# SAMPO: An Agent-based Mosquito Point Model in OpenCL

**Klaus Kofler**
**DPS Group, Institute for Computer Science**
**University of Innsburck, Austria**
**klaus@dps.uibk.ac.at**

**Gregory Davis**
**Center for Research Computing**
**University of Notre Dame, IN, USA**
**gdavis2@nd.edu**

**Sandra Gesing**
**Center for Research Computing**
**University of Notre Dame, IN, USA**
**sandra.gesing@nd.edu**

**Keywords:** agent-based modelling, OpenCL, GPGPU

## Abstract

Agent-based modeling and simulations are applied for problems where population-level patterns arise from the interaction of many autonomous individuals. These problems are compute-intensive and excellent candidates for the use of parallel algorithms and architectures. As a cross-platform software development framework for parallel architectures, OpenCL appears as an ideal tool to implement such algorithms. However, OpenCL does not natively support object-oriented development, which most of the toolkits and frameworks used to build agent-based models require.

The present work describes an OpenCL implementation of an existing agent-based model, simulating populations of the *Anopheles gambiae* mosquito, one of the most important vectors of malaria in Africa. Discussed are the methods and techniques used to overcome the design challenges, which arise when transitioning from an object-oriented program to an efficient OpenCL implementation. In particular, the parallelism inside the program has been maximized, dynamic divergent branching was reduced, and the number of data transfers between the OpenCL host and device has been minimized as far as possible.

Even though our implementation was designed for this specific use case, the approach can be generalized to other contexts, as most agent-based point models would benefit from the same basic design decisions that we took for our implementation. Comparisons between the object-oriented and the OpenCL implementation illustrate that using an OpenCL approach offers two important performance benefits: an overall simulation time speedup of up to 576 with no measurable loss of accuracy, and better scalability as the agent-population size increases. The tradeoffs necessary to achieve these performance benefits and the implications for future agent-based software development frameworks are discussed.

## 1. INTRODUCTION

Malaria is a vector-borne illness caused by parasites that are transmitted from human to human through an intermediate organism, mosquitoes. Because malaria cannot be transmitted between humans in the absence of these vectors, malaria control and eradication strategies have primarily relied on interventions that directly target the vector, such as insecticide-treated nets, larvicides, and indoor residual spraying. To maximize the effectiveness of campaigns involving these types of interventions, knowledge of population-level malaria transmission dynamics must be understood. Formal modeling and simulation of mosquito populations is an attractive technique for researchers and malaria control managers to understand malaria transmission and assess hypothetical strategies for deploying limited intervention resources. Agent-based modeling and simulation (ABMS) is becoming a popular approach used in this field [1, 2, 3] because each member of the population (i.e. each mosquito) is modeled individually with simple rules, typically derived from laboratory and field research, dictating their behavior. By simulating large populations of these agents and their interaction with each other and their environment, system-level properties emerge, which can be used to better understand the dynamics of malaria transmission.

To facilitate the use of ABMS across many fields of interest, a diverse group frameworks and tools have been developed. For example, FLAME (Flexible Large-scale Agent Modeling Environment) [4] is a generic agent-based modeling system where agents are formalized as Stream X-Machines specified in XML markup and compiled into executable code through a template system alleviating the need to develop custom code for common agent-based modeling tasks. Other libraries such as Repast [5] and MASON (Multi-Agent Simulator of Networks) [6] provide common ABMS foundation objects that can be sub-classed and extended to implement model-specific behavior. Beyond these generic frameworks, there are tools specifically for modeling populations of disease vectors such as DTK (Disease Transmission Kernel) [7] and AGiLESim [3].

