TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

TU Wien

Mathematische Modellbildung und Simulation
Institut für Analysis und Scientific Computing

BRODER Jürgen, 0425784
ECKER Valentin, 0426030

Simulationsprotokoll Parallelagenten

Im Rahmen der LVA
Simulation

BREITENECKER Felix, SCHNECKENREITHER Günter

Wien, 9. März 2010

Sich bewegende und kommunizierende Agenten

Aufgabenbeschreibung:

Implementieren Sie ein Multi-Agenten-System. Entscheiden Sie sich mit Hilfe der Tutoren für eine Pragrammiersprache und eine Parallelisierungstechnologie.

Für diese Aufgabe wäre es ideal, wenn Sie Zugang zu notwendiger Hardware (Cuda bzw. Mehrkernrechner) hätten. Ein Mehrkernrechner mit einem PVM-Setup und Nvidia-Grafikkarte (muss noch geklärt werden) könnte eventuell bereitgestellt werden.

Modellbeschreibung:

Auf einer zweidimensionalen Ebene befinden sich zwei Klassen von Agenten. Die erste Klasse bewegt sich nicht und wird vor Simulationsbeginn als Anfangsbedingung auf dem Gebiet positioniert. Jeder dieser Agenten beinhaltet einen Vektor, der eine Bewegungsrichtung und -geschwindigkeit definiert. Dieser Vektor kann während der Simulation von in der Nähe befindlichen Agenten der zweiten Klasse gelesen werden und dient als "Wegweiser" für deren Bewegung im nächsten Zeitschritt.

Jeder bewegliche Agent hat eine Anfangsgeschwindigkeit, eine Anfangsrichtungrichtung und ein Sichtfeld mit Radius r. Tritt ein Agent erster Klasse in das Sichtfeld eines beweglichen Agenten, ändert dieser seinen Bewegungsvektor für den nächsten Zeitschritt.
Sei v_t der Bewegungsvektor zum Zeitpunkt t und liegt ein Wegweiser mit Vektor w im Sichtfeld, also innerhalb des Radius r, dann ändert der Agent seinen Vektor auf v_t+1=(v_t+w)/2. Liegen zwei Wegweiser im Sichtfeld, dann gilt v_t+1=(v_t+w_1+w_2)/3 usw.

Jeder Wegweiser beinhaltet also zwei Koordianten für seine Position und zwei für seinen Richtungsvektor. Ebenso die Agenten der zweiten Klasse.

Der Sinn dieses Modells ist es, durch eine bestimmte Positionierung der statischen Agenten eine Strömungsdynamik für die beweglichen Agenten zu erzeugen.
Eine typische Anfangsbedingung wäre z.B eine gleichmässige zufällige Verteilung aller Agenten auf dem Einheitskreis um den Punkt (0,0). Den Agenten der ersten Klasse (mit Position (x,y)) wird dann der Richtungsvektor D(pi/2)*(x-0,y-0) zugewiesen. Wobei D(phi) die Drehmatrix um den Winkel phi ist. Die Agenten der zweiten Klasse erhalten einen zufälligen initialen Bewegungsvektor, dessen Länge z.B. zwischen 0 und 0.5 liegt. Dadurch sollten sich die beweglichen Agenten annähernd im Kreis um den Mittelpunkt (0,0) bewegen.
Werden während der Simulation die Richtungsvektoren der Wegweiser verändert, sollte sich das Strömungsverhalten an die neuen Bedingungen anpassen.

Parallelisierungsstrategie:

Da der Kommunkationsaufwand bei der Parallelisierung relativ groß ist (die Agenten müssen ständig Abstandsberechnungen durchführen), kann es sein, dass sich ein Zeitgewinn erst bei einer sehr großen Anzahl an Agenten einstellt.

Statten Sie die Agenten der zweiten Klasse jedoch mit der Fähigkeit aus, vorausdenkend zu berechnen, können Sie zeitaufwändige Synchronisationen (mit den Agenten der ersten Klasse) vermeiden. Merkt sich ein Agent z.B. die Koordinaten und Richtungsvektoren aller Wegweiser im Umkreis von 3*r Längeneinheiten, so muss eine Synchronisation nur noch jeden zweiten Zeitschritt erfolgen.
Dabei müssen Sie einerseits auf den verfügbaren Speicherplatz Rücksicht nehmen und gleichzeitig darf die Bewegungsstrecke nicht größer als 2*r sein, da sonst notwendige Wegweiser im Zwischenspeicher des Agenten fehlen.

Die Parallelisierungsstrategie hängt aber stark von der verwendeten Methode und den geometrischen Bedingungen (z.B. Dichte) ab!

Aufgabenstellung:

Wählen Sie ein Kombination aus Programmiersprache und Parallelisierungsstrategie!
Zur Kontrolle sollten Sie eine sehr einfache grafische Ausgabe erzeugen, oder für jeden Zeitschritt (zur späteren Visualisierung) eine nummerierte Textdatei mit den Koordinaten der beweglichen Agenten ausgeben.
Vergessen Sie nicht auf eine Beschreibung Ihres Codes und eine Dokumentation.

Programmiersprache	Parallelisierungsmethode
MATLAB + Schnittstelle (www.mb.hs-wismar.de/cea/dp/)	PVM (www.csm.ornl.gov/pvm)
MATLAB + Schnittstelle (www.nvidia.com/...)	Cuda (www.nvidia.com/...)
C/C++	PVM (www.csm.ornl.gov/pvm)
C++	Cuda (www.nvidia.com/...)
C++	Threads (z.B. pthreads)
Java	JPVM (www.cs.virginia.edu/~ajf2j/)

Inhaltsverzeichnis

1 Implementierung

1.1 Grundlengende Implementierung

Programmiersprachen: C und OpenCL
Plattform: Darwin

Wie bei Implementierungen mittels OpenCL üblich gliedert sich die Ausführung in einen Host- und einen Target-Teil. Der Host-Teil wird wie gewohnt auf der CPU kompiliert und dann ausgeführt. Der Target-Code, welcher in OpenCL geschrieben ist, wird zur Laufzeit kompiliert, und dann auf das OpenCL-Device geladen und ausgeführt. OpenCL-Devices können sowohl CPUs als auch GPUs sein. In unserem Fall beschränken wir uns auf die parallele Ausführung auf der GPU. Es steht uns eine Nvidia 9400M GPU zur Verfügung, welche 16 Threads parallel ausführen kann. Die Implementierung unterstützt sowohl multithreading als auch multiple Devices (mehrere CPU-kerne oder mehre GPUs).
Der Host-Teil hat folgende Aufgaben:

- Initialisierung der Agenten
- Kompilieren des OpenCL Codes
- Allocation des OpenCL Devices
- Kopieren der fixen Agenten in das Device Memory
- Allocation von Target-Memory für bewegliche Agenten
- Anstarten der OpenCL Execution
- kopieren der beweglichen Agenten in das Host-Memory nach jeder Execution
- generieren des Outputs in Form eines Bildes im ppm-format

Der Target-Teil hat folgende Aufgaben:

- Berechnung ob der dem thread zugewisene beweglicher Agent einen fixen Agent in Sichtweise hat
- Änderung der aktuellen Bewegungsrichtung den Regeln folgend
- Berechnung der neuen Position
- Berechnung ob neue Position ausserhalb des Simulationsfeldes liegt. Wenn ja -¿ Abprall an der Wand.

Die Visualiserung der Simulation findet in mehreren Stufen statt. In der Host-Applikation wird nach jedem Simulationsschritt ein Bild generiert das den aktuellen Zustand mittels farbkodierter Punkte in einem Bild darstellt. Dazu wird aufgrund der Einfachkeit das simple PPM Format verwendet. Nach der Simulation werden die erzeugten ppm Bilder in das PNG-Format umgewandelt und ein Film mit HUFYU Codierung erzeugt. Wichtig bei der Wahl der Formate ist eine verlustfreie Kompression, da sonst Agenten, welche als einzelne Pixel dargestellt werden, verloren gehen könnten. Diese Darstellungskette wird als bashscript mithilfe von ffmpeg für die Codierung umgesetzt welche sowohl Einzelbilder der Simulationsschritte erzeugt als auch das Video mit der gesamten Simualtion.

