe des Seeds `rand[i]` berechnet und per Box-Müller-Transformation in eine Gauß-verteilte Zufallszahl umgewandelt. Nun kommt endlich die eigentliche Zeitentwicklung der Fluktuationen f und der kollektiven Koordinate ω. Ist der Thread nicht für ω, f_0 oder $f_{\mathrm{N}-1}$ verantwortlich, geschieht diese in den Zeilen 107-109 durch

```
newf[i] = f[i] + m*deltatau*(f[i+1]+f[i-1]-2*f[i])\\
/(double)pown((float)deltat,2) - ddPot(clas((double)i * deltat, om,\\
potID) + f[i], potID) * f[i] * deltatau + dw;
```

.

Ist der Thread für f_0 oder $f_{\mathrm{N}-1}$ verantwortlich, geschieht die Zeitentwicklung in den Zeilen 62-83 bzw. 83-101. Diese hängt von den Boundaryconditions ab und hier gibt es zwei Implementierungen:

- (Zeilen 63-68 und 84-88) Die äußersten Zeitpunkte befinden sich direkt neben Minus bzw. Plus Unendlich. Dies erscheint physikalisch sinnvoll, da wir hier die Boundaryconditions kennen: Nämlich die Lokalisierung des Teilchens in den Wells.
- (Zeilen 69-73 und 89-93) Eine periodische Boundarycondition wie im Manual. Dies erscheint wenig sinnvoll, gibt aber hübschere Kurven.

Ist der Thread für ω verantwortlich, geschieht die Zeitentwicklung in Zeile 104 durch Multiplikation der Noise mit $\frac{\sqrt{3}}{2}$ im Falle des Double-Well-Potentials. Dies kommt aus Gleichung (4.22b) des Papers 'Stochastic quantization and the tunneling problem'.

In den Zeilen 113-150 passieren Stabilitätschecks, die momentan etwas ausrangiert sind, da sie noch aus dem alten Manual sind und nun entweder falsch implementiert sind oder anders sein müssen(?). Jedenfalls würden sie wenn aktiv das $\Delta\tau$ sofort auf null runterregulieren.

In Zeile 155 wird eine Work-Group synchronisiert. Es sollte wahrscheinlich zur Sicherheit noch einen solchen Call in den Zeilen 145 und 151 geben.