Simulating the Spread of Epidemics in Real-world Trading Networks using OpenCL

Martin Clauß
martin@dev2day.de

University of Leipzig, Diploma programme in Computer Science

Abstract: In this paper we investigate a solution to the problem of simulating the spread of epidemics in real-world trading networks. We developed an application that uses parallel computing devices (e.g. GPUs - Graphical Processing Units) with OpenCL (Open Computing Language). Furthermore, we use the epidemiological SIR-model to represent the nodes of the trading network. Initially, the epidemic grows locally in every node. At certain points of time a transaction happens between several nodes to spread the epidemic spatially. Our results show that a computational speedup of at least 8 times can be achieved using modern GPUs. Additional research is required to further accelerate the computation.

1 Introduction

The epidemic outbreaks of the Swine influenza (N1H1) and the Severe Acute Respiratory Syndrome (SARS) occurred globally and were spread via complex networks such as aviation routes. These problems exceeded the theoretical knowledge and computational power to grasp, predict and defeat epidemics of these large scales. In this work we aim to support and extend well-established simulations of epidemics by increasing the computational performance. We introduce a framework which can handle the task of dealing with complex networks on parallel computing devices. We intend to visualize the spread of an epidemic and to change model parameters in real-time. This requires real-time computations of complex models. This paper investigates a possible solution for this problem using OpenCL (Open Computing Language).

2 Related Work

Peter A. Kolski implemented an application that simulates the trading network mentioned in Section 1. The application uses NetEvo [GdB09], a software library to simulate complex dynamical networks and is used for performance comparison. We also use an OpenCL GPU implementation of the Runge-Kutta 4th order ordinary differential equation (ODE) solver from Böttcher [Bo10] and modified it for our purposes. Detailed information about Runge-Kutta methods can be found in [HNW09].
Table 1: Format of the generated trading data

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Amount</th>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>10002</td>
<td>12</td>
<td>1001</td>
</tr>
<tr>
<td>10033</td>
<td>12233</td>
<td>4</td>
<td>1001</td>
</tr>
<tr>
<td>10231</td>
<td>10333</td>
<td>5</td>
<td>1002</td>
</tr>
<tr>
<td>10002</td>
<td>10000</td>
<td>22</td>
<td>1004</td>
</tr>
</tbody>
</table>

Table 2: Data structure that stores the IDs of active arcs (trading transactions) per day.

<table>
<thead>
<tr>
<th>Day</th>
<th>Active Arcs (IDs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>0</td>
</tr>
<tr>
<td>1002</td>
<td>2</td>
</tr>
<tr>
<td>1004</td>
<td>3</td>
</tr>
</tbody>
</table>

3 Data and Data Structures

The data used for the simulations is generated but has the original structural characteristics of the real-world trading data obtained from the Herkunftssicherungs- und Informationssystem für Tiere [HIT]. In detail, the data consists of approximately 30,000 unique trading partners (or nodes) and approximately 40,000 unique trading connections (or arcs). Furthermore, trading transactions are given per day. Table 1 depicts the data format. The generated trading data exists in the form of plain text files. The inherent graph structure is represented using the LEMON C++ graph library. However, to use the topology information with OpenCL the data must be transformed into linear arrays. Therefore, we used an adjacency list described in [HN07]. Since not every arc is active (in the sense of a trading transaction) every day an additional data structure is introduced, cf. Table 2. In addition, the ODE systems will be stored in a linear array. Each system consists of 3 float or double values that are stored back to back (i.e. $S_1, I_1, R_1, S_2, I_2, R_2, \ldots$, where the subscripts denote the System ID). Table 1 also shows that there exist days without trading transactions and days with multiple trading transactions. Trading transactions in general are modeled as (directed) arcs in a network. Finally, the column Amount provides information about the number of animals that have been transported.