1.2 Details

Wie in 1.1 beschrieben, wird in der hier verwendeten Implementierung der eigentliche Algorithmus auf der GPU ausgeführt. Es wurde darauf Wert gelegt eine möglichst universelle und erweiterbare Implementierung zu realisieren mit dem Focus auf eine gute Skalierbarkeit. Da bei dieser Aufgabenstellung zu erwarten ist, dass der Aufwand des Speichermanagementes und Speicherkopieren um einiges Zeitaufwendiger ist als die eigentliche Berechnung der Simulation wurde darauf Wert gelegt alle Speicheroperationen möglichst schnell abzuschliessen. D.h. es wurde für alle Agenten der generelle GPU-Speicher verwendet und auf eine Allocation der Streaming-Speicher verzichtet welche schneller sind, aber auf 32kB limitiert sind. Weiters wäre wie schon beschrieben dadurch ein höherer Zeitaufwand beim Speichermanagement zu erwarten was den Vorteil des schnelleren Speichers wohl zunichte machen würde.
Generell ist anzumerken dass die Generierung des Outputs und das Anschliessende umkodieren der Ergebnisse nach bisherigen Erfahrungen um einiges mehr Zeit in Anspruch nimmt als die Simulation selbst. Auch bei über 500000 beweglichen Agenten ist dies noch der Fall. Sollten längere Simulationsintervalle gewünscht werden ist es wohl als erstes Empfehlenswert nur alle N Simulationsschritte eine Ausgabe durchzuführen. Wenn dies nicht möglich ist bzw. ungewünscht kann man sich überlegen eine Speicheroptimierung durchzuführen. Auch eine höhere Anzahl an fixen Agenten sollte diese Entscheidung begünstigen.

Nm ... all moving Agents
Nf ... all fixed Agents
n ... available GPUs

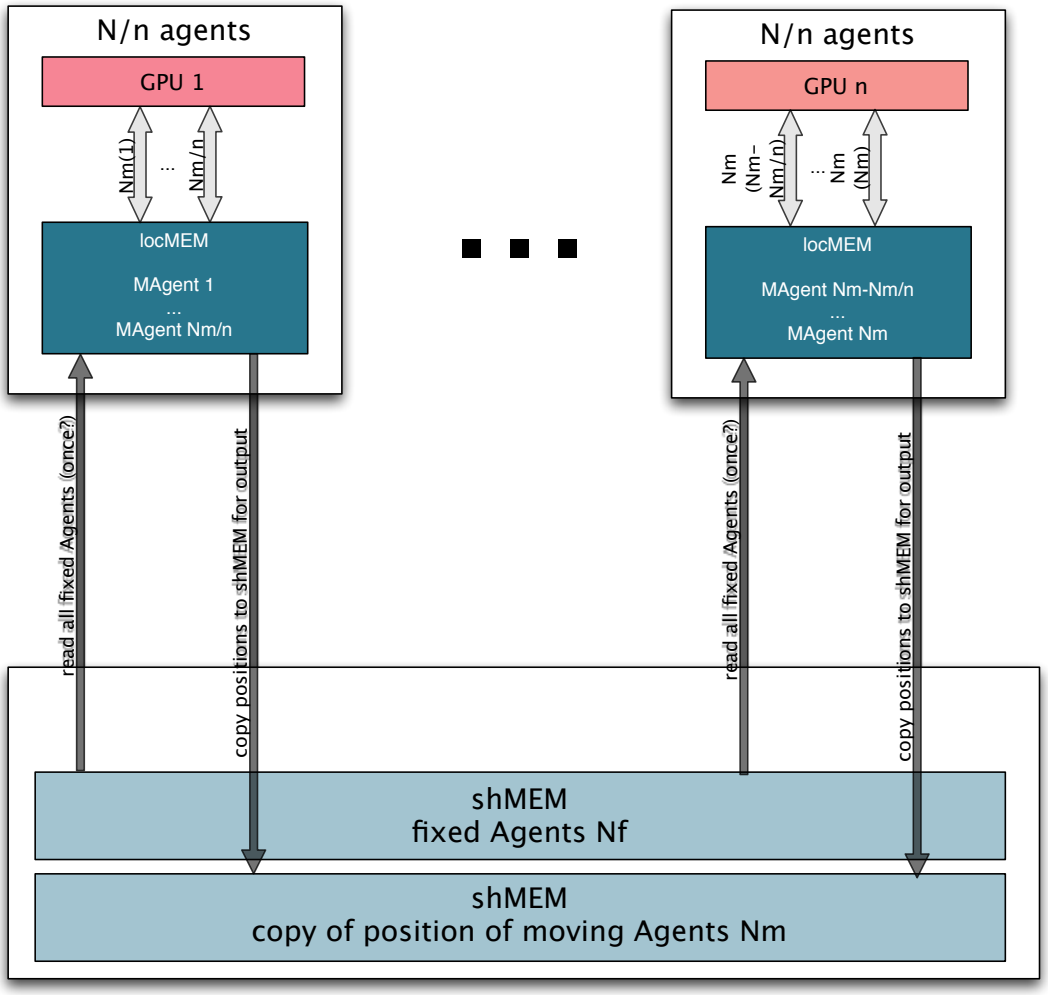

Abbildung 1.1: Basic Architecture

2 Simulationsergebnis

2.1 Simulation 1

Simulationsparameter:

- DimX, DimY: 800x800
- fixed agents: 12
- fixed agents radius: DimX/6
- moving agents: 100 000
- moving agents initial: random position/direction
- moving agents lookahead radius: 100
- influence factor: 0,5

In Abbildung 2.1 ist die Ausgangssituation dargestellt. Zu sehen ist die zufällige Verteilung der beweglichen Agenten. Bewegen sich die Agenten frei, sind diese grün eingezeichnet, sind sie von einem fixen Agenten beeinflusst(Parameter "moving agents lookahead radius“) sind sie türkis eingezeichnet. Fixe Agenten, welche die Bewegungsrichtung vorgeben, sind rot eingezeichnet. In diesem Anfangszustand mit vielen Agenten sind diese kaum erkennbar, da sie in der Zeichenebene unter den beweglichen Agenten liegen.

Wie man an Abbildung 2.2 und 2.3 beobachten kann, finden sich alle beweglichen Agenten nach einer endlichen Zeit in einem Orbit um die fixen Agenten ein. Wie wir aus der Aufgabenstellung wissen, besitzen alle fixen Agenten eine Wegweiser-Vektor der normal auf dem Eigenvektor vom Mittelpunkt des Kreises steht. D.h. dass sich alle beweglichen Agenten folglich in endlicher Zeit auf den äussersten Radius zubewegen müssen und diesen auch folglich irgendwann verlassen müssen. In Abbildung 2.3 ist gut zu erkennen wie sämtliche beweglichen Agenten regelmäßig den "Orbit“ verlassen um an der Aussenwand des Simulationsfeldes wieder abzuprallen. Dieses Abprallen steht so nciht in der Aufgabenstellung, und wurde von uns sinnvollerweise hinzugefügt. Ohne dieses vorgegebene Verhalten ist zu erwarten, dass sich gegen unendlichen Simulationsverlauf alle Agenten unendlich weit entfernen.

