GPU-beschleunigte Packungsoptimierung

André Müller, Johannes J. Schneider, Elmar Schömer

Betrachtetes Packproblem

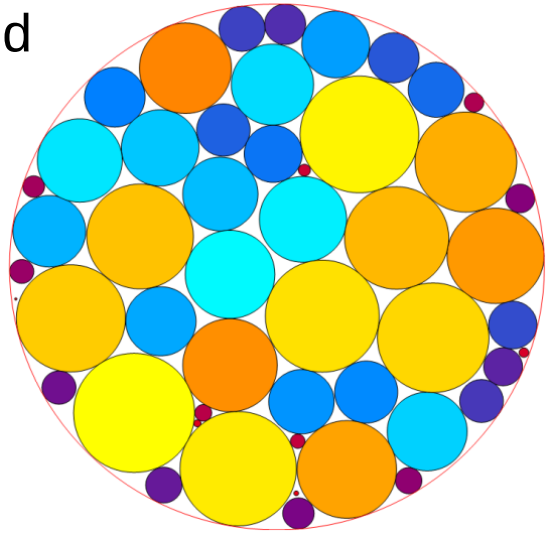
gegeben

- $N = 5, 6, \ldots, 50$ harte Kugeln in Dimension d
- mit ganzzahligen Radien $r_i = 1,2,\ldots,N$

gesucht

- Anordnung deren Umkugelradius R minimal ist

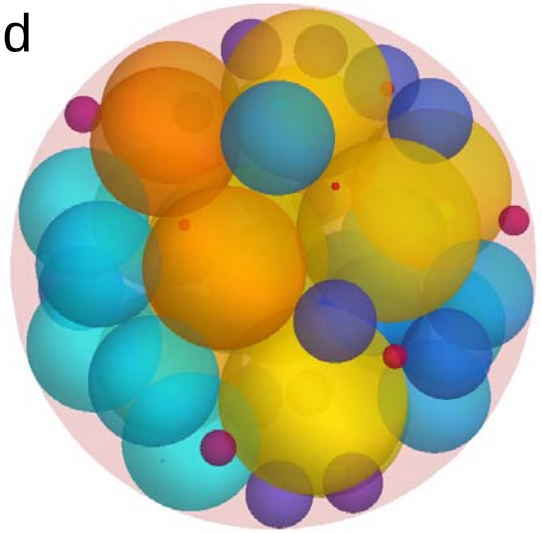
2d



$R = 220.6004187$

$N = 50$

3d



$R = 140.5841$

Optimierung: Simulated Annealing

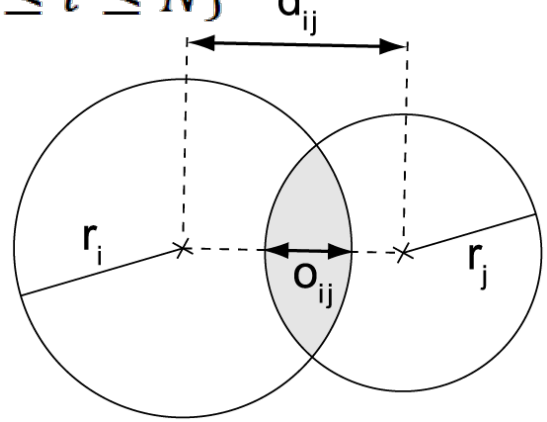
- zufällige Zustandsübergänge (Moves) $\sigma \to \tau$
  - Akzeptanz gemäß Metropolis-Kriterium:

$$p(\sigma \to \tau) = \begin{cases} \exp\left(-\frac{\Delta\mathcal{H}}{k_B T}\right), \Delta\mathcal{H} > 0 \\ 1 \qquad\qquad\quad , \Delta\mathcal{H} \leq 0 \end{cases} \qquad \Delta\mathcal{H} = \mathcal{H}(\tau) - \mathcal{H}(\sigma)$$