4 Mathematical Model

Within our graph structure, each node will be seen as a subpopulation in which the epidemic will evolve. The arcs represent contacts between nodes and are crucial for spreading the disease spatially. On each node the epidemic will be simulated by the well established SIR model [KM27]. The model splits the population into epidemiological parts of either susceptible ($S$), infectious ($I$) or recovered ($R$) states. The mathematical form is of coupled ordinary differential equations (ODEs), which describe the evolution of each $S - I - R$ state in continuous time with high numerical precision. Figure 1 depicts the mathematical model in the context of a trading network (a graph). In detail, $\frac{dS}{dt}$ describes the rate of change per time of the susceptible, $\frac{dI}{dt}$ of the infectious and $\frac{dR}{dt}$ of the recovered individuals.
OpenCL (Open Computing Language) is a standard for heterogeneous parallel computing developed by the Khronos Group [Khr10]. It aims to support a wide variety of parallel compute devices ranging from mobile phones to high-performance GPUs. An important feature of OpenCL is that it supports cross-platform and cross-vendor development. This has been realized with the introduction of Installable Client Drivers (ICDs). Those contain information about the compilation and the execution of OpenCL programs on a specific device (e.g. a CPU, GPU, DSP, Cell/B.E.). This design lets the programmer write high-level OpenCL C\(^1\) code without knowing the low-level details of the underlying hardware. A major step in an OpenCL program is the execution of the kernel. When a kernel is executed the OpenCL runtime creates a ND range (i.e. an index space to address single threads). On a NVIDIA GPU, for example, those threads are executed in blocks of 32 threads called a warp. All threads within a warp execute the same instructions (called SIMT, Single Instruction Multiple Threads, which refers to a SIMD computer classified by Flynn, see [Fly72]). If divergence occurs within a warp (e.g. if (condition) {...} else {...}) the single branches will be serialized by the hardware [NVI11]. This is an important fact to be aware of during the development of a data-parallel algorithm on a GPU. Another important field for optimizations is the choice of the memory region. The access to private memory takes about 2 cycles, to local memory approximately 20 cycles and to global memory up to 400 - 800 cycles [NVI11]. The memory bandwidth also decreases from private to global memory. Listing 1 shows sequential code to square all elements of a vector and Listing 2 shows a parallel OpenCL version to solve this problem.

\(^1\)OpenCL C is a subset of ANSI-C99 with a couple of additions and restrictions. Additions are, for example, qualifiers that specify the desired memory region that should be used on the compute device (\_global, \_local, \_private). Another addition are built-in functions that allow a thread to identify its position in the N-dimensional range (ND range), for example, get\_global\_id(0). Restrictions are, for example, no recursion, no pointers to pointers, no dynamic memory allocation on the device (i.e. no new or malloc operations).
for (unsigned int i = 0; i < size; i++)
{
    output[i] = input[i] * input[i];
}

Listing 1: Sequential code to square all elements of an input vector and store them in an output vector. size computation steps are necessary to compute the final result.

..kernel void square (..global int *input, ..global int *output)
{
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}

Listing 2: A simple kernel that squares the elements of an input vector and stores the results in an output vector. All results are available after one computation step.

6 Simulation

The simulation consists of a main loop shown in Algorithm 1 and two major function calls described in Algorithms 2 and 3. Figure 2 depicts the overall simulation process. For a certain time all ODEs are solved locally which means that there are no interactions with other nodes. At discrete points of time an interaction may occur. This scenario is depicted in Figure 3. The source nodes’ S, I, R state variables are read and new parameters are computed (write) for the target node. However, the read and write accesses are an implementation specific choice to avoid remote write access (e.g. if one reads from the current node and updates all nodes connected by an outgoing arc, hence there would be numberOfOutgoingArcs write accesses). Since graph structures are irregular in their memory layout (i.e. adjacent nodes in a graph may not reside in the same local memory space) those remote write accesses would be expensive on, for example, a GPU. The occurrence of an interaction depends on the data whether there are trading transactions on that particular day. Our implementation allows us to compute a single integration step for all ODEs of all nodes in parallel. Consecutive integration steps must be executed sequentially due to global synchronization requirements on the GPU.

