Implementation of Virtual Embryology using the Thrust library for CUDA

Anna Aichmayr Markus Dauschan

March 2, 2011
Contents

1 First Implementations 2
   1.1 Concept ............................................. 2
   1.2 Implementation in NetLogo .......................... 4
   1.3 First Implementation in C++ ........................ 4

2 A restart of the Project 6
   2.1 A new C++ Implementation .......................... 6

3 Implementation in CUDA using Thrust 8
   3.1 Introduction to Thrust ............................... 8
   3.2 Issues concerning the implementation using Thrust .... 10

4 Conclusion 12

5 Appendix 13
   5.1 References ........................................... 13
1

First Implementations

1.1 Concept

The model to be described is based on the findings of evolutionary developmental biology (evo devo) and is determined to emphasize the fundamental importance of the development process additionally to genetic material, forming virtual embryos and furthermore Artificial Neural Networks (ANN). It was originally developed by Ronald Thenius in 2008 and extended by Michael Bodi 2009 [1]. Several Genes are forming toolkits, turning on and off genes that are needed for certain adaptive reactions to the development surrounding leading to diverse shapes and forms emerging from the same genetic base [2]. The aim of the model is amongst others to create certain patterns of ANN which match Neural Networks found in real embryos. This is to be achieved by provided respectively cell emitted substances which act as morphogens and hence form the embryo and the resulting neural network by triggering certain cellular behaviors. The concept used in this model is based on multi-agent simulations where each cell acts and reacts individually to any influences applied from its surroundings. These influences may consist of the existence of specific morphogens near the cell or the presence of other cells. Potential reactions of the cell on these influences are proliferation, development to a node within the ANN or death. Apart from that a cell can emit chemical morphogens itself and therefore trigger these behaviors or the emittance of further substances within other cells in its vicinity. This process of specialization is considered to be reversible. All possible actions of the cell are defined in the genome contained by each cell. Following the concepts of evo devo an individual embryo is then exposed to evolution altering its genome. Each of the resulting individuals is then measured up against certain fitness ideals and hence allowed to reproduce or not depending on its score. Genomes are considered to be diploid, therefore the reproduction process takes two individuals to form a new one, merging their two genomes into a genuine novel genome during a simplified crossover process. Application of this virtual evolution process already results in self organized behaviour of the embryo cells that form shapes and networks which can be compared to findings in real embryos.
Figure 1.1: Model overview indicating the subunits of the model that control the development process [3]

Figure 1.2: Scheme of the evolutionary optimization process and extraction of the ANN [3]
1.2 Implementation in NetLogo

The implementation in NetLogo considers all main concepts described above. To simplify the model for simulations it is rendered to be discrete and two-dimensional. The virtual world containing the embryo consists of a grid of 'patches' which may or may not hold a cell of the embryo. Each of these patches may also contain a certain amount of morphogens, even if it is not occupied by a cell. Diffusion of the morphogens is also rendered to be discrete resulting in the possibility to monitor the substance levels for every timestep in every cell respectively patch [1]. If a cell duplicates the additional cell will move all cells in the direction that holds the least number of cells. This results from the consideration that pushing other cells always takes place in the direction of the least pressure. Genomes defining the emission of morphogens and rates which trigger cells to emit morphogens or other actions are the base of these simulations. These genomes are realised as files which are read in by NetLogo enabling an easy switch between a vast variety of embryo shapes. Every genome holds several genes which consist of a quadruple of unsigned integer values. The first number contained by the gene represents the index of a substance which triggers the gene followed by minimum and maximum values of the substance recepted. The last parameter stands for the reaction of the cell to the recepted morphogen. As shown above this can be proliferation, emittance of another morphogen, building of links and cell death. The NetLogo implementation also features an evolution process that enables the automated creation of genomes that is followed by a selection of fitness. In this case fitness is related to the shape the cells form given a certain genome.