  - Starttemperatur oberhalb Peak der spez. Wärme
  - Temperatur schrittweise absenken: $T_{s+1} = T_s \cdot \vartheta$
- Hamiltonian $\mathcal{H} = R + \lambda \cdot \mathcal{P}$
  - Umkugelradius $R = \max\{\|\vec{c}_i\| + r_i \mid 1 \leq i \leq N\}$
  - Straffunktion $\mathcal{P} = \sum_{i<j}^{N} o_{ij}$

$$o_{ij} = (r_i + r_j - d_{ij}) \cdot \Theta(r_i + r_j - d_{ij})$$

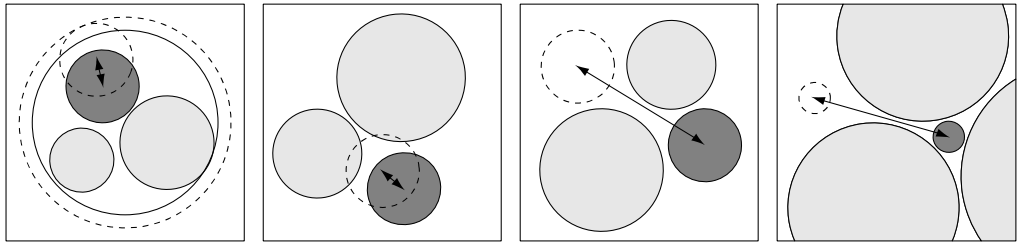
$$d_{ij} = \|\vec{c}_i - \vec{c}_j\| \qquad \Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$



Moves

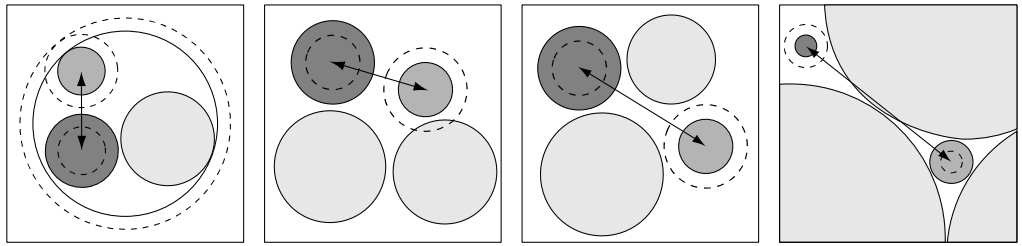
Zustandsübergang $\sigma = (c_1, c_2, \dots, c_N) \rightarrow \tau$

- 1 Kugel Verschieben $\tau = (c_1, c_2, \dots, c_i + \xi, \dots, c_N)$
  - „schieben“ $0 \leq \xi_{shift} \leq 1$
  - „springen“ $0 \leq \xi_{jump} \leq 1000$



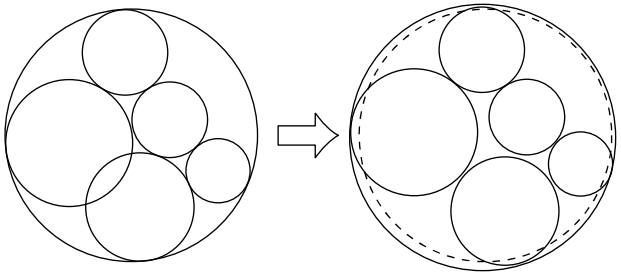
- 2 Kugeln i, j austauschen wobei $|r_i - r_j| = 1$

  $\tau = (c_1, c_2, \dots, c_{i+1}, c_i, \dots, c_N)$



Nachbrenner: Kontaktsimulation

- Überlappungen beseitigen
  - in der Ausgangskonfiguration
  - am Ende der Optimierung



- Vorgehensweise
  - korrigierte Positionen $\vec{c}_i' = \vec{c}_i - \sum_{\substack{i=1 \\ i \neq j}}^{N} \frac{\vec{c}_j - \vec{c}_i}{\|\vec{c}_j - \vec{c}_i\|} \cdot \frac{o_{ij}}{2}$
  - Kompressionsfaktoren

$$f_i = \begin{cases} |\vec{c}_i'| + r_i - R_{old}, & |\vec{c}_i'| + r_i > R_{old} \\ 0, & sonst \end{cases}$$