Agent-based simulations can be compute-intensive, especially when there is a need to simulate large numbers of agents to approximate a given population. Furthermore, since the outcomes of these simulations typically depend on stochastic factors, the simulations are often rerun several times to more accurately characterize the model results, extending computing demands even more. Importantly, the agent-based approach is also inherently parallel as the behav-

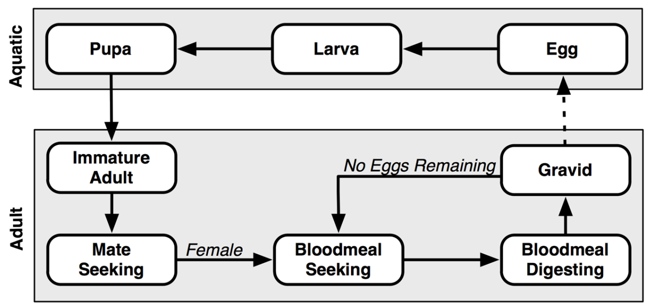

**Figure 1.** Mosquito life cycle.

ior of one agent is not generally directly dependent on the behavior of another. This is reflected in the fact that each of the aforementioned frameworks provide mechanisms for parallelizing the simulations they implement using either a distributed architecture with nodes communicating using MPI or taking advantage of multi-core CPUs using OpenMP and/or multithreading. However, none of these frameworks takes advantage of GPU-based computing to automatically scale with the available resources of the machine the simulation is executing on except for FLAME GPU [8]. This extension of FLAME is implemented in CUDA so that it runs on NVIDIA GPUs, but is not compatible with GPUs from other vendors.

In the present work we developed SAMPO (Scalable Agent-based Mosquito POint model), a derivative model of the Java version of AGiLESim, implemented using OpenCL [9] where each mosquito agent follows the lifecycle shown in Figure 1. As explained in [3], the model is well-suited for simulating the evolution of mosquito populations. We chose OpenCL because it supports a large variety of processors. OpenCL programs not only run on GPUs of several vendors (e.g. from AMD and NVIDIA), but can also be executed on most multi-core CPUs as well as various accelerators (e.g. Intel Xeon Phi) and embedded processors (e.g. ARM CPUs). Whereas HPC resources are expensive and not necessarily accessible to users of agent-based simulations, GPUs and multi-core CPUs are widely available even in workstations.

The remainder of this paper is structured as follows: Section 2. introduces the implementation details of SAMPO including the data model and the design of the agents. Section 3. illustrates the correctness of the model relative to AGiLESim and Section 4. presents performance comparisons of SAMPO on various hardware to the Java version of AGiLESim [3]. We conclude with a summary and our vision as to how the agent-based modeling of SAMPO can be further developed.

## 2. IMPLEMENTATION

Our simulation is implemented in *OpenCL* [9], using the LibWater library [10] to reduce the implementation effort. OpenCL is a language based on c99 which some extensions, mainly to express parallelism in the program. It is based on a *host/device architecture* [9]. The host code is executed by the CPU. Its main responsibilities are the distribution and collection of data to/from the devices. The parallel workload of the program is executed by the device. The functions executed by the device are called *kernel functions*. A device can be a GPU, some other form of accelerator attached to the system, or the CPU which acts as a host itself. The device is not capable of dynamically changing its allocated memory or transferring data from/to the host. These tasks are performed by the host.

Our implementation consists of twelve kernels which are called in every iteration of the simulation. In general, it is beneficial for OpenCL programs, to have as many independent threads running at a time as possible. In order to achieve that, we create one thread for each agent for most of our kernels. The control flow of our implementation, showing the discrete operations performed during the simulation, can be seen in Figure 2.

Although the program may run on many different processors, it is optimized for GPUs with dedicated memory that are connected to the host CPU via PCIe. For such architectures, the connection bandwidth between the host and the device is often a bottleneck. Therefore, this implementation aims at minimizing the communication between host and device. All agents are generated by the device and will never be copied to host memory. The only data that has to be copied to host memory are the snapshots of the environment (containing the number of mosquitoes in each state as well as the number of total and infective bites that occurred) after each iteration. The only data that has to be written to the device are the seeds for the random number generation (see Section 2.3. for more details). The following paragraphs describe the most important design decisions that we made during our implementation. For more detailed information, we refer to the source code which is publicly available on `https://github.com/klois/SAMPO`.

### 2.1. Data Layout

The mapping of data to memory has a big impact on the performance, especially in GPU-based architectures. Because GPUs do not possess features such as sophisticated caching hierarchies similar to traditional CPUs, the data layout and mapping is crucial. This has a number of important implications for design decisions in porting AGiLESim to OpenCL and the most important of these are described below.