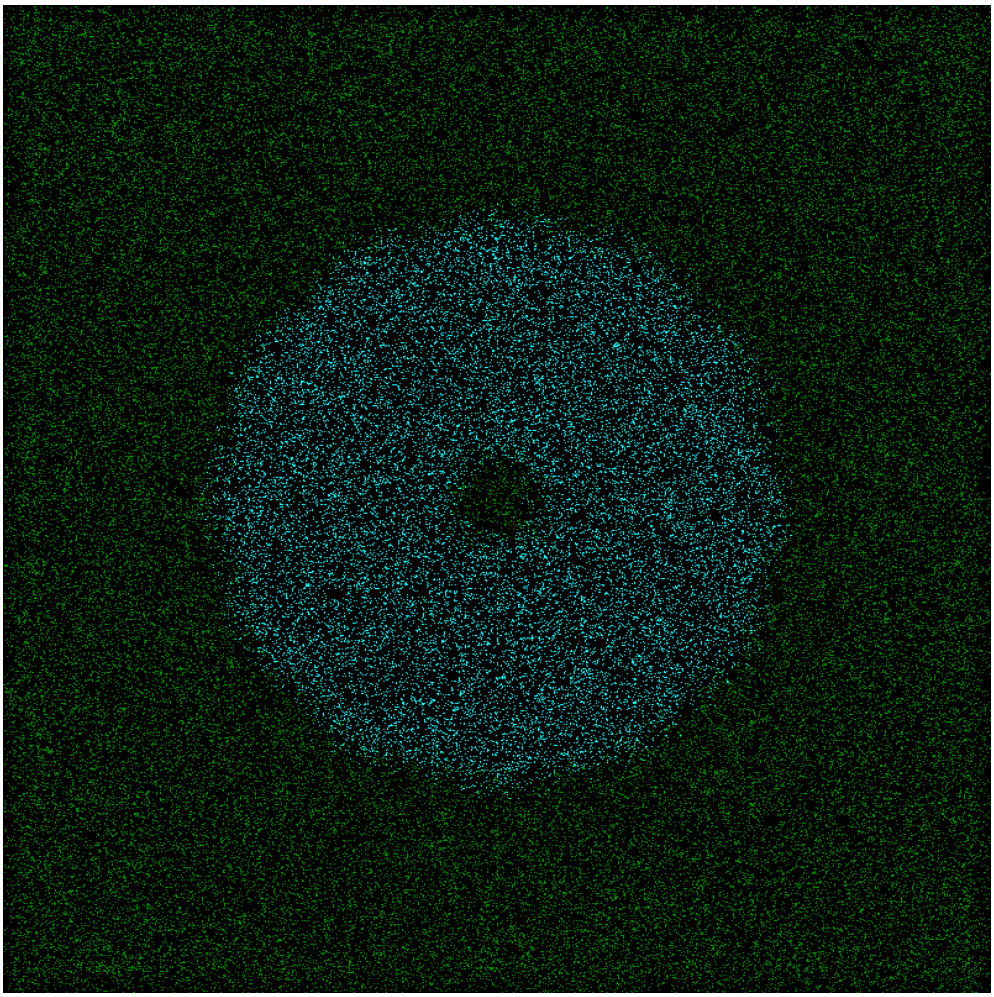

Abbildung 2.1: Sim1, Step 1

Eine interessante Beobachtung ist auch, dass sich die Austrittspunkte aus dem Orbit immer kurz vor dem Einfluss eines im Orbit folgendem fixen Agenten befinden. Diese Beobachtung lässt sich auch mit dem im letzten Punkt behandelten Verhalten erklären, dass sich alle beweglichen Agenten immer weiter richtung äussersten Orbit bewegen und auch am ehesten den rbit verlassen wenn die Anzahl der beinflussenden fixen Agenten am geringsten ist. Weiters ist mit fortlaufender Simulation zu beobachten dass besonders bei den in N,O,S,W stehenden fixen Agenten die beweglichen Agenten austreten bzw. häufen. Das ist dadurch zu erklären, dass bei einer Berührung der imaginären Wand die Agenten zurückgeworfen werden mit einer einfachen invertierung der kollidierenden Bewegungsrichtung. Vergleichbar auch mit einem Abprall einer Billardkugel an einer Bande. Da nun bei einem Austritt in N,O,S,W die Agenten sich auf einem fast direktem Kollisionskurs mit der begrenzenden Wand befinden, werden diese bei Erreichen auch fast direkt zurückgeworfen. Damit komme selbige wieder direkt in den Einfluss des gleichen fixen Agenten was in Folge wieder eine Ablenkung in die gegengesetzte Richtung gegen Wand bedeutet. Dieses Verhalten wiederholt sich bis sich der Agent im Einfluss eines anderen fixen Agenten(meistens der in Folgerichtung des Orbits) befindet, und von diesem mitabgelenkt wird. Abbildung

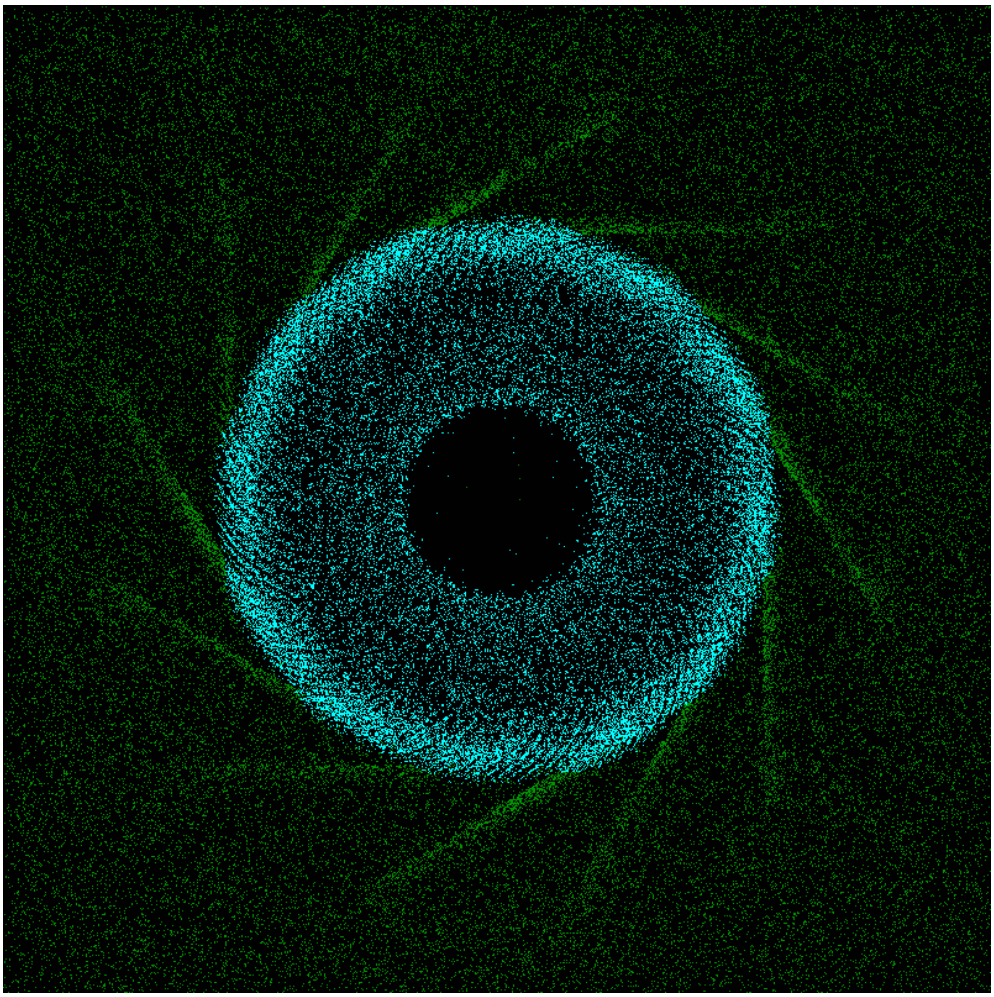

Abbildung 2.2: Sim1, Step 25

2.4 bestätigt diese Theorie.