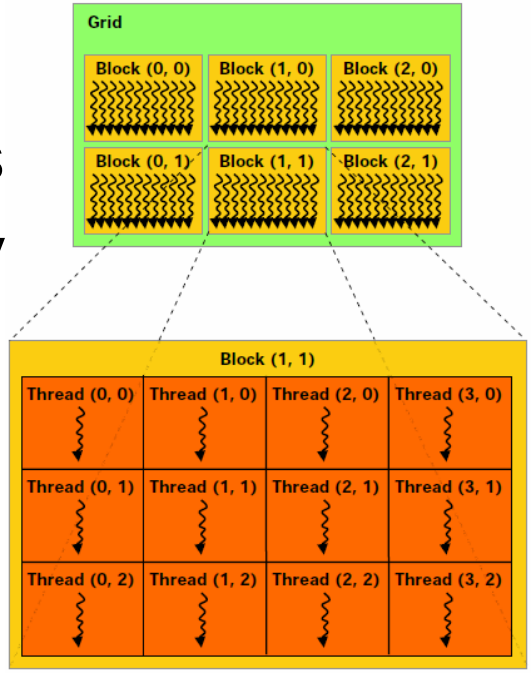
  - setze neue Positionen $\vec{c}_i'' = \vec{c}_i' \cdot \left(1 - \frac{g f_i}{|\vec{c}_i'|}\right)$
    g : Kompressionsgrad
  - wiederhole bis Summe aller Überschneidungen unterhalb vorgegebener Schwelle

CUDA Programming Model

- proprietäres Framework von NVIDIA
- GPU: device code
  - C und Erweiterungen, einige C++ Syntaxregeln
  - unterteilt in Funktionen: ***kernels***
- CPU: host code
  - vollständige Unterstützung von C++
  - Aufruf von Kernelfunktionen mit neuer Syntax kernel<<<configuration>>>(parameters)
- nvcc compiler driver
  - kompiliert Host- und Devicecode
  - benutzt gcc (Linux) oder Microsoft Visual C++ (Windows)

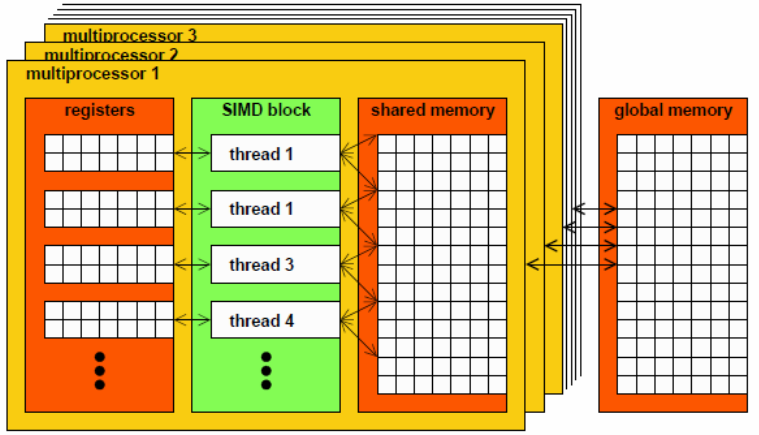
CUDA Computing Model

- Ausführungshierarchie
  - ***Thread*** führt eine Kernelinstanz aus
  - ***Block*** 1, 2 od. 3-dim. Thread-Array
  - ***Grid*** 1 od. 2-dim. Block-Array



- asynchrone Ausführung

- Multiprozessoren
  - max. 8 Blocks parallel
  - max. 24 Warps parallel
  - Warp = 32 Threads



CUDA Memory Model

- ***Global Memory***
  - hohe Latenz (bis zu 600 Zyklen)
- ***Shared Memory*** innerhalb eines Blocks
  - niedrige Latenz (~4 Zyklen), momentan nur 16kB
- lokaler ***Thread Memory*** (Register)
- empfohlener Datenfluss

  host → global → shared → computation (local) → shared → global → host
- Performance-Regeln:
  - Quotient Berechnungen/Speicherzugriff möglichst groß → eher neu Berechnen statt aus dem Speicher lesen
  - Vermeide divergierende Threads, d.h. viele Verzweigungen (if, switch, …)

Implementierungs-Ansatz

- Aufteilung der Berechnungen
  - Moves auf CPU, Hamiltonian auf GPU (getestet)
  - Moves *und* Hamiltonian auf GPU (gewählt)
- Verteilung von p unabhängigen Simulationen
  - Szenario I: **1 Thread pro Simulation** (getestet)
    - 1 Block, $p \cdot N$ Threads
    - hohe Zahl paralleler Prozesse möglich
    - langsam: viele *Global Memory*-Zugriffe nötig
  - Szenario II: **1 Block pro Simulation** (gewählt)
    - 1Thread pro Kugel, N Threads pro Block
    - p Blöcke, $p \cdot N$ Threads
    - schneller: *Shared Memory*-Nutzung innerhalb der Blöcke
    - skaliert besser mit Zahl der Multiprozessoren