Algorithm 1 Main simulation loop.

numberOfDaysToSimulate ← 10
currentDay ← 0
while currentDay < numberOfDaysToSimulate or not termination criterion is met do
    simulateLocal() {compute ODEs per node}
    simulateInteraction() {compute trading transactions between nodes}
    currentDay ← currentDay + 1
end while
Algorithm 2 simulateLocal

\[
\begin{align*}
\text{numberOfIntegrationSteps} & \leftarrow 1000 \\
\text{stepSize} & \leftarrow 0.001 \\
t & \leftarrow 0 \quad \text{[time]} \\
\text{for } i = 0 \text{ to } \text{numberOfIntegrationSteps} \text{ do} \\
& \quad \text{executeRungeKutta4Kernel(stepSize, } t) \quad \text{\{Solves the ODEs of all nodes using Runge-Kutta 4th order\}} \\
& \quad t \leftarrow t + \text{stepSize} \\
\text{end for}
\end{align*}
\]

Algorithm 3 simulateInteraction

\begin{itemize}
\item Build the binary array of arc activity for \textit{currentDay}
\item (e.g. 1, 0, 0, 1, 0, …, 0 means arcs 0 and 4 are active)
\item executeInteractionKernel \{The interaction kernel gets called only once\}
\end{itemize}

Algorithm 4 InteractionKernel

\begin{itemize}
\item \{Every node is handled by a single thread\}
\item \text{nodeID} \leftarrow \text{threadID} \\
\item \text{for all} incoming nodes of node \text{nodeID} \text{ do} \\
& \quad \text{if the arc } (\text{incomingNode, nodeID}) \text{ is active on } \text{currentDay} \text{ then} \\
& \quad \quad \text{transport animals and update } S, I, R \text{ of node } \text{nodeID} \\
& \quad \text{end if} \\
\text{end for}
\end{itemize}

Figure 2: Simulation per node.

Figure 3: A trading transaction (i.e. animal transport) between two nodes.
7 Results

It is important to mention that we only computed the local ODEs without interactions. The interaction kernel has not been completed before the submission deadline of this paper. All programs being tested have been compiled with g++ 4.5 and with an optimization level of -O3. To provide a fair sequential result we used the fastest CPU available. The results are shown in Table 3. Since the NetEvo application uses double precision, we used double precision for the GPUs as well. This is a problem since consumer GPUs do not have full double precision support. Full support is only available for high-end GPUs such as NVIDIA’s Tesla series. Consequently, the speedup would be even higher.

<table>
<thead>
<tr>
<th></th>
<th>sequential</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intel Core i7 950 @ 3.07GHz</td>
<td>Quadro 1000M</td>
</tr>
<tr>
<td>Execution time (ms)</td>
<td>13128</td>
<td>8399</td>
</tr>
<tr>
<td>Speedup</td>
<td>1</td>
<td>1.56</td>
</tr>
</tbody>
</table>

Table 3: Performance comparison using double precision.

8 Future Work

A first step in the future will be the completion of the program’s functionality so that the epidemic can spread between nodes. Furthermore, we aim to optimize the kernel code for execution on GPUs. This includes memory access optimizations and the reduction of control flow divergence with the kernels. A long-term goal is to visualize the network and interact with it in real-time. Moreover, an integration of OpenCL into NetEvo is a further idea to realize. Finally, there exists the possibility to generate hardware in the form of a FPGA based on OpenCL code [HGB11]. This could result in a tremendous speedup for our application.

9 Acknowledgement

I would like to thank my thesis advisors Dipl.-Inf. Jörn Hoffman and Dipl.-Phys. Peter A. Kolski for their support. Furthermore, I would like to thank the reviewers for their constructive input.
Simulating the Spread of Epidemics in Real-world Trading Networks using OpenCL

References