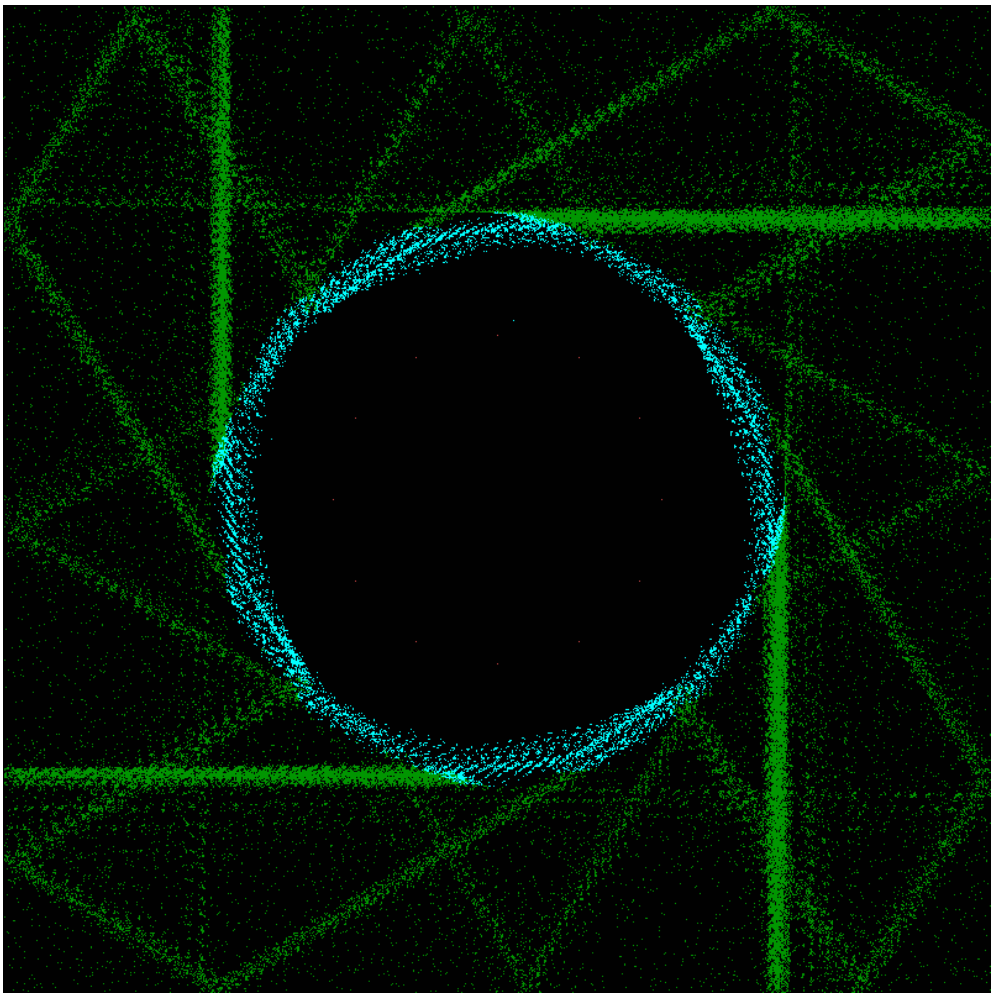

Abbildung 2.3: Sim1, Step200

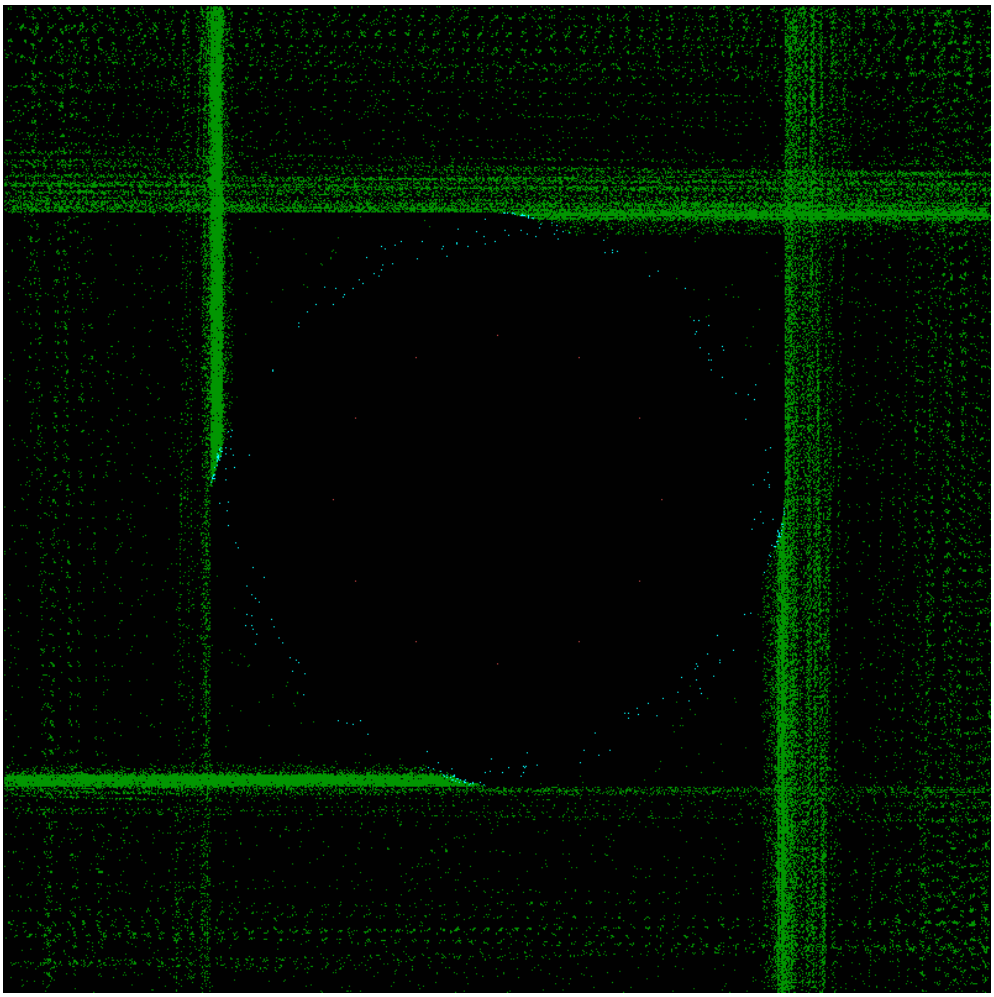

Abbildung 2.4: Sim1, Step1000

3 Listings

3.1 Invoking Script

```
# bin/sh

BIN="oclAgents"
BINPATH="bin/darwin/release"

#FFMPEG_OUT="out.gif"
FFMPEG_VIDEO_OUT="result/out.avi"
FFMPEG_PNG_OUT="result/png/out_%d.png"
#FFMPEG_PARAMS="-r 10 -b 1800 -r 25 -vcodec huffyuv -i out_%d.ppm"
FFMPEG_PARAMS="-i out_%d.bmp -r 25 -vcodec huffyuv"


make clean
make
TEMP_PATH=$(PWD)
cd $BINPATH
./$BIN
cd $TEMP_PATH
mv ./$BINPATH/*.ppm ./
rm -f $FFMPEG_VIDEO_OUT
rm -f result/png/*.png
ffmpeg -i out_%d.ppm $FFMPEG_PNG_OUT
ffmpeg -i out_%d.ppm out_%d.bmp
ffmpeg $FFMPEG_PARAMS $FFMPEG_VIDEO_OUT
rm -f *.ppm *.bmp
```