Implementation Outline

- **host**: initialize
  - distribute spheres randomly within a given fixed sphere
  - remove overlaps via contact simulation
  - transfer states: host memory → global device memory
  - set $T = T_{init}$
- **host**: while($T > T_{final}$) invoke kernel and wait
  - **device**: for each temperature T do
    - transfer states: global memory → shared memory
    - do a series of moves
    - transfer states: shared memory → global memory
  - **host**: decrease T
- **host**
  - transfer states: global memory → host memory
  - determine best state and write to hard disk
  - eliminate overlaps via contact simulation

Basic Kernels

- linear congruential random number generator
  - use register overflow to emulate modulo operation

```
#define RAND_MOD 1013904223
#define RAND_SEED 1664525
#define RAND_NORM (1./2147483647.)  //1.0/(pow(2.,31)-1)

// returns a random float ranging from 0 to 1
__device__ float _random_num_from_0_to_1(int* seed)
{
    *seed = *seed * RAND_SEED + RAND_MOD;
    return fabs(*seed * RAND_NORM);
}
```

- Metropolis acceptance criterion

```
__device__ int _accept_metropolis(float T, int* seed, float deltaE)
{
    return deltaE <= 0 ||
        (T > 0 && _random_num_from_0_to_1(seed) <= expf(-deltaE/T));
}
```

Optimization Kernel (simplified)

```
__global__ void optimize(float T, int* seed, float3* pos){
    int t = threadIdx.x; int b = blockIdx.x; int p = b*blockDim.x+t;
    //load positions into shared memory
    __shared__ float3 sPos[NUM_SPHERES]; sPos[t] = pos[p]; __syncthreads();
    __shared__ int sSeed; if(t==0) sSeed=*seed+b; __syncthreads();
    //determine circumsphere radius
    float csr = _circumsphere_radius(sPos);
    //perform moves
    for(unsigned long i = 0; i < GPU_MOVES; i++) {
        __shared__ int move; if(t==0) move =_random_int_from_0_to(2,&sSeed);
        __syncthreads();
        switch(move) {
            case 0:_translate_sphere(1.,T,&sSeed,sPos,&csr); break;
            case 1:_translate_sphere(1000.,T,&sSeed,sPos,&csr); break;
            case 2:_swap_2_spheres(T,&sSeed,sPos,&csr); break; }
    }
    //write current state to global memory:
    if(t==0) *seed = sSeed; __syncthreads(); pos[p] = sPos[t];
}
```