1.3 First Implementation in C++

The implementation of the model core in C++ was done in 2009 by Christopher Schwarzer. Following an object oriented approach the C++ implementation of the model uses the following main classes separated into two main routines:

- NeuralGenesis
  - class NeuralGenesis
  - struct Patch
  - class Cell
  - class Genome
- RTNN (Recurrent Time Neural Network)
  - class rtnn
  - class Neuron
  - struct Link
  - class ActivationFunctions
  - struct fixedtype
Neural Genesis is the main interface class for embryonal development. It generates a grid of patches which is occupied at first with only one cell. During the process of embryogenesis the cell proliferates and can be set to emit morphogens to influence the shape developed by the group of cells.

RTNN stands for Recurrent Time Neural Network which is the second main routine in the program. It is basically intended to create and extract the neural network resulting from the 'NeuralGenesis' process. In future development of the program this class can act as an interface to a neural net interpreter. All cells are transformed into neurons and links between them are established according to the data obtained in the 'NeuralGenesis' process.

One main difference between the NetLogo and the C++ implementation also lies within the genomes. The C++ genome holds an additional fifth parameter $\in [0, ..., 255]$ which stands for the release rate of morphogens increasing the grade of complexity of the simulation.
A restart of the Project

2.1 A new C++ Implementation

Due to a change of requirements to the core of the software, the implementation had to be restarted completely in January 2011. These requirements were including a correct object-oriented implementation approach and a major change within the cell class, as it should also be able to pose as a robot. The basic idea behind this was to be also able to form robot organisms out of homogenous robot swarms that are featuring an ability to dock to each other. Also classes 'Cell' and 'Patch' were merged into the new cell class, which does not contain information about its position.

Figure 2.1: C++ class hierarchy after the reimplementation featuring Evolution. Dotted lines indicate pointer class members.
Figure 2.2: First results using the new implemented virtual embryogenesis and its evolutionary engine. Visible is the formed organism and the primary gradient which is induced into the cell in the center of the grid [4].
3

Implementation in CUDA using Thrust

3.1 Introduction to Thrust

Thrust is an open-source template library for developing CUDA applications. It is modeled after the C++ STL and brings a familiar abstraction layer to the realm of GPU computing.

A big advantage of Thrust is the programmer productivity, one can rapidly develop complex applications due to many pre-assembled algorithms. Furthermore one does not have to care about details of CUDA, because they are all hidden in the implemented algorithms. So high performance can be obtained with minimal programmer effort.

Generic programming is encouraged - the idea behind Thrust is that you don’t have to ”reinvent the wheel.” Due to its constitution as a template library Thrust can be easily integrated in existing CUDA and C/C++ code.

Two containers are implemented in Thrust:

```cpp
thrust::host_vector<T>
thrust::device_vector<T>
```

Using those vectors makes the code more concise and readable, especially because cudaMemcpy, cudaMemcpy and cudaMemcpy are hidden. Moreover they are compatible with STL containers, the integration is very easy. In the following paragraph two examples are shown.

This example describes allocation and manipulation of Thrust vectors.

```cpp
1 // allocate host vector with two elements
2 thrust::host_vector<int> h_vec(2);
3
4 // copy host vector to device
5 thrust::device_vector<int> d_vec = h_vec;
```
// manipulate device values from the host

d_vec[0] = 13;
d_vec[1] = 27;
std::cout << "sum:" << d_vec[0] + d_vec[1] << std::endl;

// vector memory automatically released

In this second example one can see the easy interchangeability of Thrust
vectors and the C++ STL containers.

// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(),
             d_vec.begin());

// alternative method
thrust::device_vector<int> d_vec(h_list.begin(),
                                 h_list.end());

Besides vectors there is another very important class which comes along with
Thrust: Iterators.
In this context they also behave like pointers, e.g. one can define a sequence
by a pair of iterators. The memory space can be kept track of and they are
convertible to raw (traditional) pointers.
The use is similar and as easy as the use of raw pointers as one can see in the
following example:

// allocate device vector
thrust::device_vector<int> d_vec(4);
thrust::device_vector<int>::iterator begin = d_vec.begin();
thrust::device_vector<int>::iterator end = d_vec.end();

// compute size of sequence [begin, end)
int length = end - begin;

// define a sequence of 3 elements
end = d_vec.begin() + 3;
*begin = 13; // same as d_vec[0] = 13;
int temp = *begin; // same as temp = d_vec[0];
begin++; // advance iterator one position
*begin = 25; // same as d_vec[1] = 25;

Last but not least Thrust provides many standard algorithms, like transfor-
mations, reductions, sums, sortings. Those algorithms are generic, which means
that general types like int, float and user-defined structures and operators like
+, are supported. They have default arguments as we know from the C++
STL, like reduce(begin, end).

Due to the fact, that Thrust is still under development, the algorithms are
improving and even more and complex algorithms are in progress.

9
3.2 Issues concerning the implementation using Thrust

After getting to know thrust, compiling and testing the example programs, the way to implement our virtual embryogenesis on a graphic device seemed very simple and straightforward. The basic idea was to create a host vector of instances of 'EmbryoGenesis', copy its contents into a device vector and afterwards calculate all individuals forming one generation in parallel.

```cpp
EmbryoGenesis::EmbryoGenesis * TestGenesis =
    new EmbryoGenesis::EmbryoGenesis ( Settings, TestGenome, RandomValues );
vector < EmbryoGenesis::EmbryoGenesis > h_vec (100*sizeof(*Test Genesis));
for ( unsigned int i = 0; i < 100; ++i )
{    h_vec.push_back( *Test Genesis );}
thrust::devicevector < EmbryoGenesis::EmbryoGenesis > d_vec = h_vec;
for ( int j = 0; j < 100; ++j )
{    d_vec[j] -> calculateIndividual ( );}
```

Unfortunately Thrust does not support that kind of device operation and can only handle transformations of device data via its own transform algorithms or derivatives of them. So one solution to obtain a result was to implement a unary function derived from the Thrust library and use the transform algorithm which is provided by Thrust. We tried to tackle this problem using some dummy test classes before we dared to take it one step further:

```cpp
struct test_transformation : public thrust::unary_function<Test, Test>
{
    __device__
    Test operator()( Test a )
    {
        a.seta(5);
        return a;
    }
};

int main()
{
    thrust::host_vector<Test> H (10);
    Test2 testgenome;
    for ( int i = 0; i < H.size(); ++i )
    {    H[i] = Test(testgenome);
        H[i].testMethod();
    }
    thrust::device_vector<Test> D = H;
```
Test init;

test_transformation unary_op;

thrust::transform(D.begin(),D.end(),D.begin(),unary_op);

This however did work but as soon as we tried to go deeper into representing
the hierarchy of our 'EmbryoGenesis' methods where one method calls the next
on a lower level of the hierarchy we faced the problem that we would have to
reimplement the whole project recursively using unary function structs. Con-
sidering the size of the project (>10k lines) this would take another few months
and would equal the effort of a pure CUDA implementation without using the
benefits of the Thrust template library.
4

Conclusion

Thrust offers a nice and easy possibility to implement C++ projects in CUDA. Especially if one is aiming for CUDA parallelisation of the code from the beginning Thrust is a great tool. Due to the structure and quite advanced state of our project a complete reimplementation would have been necessary. To gain more calculation power in our case it seems opportune to go into the - less time consuming - direction of OpenMP or MPI parallelization.
Appendix

5.1 References

1. THENIUS, R. Novel concept of modelling embryology for structuring an artificial neural network, MATHMOD Vienna (2009)


3. Diagrams courtesy of Ronald Thenius

4. The eplsplot library used to plot the embryos was programmed by Olli Niemitalo in 2001 and is freely available under http://www.student.oulu.fi/oniemitalo/DSP/INDEX.HTM