3.2 Host Code

```
/*
 * Agent based Simulation
 * Host File
 *
 *
 * 08 - Nov - 2009  VEC     init, not functional yet
 * 08 - Mar - 2010  VEC     fully functional
 */

// standard utilities and system includes
#include <oclUtils.h>
#include <math.h>
#include <stdlib.h>

// project include
#include "agents.h"

// max GPU's to manage for multi-GPU parallel compute
const unsigned int MAX_GPU_COUNT = 8;

// global variables
cl_context cxGPUContext;
cl_kernel multiplicationKernel[MAX_GPU_COUNT];
cl_command_queue commandQueue[MAX_GPU_COUNT];

////////////////////////////////////////////////////////////////////////////////
// declaration, forward
int runTest(int argc, const char** argv);
void printDiff(float*, float*, int, int);
void matrixMulGPU(cl_uint ciDeviceCount, cl_mem h_A, float* h_B_data, unsigned int mem_size_B, float* h_C );

extern "C"
void computeGold(float*, const float*, const float*, unsigned int, unsigned int, unsigned int);

////////////////////////////////////////////////////////////////////////////////
// Helper functions
////////////////////////////////////////////////////////////////////////////////

double executionTime(cl_event &event)
{
    cl_ulong start, end;

    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
```

```
44      clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
45
46      return (double)1.0e-9 * (end - start); // convert nanoseconds to seconds on return
47  }
48
49
50  void f_agents_init_circle(agent_container_t * container)
51  {
52     // angle differences in rad
53     float phi = (2*M_PI)/(container->f_count);
54
55
56     for(unsigned int i = 0; i < container->f_count; i++)
57     {
58          float * pos_x = &(container->f_agent_array[i].pos_x);
59          float * pos_y = &(container->f_agent_array[i].pos_y);
60
61          float * mov_x = &(container->f_agent_array[i].mov_x);
62          float * mov_y = &(container->f_agent_array[i].mov_y);
63
64          // calc positions
65          *pos_x = (-1)*CIRCLE_RADIUS*sin(phi*i);
66          *pos_y = CIRCLE_RADIUS*cos(phi*i);
67
68          // calc moving directions
69          *mov_x = (-1)*(*pos_y)*INIT_MOVE_FACTOR_FIXED;
70          *mov_y = (*pos_x)*INIT_MOVE_FACTOR_FIXED;
71
72          //type
73          container->f_agent_array[i].stat = STAT_FREE;
74
75          // output
76          shrLog(LOGBOTH, 0, "Element␣%d:␣pos␣x%f␣y%f;␣direction␣x%f␣y%f\n", i, *pos_x, *pos_y, *mov_x, *mov_y)
                ;
77     }
78
79  }
80
81  void m_agents_init_rand(agent_container_t * container)
82  {
83
84      srand( time(NULL) );
85
86      for(unsigned int i = 0; i < container->m_count; i++)
87      {
88          container->m_agent_array[i].pos_x = (rand() % DIM_X) - DIM_X/2;
89          container->m_agent_array[i].pos_y = (rand() % DIM_Y) - DIM_Y/2;
90          container->m_agent_array[i].mov_x = ((rand() % DIM_X) - DIM_X/2)*INIT_MOVE_FACTOR_MOVING;
91          container->m_agent_array[i].mov_y = ((rand() % DIM_Y) - DIM_Y/2)*INIT_MOVE_FACTOR_MOVING;
92          container->m_agent_array[i].stat = STAT_FREE;
93      }
94  }
95
96
97  void m_agents_init_null(agent_container_t * container)
98  {
99
100     for(unsigned int i = 0; i < container->m_count; i++)
101     {
102         container->m_agent_array[i].pos_x = 00.00;
103         container->m_agent_array[i].pos_y = 00.00;
104         container->m_agent_array[i].mov_x = 00.00;
105         container->m_agent_array[i].mov_y = 00.00;
106         container->m_agent_array[i].stat = STAT_FREE;
107     }
108 }
109
110
111 void print_moving_agents(agent_vector_t * m_agents, unsigned int count, const char * prefix)
112 {
113     unsigned int i;
114
115     if(prefix)
116     { shrLog(LOGBOTH, 0, "%s\n",prefix); }
117
118     for(i=0; i < count; i++)
119     {
120         agent_vector_t * a = &m_agents[i];
121         shrLog(LOGBOTH, 0, "Agent␣%d␣pos␣x%f␣y%f;␣direction␣x%f␣y%f␣occupied:␣%d\n", i, a->pos_x, a->pos_y, a
                ->mov_x, a->mov_y, a->stat);
122     }
123
124 }
125
126 void print_ppm(agent_container_t * container, unsigned int run_nr)  // do on gpu later on!
127 {
128     char ppm[DIM_X][DIM_Y][3];
129     char ppm_out[(DIM_X*DIM_Y*3)+1];
130
131     char filename[FILENAME_LENGTH_MAX];
132
133     memset(&ppm[0][0], 1, DIM_X*DIM_Y*3);
134
135     // fill fixed agents
136     for(unsigned int i = 0; i < container->f_count; i++)
137     {
138         int x = container->f_agent_array[i].pos_x;
```

```
139         int y = container->f_agent_array[i].pos_y;
140 
141         if ((x+DIM_X/2) >= 0 && (x+DIM_X/2) < DIM_X && (y+DIM_Y/2) >= 0 && (y+DIM_Y/2) < DIM_Y) // check for
                safety
142         {
143 
144             if(container->f_agent_array[i].stat == STAT_OCCUPIED1)
145             {
146                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][0] = COLOR_FIXED_AGENTS_OCC1_R;
147                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][1] = COLOR_FIXED_AGENTS_OCC1_G;
148                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][2] = COLOR_FIXED_AGENTS_OCC1_B;
149             }
150             else
151             {
152                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][0] = COLOR_FIXED_AGENTS_FREE_R;
153                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][1] = COLOR_FIXED_AGENTS_FREE_G;
154                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][2] = COLOR_FIXED_AGENTS_FREE_B;
155             }
156 
157         }
158         else
159         { shrLog(LOGBOTH,0, "ERROR:␣ppm_gen:␣agent␣%d␣out␣of␣range␣with␣x%d␣y%d\n", i,x,y); }
160 
161     }
162 
163     // fill mobile agents
164     for(unsigned int i = 0; i < container->m_count; i++)
165     {
166         int x = container->m_agent_array[i].pos_x;
167         int y = container->m_agent_array[i].pos_y;
168 
169         if ((x+DIM_X/2) >= 0 && (x+DIM_X/2) < DIM_X && (y+DIM_Y/2) >= 0 && (y+DIM_Y/2) < DIM_Y) // check for
                safety
170         {
171             if(container->m_agent_array[i].stat == STAT_OCCUPIED1)
172             {
173                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][0] = COLOR_MOVING_AGENTS_OCC1_R;
174                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][1] = COLOR_MOVING_AGENTS_OCC1_G;
175                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][2] = COLOR_MOVING_AGENTS_OCC1_B;
176             }
177             else
178             {
179                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][0] = COLOR_MOVING_AGENTS_FREE_R;
180                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][1] = COLOR_MOVING_AGENTS_FREE_G;
181                 ppm[-(x-DIM_X/2)][y+DIM_Y/2][2] = COLOR_MOVING_AGENTS_FREE_B;
182             }
183 
184         }
185         else
186         { shrLog(LOGBOTH,0, "ERROR:␣ppm_gen:␣agent␣%d␣out␣of␣range␣with␣x%d␣y%d\n", i,x,y); }
187 
188     }
189 
190     memcpy(ppm_out, ppm, DIM_X * DIM_Y * 3);
191     ppm_out[DIM_X*DIM_Y*3] = NULL;
192 
193     snprintf(filename, FILENAME_LENGTH_MAX, "out_%d.ppm", run_nr);
194 
195     FILE* f; // create a new file pointer
196     if((f=fopen(filename,"w"))==NULL)
197     { shrLog(LOGBOTH,0, "ERROR:␣could␣not␣open␣file\n"); }
198 
199     //ppm header
200     fprintf(f,"P6\n%d␣%d␣%d\n", DIM_X, DIM_Y, COLOR_MAX);
201 
202     // ppm binary info
203     fputs(ppm_out,f);
204 
205     fclose(f);
206 
207 }
208 
209 
210 ////////////////////////////////////////////////////////////////////////////
211 // Program main
212 ////////////////////////////////////////////////////////////////////////////
213 int main(int argc, const char** argv)
214 {
215     // start the logs
216     shrSetLogFileName ("oclMatrixMul.txt");
217     shrLog(LOGBOTH, 0, "%s␣Starting...\n\n", argv[0]);
218 
219     // run the code
220     if (runTest(argc, argv) != 0)
221     {
222         shrLog(LOGBOTH, 0, "FAILED␣!!!\n\n");
223     }
224 
225 }
226 
227 void AgentsGPU(cl_uint ciDeviceCount, agent_container_t * container)
228 {
229 
230     unsigned int f_count = container->f_count;
231     unsigned int m_count = container->m_count;
232 
233     int ret_code[MAX_GPU_COUNT];
```

```
234
235     agent_vector_t * f_host = container->f_agent_array;
236     agent_vector_t * m_host = container->m_agent_array;
237
238     agent_vector_t result_fixed[f_count];
239     agent_vector_t result_moving[m_count];
240
241     cl_mem GPUmem_fixed[MAX_GPU_COUNT];
242     cl_mem GPUmem_moving[MAX_GPU_COUNT];
243     cl_mem GPUmem_fcount[MAX_GPU_COUNT];
244     cl_mem GPUmem_mcount[MAX_GPU_COUNT];
245     cl_mem GPUmem_ret[MAX_GPU_COUNT];
246
247     cl_event GPUDone[MAX_GPU_COUNT];
248
249     cl_int ret = 0;
250
251     // Start the computation on each available GPU
252
253     // Create buffers for each GPU
254     // Each GPU will compute sizePerGPU rows of the result
255     int sizePerGPU = m_count / ciDeviceCount;
256
257     shrLog(LOGBOTH, 0, "every␣GPU␣does␣%d␣moving␣agents\n", sizePerGPU);
258
259     int workOffset[MAX_GPU_COUNT];
260     int workSize[MAX_GPU_COUNT];
261
262     workOffset[0] = 0;
263     for(unsigned int i=0; i < ciDeviceCount; ++i)
264     {
265         // Input buffer
266         workSize[i] = (i != (ciDeviceCount - 1)) ? sizePerGPU : (m_count - workOffset[i]);        /* last
                unit might not have to do full size */
267         shrLog(LOGBOTH, 0, "GPU␣%d␣does␣%d␣moving␣agents\n", i, workSize[i]);
268
269         //init errcode
270         ret_code[i] = 0;
271
272         GPUmem_fixed[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE, f_count * sizeof(agent_vector_t),
                NULL,NULL);
273         GPUmem_moving[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE, workSize[i] * sizeof(
                agent_vector_t), NULL,NULL);
274         GPUmem_fcount[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, sizeof(
                unsigned int), &f_count,NULL);
275         GPUmem_mcount[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR, sizeof(
                unsigned int), &workSize[i],NULL);
276         GPUmem_ret[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR, sizeof(int), &
                ret_code[i],NULL);
277
278         // Copy fixed agents and only assigned moving agents from host to device
279         clEnqueueWriteBuffer(commandQueue[i], GPUmem_fixed[i], CL_TRUE, 0, f_count * sizeof(agent_vector_t),
                f_host, NULL, NULL, NULL);
280         clEnqueueWriteBuffer(commandQueue[i], GPUmem_moving[i], CL_TRUE, workOffset[i], m_count * sizeof(
                agent_vector_t), m_host, NULL, NULL, NULL);
281
282         // set the args values
283         clSetKernelArg(multiplicationKernel[i], 0, sizeof(cl_mem), (void *)&(GPUmem_fixed[i]));  /* address
                of fixed agents */
284         clSetKernelArg(multiplicationKernel[i], 1, sizeof(cl_mem), (void *)&(GPUmem_moving[i])); /* address
                of moving agents */
285         clSetKernelArg(multiplicationKernel[i], 2, sizeof(cl_mem), (void *)&GPUmem_fcount);       /* count of
                 fixed agents */
286         clSetKernelArg(multiplicationKernel[i], 3, sizeof(cl_mem), (void *)&GPUmem_mcount);   /* count of
                moving agents */
287         clSetKernelArg(multiplicationKernel[i], 4, sizeof(cl_mem), (void *)&GPUmem_ret[i]);   /* count of
                moving agents */
288
289         if(i+1 < ciDeviceCount)
290             workOffset[i + 1] = workOffset[i] + workSize[i];
291     }
292
293     // launch kernels on devices
294
295     shrLog(LOGBOTH, 0, "INIT:\n");
296     print_moving_agents(m_host, m_count, "MOVING␣AGENTS");
297
298     for(unsigned int run = 0; run < RUNS_MAX; run++)
299     {
300
301         for(unsigned int i = 0; i < ciDeviceCount; i++)
302         {
303
304             shrLog(LOGBOTH, 0, "run␣#%d\n", run);