Hamiltonian Kernel for R

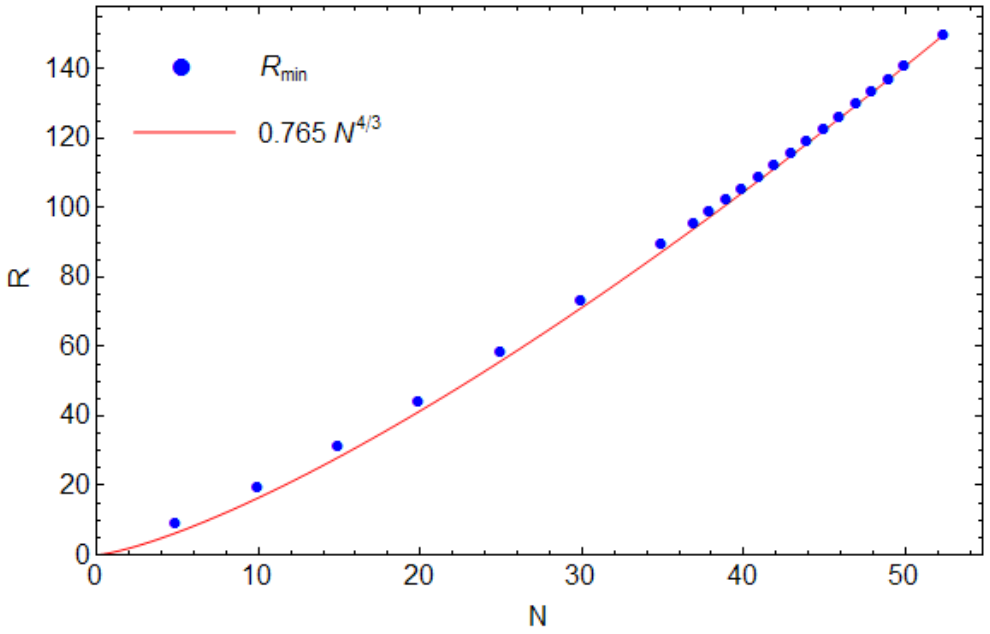
```
__device__ float _circumsphere_radius(float3* pos) {
    int t = threadIdx.x; //thread ID
    float3* p = pos + t; //points to sphere position #t
    float csr = sqrtf(p->x * p->x + p->y * p->y + p->z * p->z) + t+1;
    //store maximum distances of all spheres in one shared array
    __shared__ float dist[NUM_SPHERES];
    dist[t] = csr; __syncthreads();
    //compute maximum using binary reduction scheme
    int o = NUM_SPHERES;
    while(o > 1) {
        o = o/2 + o%2;
        if(t < o && t+o < NUM_SPHERES) {
            csr = fmax(csr,dist[t+o]);
            dist[t] = csr;
        }
        __syncthreads();
    }
    //note: only thread 0 returns the actual circumsphere radius
    return csr;
}
```

Shift/Jump Move Kernel

```
__device__ void _translate_sphere(
        float maxDist, float T, int* seed, float3* pos, float* csr)
{   int t = threadIdx.x; //thread ID
    __shared__ int c; if(t==0) c =_random_int_greq_0_less(NUM_SPHERES,seed);
    __syncthreads();
    //backup old state
    float3 oldPos = pos[c]; float oldPenalty = _penalty_for_sphere(c,pos);
    //translate sphere
    if(t==0) {pos[c].x += _random_num_within(-1,1,seed)*maxDist; . . .}
    __syncthreads();
    //compute energy delta and ckeck acceptance
    float newCsr = _circumsphere_radius(pos);
    float newPenalty = _penalty_for_sphere(c,pos);
    if(t==0) {
       float deltaE = newCsr-*csr + newPenalty-oldPenalty;
       if(_accept_metropolis(T, seed, deltaE))
           *csr = newCsr; else pos[c] = oldPos;
    } __syncthreads();
}
```

Optimierungsergebnisse

- Resultate auf der GPU
  - *R(N)* in 3d



- CPU-Lösung für $N = 50$: $R = 140.5841$
- GPU-Lösungen tendenziell schlechter
  → möglicherweise Präzisionsprobleme

n	$\mathcal{H}_{min}$
5	9.0015
10	19.5361
15	31.1526
20	44.2648
25	58.4827
30	73.3998
35	89.3447
37	95.5049
38	98.9841
39	102.2352
40	105.4217
41	108.9321
42	112.1145
43	115.7380
44	119.2453
45	122.6828
46	126.1462
47	129.8170
48	133.3276
49	137.0085
50	140.7109

Fazit

Vorteile

- Speedup GPU vs. CPU
  - CPU: 1 Kern eines Core2Duo T9300@2.5GHz ($p = 1$)
  - NVIDIA GeForce 8800 GT: bis zu 63x

Nachteile

- Maximale Geschwindigkeit nur, falls
  - $N = 32$ Threads (1 Warp)
  - p das 8-fache der Multiprozessor-Anzahl
- mehr Hindernisse bei der Entwicklung
  - sehr Hardwareorientiert, geringerer Abstraktionsgrad
  - eingeschränkte Erweiterbarkeit
  - kleinste Änderungen im Code können zu drastischen Geschwindigkeitsreduktionen führen

Veröffentlichungen

- **A. Müller, J.J. Schneider, E. Schömer**
  Packing a mulidisperse system of hard disks in a circular environment
  Physical Review E, Vol. 79, Issue 2, 021102 (2009)

- **J.J. Schneider, A. Müller, E. Schömer**
  Ultrametricity property of energy landscapes of multidisperse packing problems
  accepted by Physical Review E

- **www.uni-mainz.de/~schneidj**

- **www.uni-mainz.de/~schoemer**