#### 2.1.1. Agent Representation

Each agent (i.e. mosquito) in our simulation consists of twelve fields, listed in Table 1. The most natural way to store the agents of this simulation would be an array of structures where each structure represents one agent. This solution however, leads to a lot of unnecessary load and store opera-

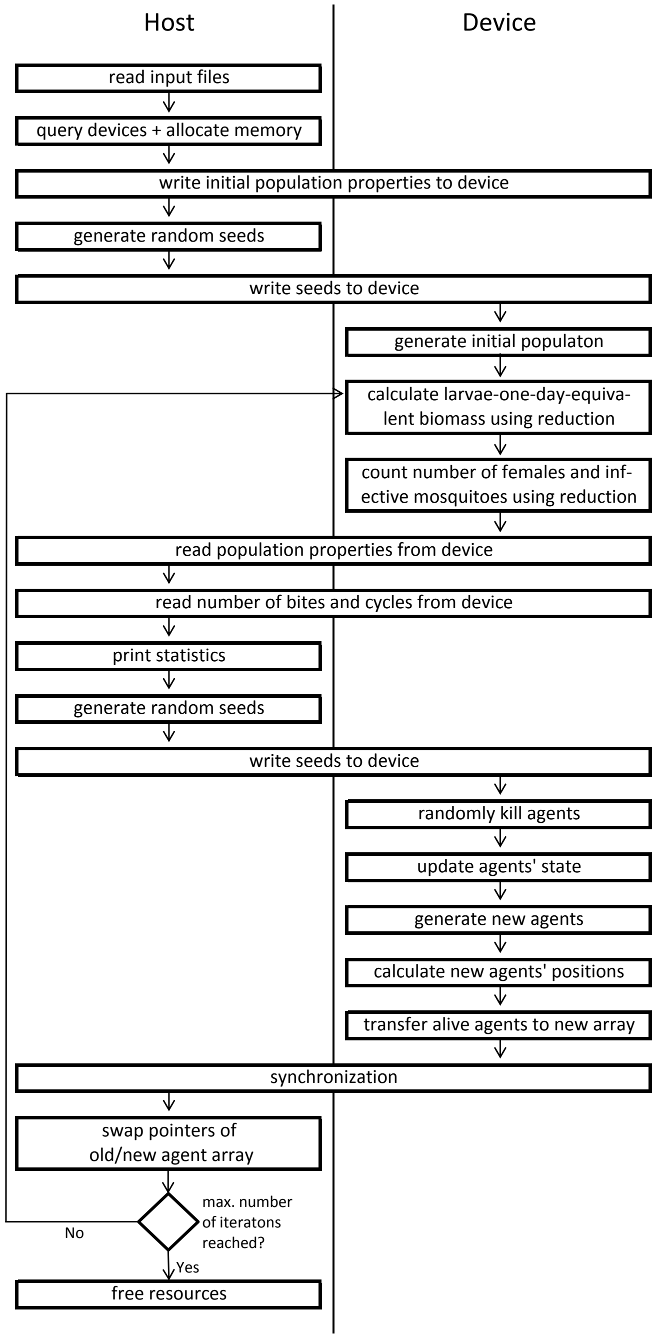
Host
Device
read input files
query devices + allocate memory
write initial population properties to device
generate random seeds
write seeds to device
generate initial populaton
calculate larvae-one-day-equiva-lent biomass using reduction
count number of females and inf-ective mosquitoes using reduction
read population properties from device
read number of bites and cycles from device
print statistics
generate random seeds
write seeds to device
randomly kill agents
update agents' state
generate new agents
calculate new agents' positions
transfer alive agents to new array
synchronization
swap pointers of old/new agent array
max. number of iteratons reached?
No
Yes
free resources

**Figure 2.** Control flow of SAMPO. All tasks executed on the host are of constant time complexity with regard to the population size. Thus, the execution time of simulations with larger populations is dominated by the tasks executed on the device.

tions (when the data of an entire agent is loaded while only few fields of the structure are required) and/or non-coalesced memory accesses [11] (when only a single field of the agent is loaded). Therefore, the transformation of the array of structures into a structure of arrays could be considered. However, since structures containing pointers are not supported in OpenCL, this would lead to a separate array for each field of the agent structure, resulting in a large number of arguments for each kernel. Maintenance and readability of the code would heavily suffer from that transformation. Moreover, if memory accesses occur in a non-coalesced fashion, it can be beneficial to load a single struct instead of multiple scalars to minimize non-coalesced memory accesses.