305             size_t global_work_size = AGENTS_MOVING_COUNT;
306             ret = clEnqueueNDRangeKernel(commandQueue[i], multiplicationKernel[i], 1, NULL, &global_work_size
                    , NULL, 0, NULL, NULL);
307             //ret = clEnqueueTask(commandQueue[i], multiplicationKernel[i], 0, NULL, &GPUDone[i]);
308             if (ret != CL_SUCCESS)
309             { shrLog(LOGBOTH, 0, "enqueue␣tsk␣failed␣with␣ret␣%d\n",ret); }
310             (void)clFlush(commandQueue[i]);
311         }
312
313         // CPU sync with GPU
314         //clWaitForEvents(ciDeviceCount, GPUDone);
315
316         for(unsigned int i = 0; i < ciDeviceCount; i++)
```

```
317         {
318             clEnqueueReadBuffer(commandQueue[i], GPUmem_fixed[i], CL_TRUE, 0,
319                                 f_count * sizeof(agent_vector_t), result_fixed, 0, NULL, NULL);
320             // blocking copy of result from device to host
321             clEnqueueReadBuffer(commandQueue[i], GPUmem_moving[i], CL_TRUE, workOffset[i] * sizeof(
                    agent_vector_t),
322                                 workSize[i] * sizeof(agent_vector_t), &result_moving[workOffset[i]], 0, NULL, &
                                    GPUDone[i]);
323             clEnqueueReadBuffer(commandQueue[i], GPUmem_moving[i], CL_TRUE, workOffset[i] * sizeof(
                    agent_vector_t),
324                                 workSize[i] * sizeof(agent_vector_t), &(container->m_agent_array[workOffset[i]]),
                                       0, NULL, &GPUDone[i]);
325
326             shrLog(LOGBOTH, 0, "ret␣from␣Device␣%d␣is␣%d\n", i, ret_code[i]);
327
328         }
329
330         // CPU sync with GPU
331         clWaitForEvents(ciDeviceCount, GPUDone);
332
333         print_ppm(container, run);
334
335         shrLog(LOGBOTH, 0, "RUN␣%d\n", run);
336         //print_moving_agents(result_fixed, f_count, "FIXED AGENTS");
337         //print_moving_agents(result_moving, m_count, "MOVING AGENTS");
338
339     }
340
341     // stop and log timer
342     #ifdef GPU_PROFILING
343         double dSeconds = shrDeltaT(0);
344         double dSize = ((double)WA * (double)HA * (double)WB * (double)HB);
345         shrLog(LOGBOTH | MASTER, 0, "oclMatrixMul,␣Throughput␣=␣%.4f,␣Time␣=␣%.5f,␣Size␣=␣%.0f,␣NumDevsUsed␣=
                ␣%d,␣Workgroup␣=␣%u\n",
346                1.0e-9 * dSize/dSeconds, dSeconds, dSize, ciDeviceCount, localWorkSize[0] * localWorkSize[1])
                       ;
347
348         // Print kernel timing per GPU
349         for(unsigned int i = 0; i < ciDeviceCount; i++)
350         {
351             shrLog(LOGBOTH, 0, "␣␣Kernel␣execution␣time␣on␣GPU␣%d␣\t:␣%.5f␣s\n", i, executionTime(
                    GPUExecution[i]));
352         }
353         shrLog(LOGBOTH, 0, "\n");
354      #endif
355
356     // Release mem and event objects
357     for(unsigned int i = 0; i < ciDeviceCount; i++)
358     {
359         clReleaseMemObject(GPUmem_fixed[i]);
360         clReleaseMemObject(GPUmem_moving[i]);
361       clReleaseMemObject(GPUmem_fcount[i]);
362       clReleaseMemObject(GPUmem_mcount[i]);
363       clReleaseEvent(GPUDone[i]);
364     }
365 }
366
367
368 ////////////////////////////////////////////////////////////////////////////
369 //! Run a simple test for
370 ////////////////////////////////////////////////////////////////////////////
371 int runTest(int argc, const char** argv)
372 {
373     cl_uint ciDeviceCount = 0;
374     cl_int ciErrNum = CL_SUCCESS;
375
376     // create the OpenCL context on available GPU devices
377     cxGPUContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, &ciErrNum);
378     if (ciErrNum != CL_SUCCESS)
379     {
380         shrLog(LOGBOTH, 0, "Error:␣Failed␣to␣create␣OpenCL␣context!␣return␣with:␣%d\n", ciErrNum);
381         return ciErrNum;
382     }
383
384     if(shrCheckCmdLineFlag(argc, (const char**)argv, "device")) /* not tested, and thus should not be used */
385     {
386         // User specified GPUs
387         char* deviceList;
388         char* deviceStr;
389         shrGetCmdLineArgumentstr(argc, (const char**)argv, "device", &deviceList);
390
391         #ifdef WIN32
392             deviceStr = strtok_s (deviceList,"␣,.-", &next_token);
393         #else
394             deviceStr = strtok (deviceList,"␣,.-");
395         #endif
396         while(deviceStr != NULL)
397         {
398             // get and print the device for this queue
399             cl_device_id device = oclGetDev(cxGPUContext, atoi(deviceStr));
400             shrLog(LOGBOTH, 0, "Device␣%d:\n", ciDeviceCount);
401             oclPrintDevName(LOGBOTH, device);
402
403             // create command queue
404             commandQueue[ciDeviceCount] = clCreateCommandQueue(cxGPUContext, device, 0, &ciErrNum);
405             if (ciErrNum != CL_SUCCESS)
406             {
```

```
407                 shrLog(LOGBOTH, 0, "␣Error␣%i␣in␣clCreateCommandQueue␣call␣!!!\n\n", ciErrNum);
408                 return ciErrNum;
409             }
410 
411             #ifdef GPU_PROFILING
412                 ciErrNum = clSetCommandQueueProperty(commandQueue[ciDeviceCount], CL_QUEUE_PROFILING_ENABLE,
                        CL_TRUE, NULL);
413                 if (ciErrNum != CL_SUCCESS)
414                 {
415                     shrLog(LOGBOTH, 0, "␣Error␣%i␣in␣clSetCommandQueueProperty␣call␣!!!\n\n", ciErrNum);
416                     return ciErrNum;
417                 }
418             #endif
419 
420              ++ciDeviceCount;
421 
422             #ifdef WIN32
423                 deviceStr = strtok_s (NULL,"␣,.-", &next_token);
424             #else
425                 deviceStr = strtok (NULL,"␣,.-");
426             #endif
427         }
428 
429         free(deviceList);
430     }
431     else
432     {
433         // Find out how many GPU's to compute on all available GPUs
434       size_t nDeviceBytes;
435       ciErrNum |= clGetContextInfo(cxGPUContext, CL_CONTEXT_DEVICES, 0, NULL, &nDeviceBytes);
436       ciDeviceCount = (cl_uint)nDeviceBytes/sizeof(cl_device_id);
437 
438         if (ciErrNum != CL_SUCCESS)
439         {
440             shrLog(LOGBOTH, 0, "␣Error␣%i␣in␣clGetDeviceIDs␣call␣!!!\n\n", ciErrNum);
441             return ciErrNum;
442         }
443         else if (ciDeviceCount == 0)
444         {
445             shrLog(LOGBOTH, 0, "␣There␣are␣no␣devices␣supporting␣OpenCL␣(return␣code␣%i)\n\n", ciErrNum);
446             return -1;
447         }
448 
449         // create command-queues
450         for(unsigned int i = 0; i < ciDeviceCount; ++i)
451         {
452             // get and print the device for this queue
453             cl_device_id device = oclGetDev(cxGPUContext, i);
454             shrLog(LOGBOTH, 0, "Device␣%d:\n", i);
455             oclPrintDevName(LOGBOTH, device);
456 
457             // create command queue
458             commandQueue[i] = clCreateCommandQueue(cxGPUContext, device, 0, &ciErrNum);
459             if (ciErrNum != CL_SUCCESS)
460             {
461                 shrLog(LOGBOTH, 0, "␣Error␣%i␣in␣clCreateCommandQueue␣call␣!!!\n\n", ciErrNum);
462                 return ciErrNum;
463             }
464             #ifdef GPU_PROFILING
465                 clSetCommandQueueProperty(commandQueue[i], CL_QUEUE_PROFILING_ENABLE, CL_TRUE, NULL);
466             #endif
467         }
468     }
469 
470     // allocate the memory for the fixed agents
471     // the array keeping all the fixed agents
472     agent_vector_t * f_agent_array = (agent_vector_t *)malloc(AGENTS_FIXED_COUNT*sizeof(agent_vector_t));
473     shrLog(LOGBOTH, 0, "␣alloced␣%d␣bytes␣host␣memory␣for␣%d␣fixed␣agents\n", AGENTS_FIXED_COUNT*sizeof(
            agent_vector_t), AGENTS_FIXED_COUNT);
474 
475     // array keping all the moving agents
476     agent_vector_t * m_agent_array = (agent_vector_t *)malloc(AGENTS_MOVING_COUNT*sizeof(agent_vector_t));
477     shrLog(LOGBOTH, 0, "␣alloced␣%d␣bytes␣host␣memory␣for␣%d␣moving␣agents\n", AGENTS_MOVING_COUNT*sizeof(
            agent_vector_t), AGENTS_MOVING_COUNT);
478 
479     // container init
480     agent_container_t * agent_container = (agent_container_t *)malloc(sizeof(agent_container_t));
481     shrLog(LOGBOTH, 0, "␣alloced␣%d␣bytes␣host␣memory␣for␣the␣container\n", sizeof(agent_container_t));
482 
483     agent_container->f_count = AGENTS_FIXED_COUNT;
484     agent_container->m_count = AGENTS_MOVING_COUNT;
485     agent_container->f_agent_array = f_agent_array;
486     agent_container->m_agent_array = m_agent_array;
487 
488 
489 
490     // init fixed agents in circle
491     f_agents_init_circle(agent_container);
492     //m_agents_init_null(agent_container);
493     m_agents_init_rand(agent_container);
494 
495 
496     // Program Setup
497     size_t program_length;
498     const char* header_path = shrFindFilePath("agents.h", argv[0]);
499     char* header = oclLoadProgSource(header_path, "", &program_length);
500     if(!header)
```

```
501     {
502         shrLog(LOGBOTH, 0, "Error: Failed to load the header %s!\n", header_path);
503         return -1000;
504     }
505     const char* source_path = shrFindFilePath("agents.cl", argv[0]);
506     char *source = oclLoadProgSource(source_path, header, &program_length);
507     if(!source)
508     {
509         shrLog(LOGBOTH, 0, "Error: Failed to load compute program %s!\n", source_path);
510         return -2000;
511     }
512
513     // create the program
514     cl_program cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char **)&source,
515                                                      &program_length, &ciErrNum);
516     if (ciErrNum != CL_SUCCESS)
517     {
518         shrLog(LOGBOTH, 0, "Error: Failed to create program\n");
519         return ciErrNum;
520     }
521     free(header);
522     free(source);
523
524     // build the program
525     ciErrNum = clBuildProgram(cpProgram, 0, NULL, "-cl-mad-enable", NULL, NULL);
526     if (ciErrNum != CL_SUCCESS)
527     {
528         // write out standard error, Build Log and PTX, then return error
529         shrLog(LOGBOTH | ERRORMSG, ciErrNum, STDERROR);
530         oclLogBuildInfo(cpProgram, oclGetFirstDev(cxGPUContext));
531         oclLogPtx(cpProgram, oclGetFirstDev(cxGPUContext), "oclAgents.ptx");
532         return ciErrNum;
533     }
534
535     // Create Kernel
536     for(unsigned int i=0; i<ciDeviceCount; ++i) {
537         shrLog(LOGBOTH, 0, "Creating Kernel for Device %d of total %d devices.\n", i, ciDeviceCount);
538         multiplicationKernel[i] = clCreateKernel(cpProgram, "agents", &ciErrNum);
539         if (ciErrNum != CL_SUCCESS)
540         {
541             shrLog(LOGBOTH, 0, "Error: Failed to create kernel err %d\n", ciErrNum);
542             return ciErrNum;
543         }
544     }
545
546     // Run multiplication on 1..deviceCount GPUs to compare improvement
547     shrLog(LOGBOTH, 0, "\nRunning Computations on 1 - %d GPU's...\n", ciDeviceCount);
548     for(unsigned int k = 1; k <= ciDeviceCount; ++k)
549     {
550         AgentsGPU(k, agent_container);
551         /* AgentsGPU(k, buffer_fixed, buffer_moving, agent_container->f_count, agent_container->m_count); */
552     }
553
554     // clean up OCL resources
555     // clReleaseMemObject(buffer_moving);
556     // clReleaseMemObject(buffer_fixed);
557     for(unsigned int k = 0; k < ciDeviceCount; ++k)
558     {
559         clReleaseKernel( multiplicationKernel[k] );
560         clReleaseCommandQueue( commandQueue[k] );
561     }
562     clReleaseProgram(cpProgram);
563     ciErrNum = clReleaseContext(cxGPUContext);
564     if( ciErrNum != CL_SUCCESS)
565         shrLog(LOGBOTH, 0, "Error: Failed to release context: %d\n", ciErrNum);
566
567     // clean up memory
568     free(f_agent_array);
569     free(m_agent_array);
570     free(agent_container);
571
572     return 0;
573 }
```

3.3 Target Code

```
1  /*
2   * Agent based Simulation
3   * Device File
4   *
5   *
6   * 08 - Nov - 2009  VEC     init, not functional yet
7   * 08 - Mar - 2010  VEC     fully functional, and tested
8   */
9
10
11 //unsigned int gpu_run = 0;
12 ////////////////////////////////////////////////////////////////////////////
13 //! Matrix multiplication on the device: C = A * B
14 //! WA is A's width and WB is B's width
15 ////////////////////////////////////////////////////////////////////////////
```

```
16 __kernel void
17 agents( __global agent_vector_t * fixed_agents, __global agent_vector_t * moving_agents, __global unsigned
       int * f_count,
18         __global unsigned int * m_count, __global int * ret)
19 {
20     unsigned int r2 = LOOKAHEAD_RADIUS * LOOKAHEAD_RADIUS;
21     unsigned int m,f,i;
22 
23     // get moving agent to work on
24     m = get_global_id(0);
25 
26     unsigned int in_radius_count = 0;
27     int new_dir_x = 0;
28     int new_dir_y = 0;
29     //agent_vector_t * in_radius[*f_count];
30     agent_vector_t in_radius[AGENTS_FIXED_COUNT]; // improve performance with pointer traversal
31 
32     // check for every fixed agent if in radius
33     for(f=0; f < *f_count; f++)
34     {
35         float dist_x = fixed_agents[f].pos_x - moving_agents[m].pos_x;
36         float dist_y = fixed_agents[f].pos_y - moving_agents[m].pos_y;
37 
38         //if(hypot(dist_x,dist_y) <= LOOKAHEAD_RADIUS) // check if in radius
39 
40         if((dist_x*dist_x)+(dist_y*dist_y) <= r2)
41         {
42             in_radius[in_radius_count] = fixed_agents[f];
43             in_radius_count++;
44 
45         }
46     }
47     // now we know all fixed agents in range, calculate new dir now
48     for(i=0; i < in_radius_count; i++)
49     {
50         new_dir_x += in_radius[i].mov_x * INFLUENCE_FACTOR;
51         new_dir_y += in_radius[i].mov_y * INFLUENCE_FACTOR;
52 
53         //moving_agents[m].mov_x += in_radius[i].mov_x;
54         //moving_agents[m].mov_y += in_radius[i].mov_y;
55     }
56 
57     if(in_radius_count) // only if there are some in radius
58     {
59         moving_agents[m].mov_x += new_dir_x;
60         moving_agents[m].mov_y += new_dir_y;
61         moving_agents[m].mov_x = moving_agents[m].mov_x / (in_radius_count * INFLUENCE_FACTOR + 1);
62         moving_agents[m].mov_y = moving_agents[m].mov_y / (in_radius_count * INFLUENCE_FACTOR + 1);
63         moving_agents[m].stat  = STAT_OCCUPIED1;
64     }
65     else
66     {
67         moving_agents[m].stat  = STAT_FREE;
68     }
69 
70     // check borders
71     if ((moving_agents[m].pos_x + moving_agents[m].mov_x) >= DIM_X/2) { moving_agents[m].mov_x = -
           moving_agents[m].mov_x; }
72     if ((moving_agents[m].pos_x + moving_agents[m].mov_x) < -DIM_X/2) { moving_agents[m].mov_x = -
           moving_agents[m].mov_x; }
73     if ((moving_agents[m].pos_y + moving_agents[m].mov_y) >= DIM_Y/2) { moving_agents[m].mov_y = -
           moving_agents[m].mov_y; }
74     if ((moving_agents[m].pos_y + moving_agents[m].mov_y) < -DIM_Y/2) { moving_agents[m].mov_y = -
           moving_agents[m].mov_y; }
75 
76     // now move agent
77     moving_agents[m].pos_x += moving_agents[m].mov_x;
78     moving_agents[m].pos_y += moving_agents[m].mov_y;
79 
80     //debug
81     //moving_agents[m].pos_x = gpu_run;
82 
83     //gpu_run++;
84 
85 
86     *ret = 1;
87 
88 
89 
90 }
```

3.4 shared Header

```
1 /*
2  * Agent based Simulation
3  * shared Header File
4  *
5  *
6  * 08 - Nov - 2009  VEC     init, not functional yet
7  * 08 - Mar - 2010  VEC     fully functional, and tested
8  */
```

```
9
10
11  #ifndef _AGENTS_H_
12  #define _AGENTS_H_
13
14  // Thread block size
15  #define BLOCK_SIZE 16
16
17  // Dimensions of the Field
18  #define DIM_X 800
19  #define DIM_Y 800
20  #define INIT_MOVE_FACTOR_FIXED 0.1
21  #define INIT_MOVE_FACTOR_MOVING 0.05
22
23  // how many runs
24  #define RUNS_MAX 1000
25
26  // How many Agents?
27  #define AGENTS_FIXED_COUNT 12
28  #define AGENTS_MOVING_COUNT 100000       /* i think it should be a multiple of the available GPUs*/
29
30  // looking distance of moving agents
31  #define LOOKAHEAD_RADIUS 100
32  #define INFLUENCE_FACTOR 0.5
33
34  // radius of the initial circle; at least DIM_X/2
35  #define CIRCLE_RADIUS DIM_X/6
36
37  // print stuff
38  // NOTE: do not use 0 as RGB value!
39  #define FILENAME_LENGTH_MAX 20
40  #define COLOR_MAX 255
41  #define COLOR_MOVING_AGENTS_FREE_R 1
42  #define COLOR_MOVING_AGENTS_FREE_G 150
43  #define COLOR_MOVING_AGENTS_FREE_B 1
44  #define COLOR_MOVING_AGENTS_OCC1_R 1
45  #define COLOR_MOVING_AGENTS_OCC1_G 255
46  #define COLOR_MOVING_AGENTS_OCC1_B 255
47
48  #define COLOR_FIXED_AGENTS_FREE_R 200
49  #define COLOR_FIXED_AGENTS_FREE_G 70
50  #define COLOR_FIXED_AGENTS_FREE_B 70
51  #define COLOR_FIXED_AGENTS_OCC1_R 200
52  #define COLOR_FIXED_AGENTS_OCC1_G 1
53  #define COLOR_FIXED_AGENTS_OCC1_B 1
54
55  // how many entries per Agent
56  //#define AGENT_ELEMENT_COUNT 4 /* pos_x, pos_y, mov_x, mov_y - should be done by sizeof()/sizeof(float or so
        )*/
57  //#define AGENT_ELEMENT_COUNT ((sizeof(agent_vector_t)/sizeof(float))) /* like this? */
58
59  // typedefs
60  #define STAT_FREE 0
61  #define STAT_OCCUPIED1 1
62
63  typedef short t_stat;
64
65  typedef struct agent_vector {
66      float  pos_x;
67      float  pos_y;
68      float  mov_x;
69      float  mov_y;
70      t_stat stat;
71  } agent_vector_t;
72
73  /* not sure if this a good iea to keep both in the same container */
74  typedef struct agent_container {
75      unsigned int f_count;
76      unsigned int m_count;
77      agent_vector_t * f_agent_array;
78      agent_vector_t * m_agent_array;
79  } agent_container_t;
80
81
82  #endif // _AGENTS_H_
```