In order to exploit the advantages of both previously mentioned techniques, we use a hybrid approach for our data layout. The agent structure is split into three smaller ones, called A, B and C in Table 1. The layout of the structures is chosen in a way so that each kernel function that has to load one of those structures uses as much of its fields as possible.

**Table 1.** Data structure representing an agent, i.e. one mosquito. The agent's fields are distributed over three different structures, the structure of each field is indicated in column *Struct*.

| Name | Type | Struct |
|---|---|---|
| Available Eggs | UINT | A |
| Gonotrophic Cycle Length | UINT | A |
| Human Bloodmeal Count | UINT | A |
| Delay to State Transition | REAL | A |
| Number of Egg Batches | UINT | A |
| Ovi Positioning Attempts | UINT | A |
| Is Dead | BOOL | B |
| Current State | ENUM | B |
| Age in Hours | REAL | C |
| Hours in State | REAL | C |
| Is Female | BOOL | C |
| Cumulative Sporgonic Development | REAL | C |

### 2.1.2. Agent Storing

In order to maximize the performance, all arrays have a fixed size and are allocated at the start of the simulation. For arrays of constant size as well as for arrays whose size depends on the simulation length (i.e. the number of iterations) the size can be calculated accurately at the beginning of the simulation. For arrays whose size depends on the number of active agents, it is more difficult, since the maximum number of agents that will be reached during the simulation is not known at the beginning. In our program, the user has to estimate the maximum number of agents that can be reached during the simulation. This number will then be used to allocate memory for the agents. In our experiments we figured out, that 10$\times$ the maximum carrying capacity of the environment is a well-suited estimation. The total memory consumption on the OpenCL host and device is shown in Table 2.

Since our implementation aims at minimizing the communication between host and device, the agents are generated

**Table 2.** Memory used and transferred during the simulation

| | Size in Bytes |
|---|---|
| host[a] | $240 + 24 * \#iterations$[b] |
| device[a] | $240 + 116 * \#agents$[c] |
| host to device[d] | 64 |
| device to host[d] | 100 |

[a] total amount of allocated memory, constant during the entire simulation
[b] number of time-steps in the simulation
[c] maximum number of agents specified as described in Section 2.1.2.
[d] data transferred in each iteration

and stored only on the device. They will never be transferred to the host during the entire simulation. The only data that will be transferred to the host are the statistics that are of interest to the user of the simulation as described in Section 2.4..

## 2.2. Divergent dynamic branching minimization

In OpenCL, threads are organized in a two-level hierarchy. The entirety of threads is subdivided into `work_groups`, which consist of several threads. The size of the `work_groups` is defined by the `local_work_size` [9]. To maximize the performance on GPUs, all threads in one `work_group` should follow the same path in the control flow, i.e. should execute the same code. When branches cause threads inside a `work_group` to execute different code, it is called *divergent dynamic branching* [12]. Since our main target architecture for this program are GPUs, minimizing divergent dynamic branching [12] is crucial to obtain a good performance.

In our implementation, consecutive agents in memory are mapped to consecutive threads. Therefore, agents who are likely to execute the same code should be packed together in memory in order to minimize dynamic branching. To achieve this, our implementation blocks agents that are in the same state. At the end of each block, padding to the next multiple of the `local_work_size` is added in order to avoid agents belonging to two different states inside a single `work_group`. While this approach is effective in reducing the number of divergent branches to a minimum, it requires a reordering of the agents after each iteration. During this reordering, dead agents are removed from the array, agents that changed their state are moved to the block of their new state and the entire array is packed in order to avoid empty slots within the blocks of states.

Doing this reordering in parallel requires duplicate arrays that hold agent information. In each iteration, all living agents are moved from the currently used array to the other one, doing the previously described reordering at the same time. Duplicating the agent arrays means, that the memory requirement on the device almost doubles. While this is reducing the maximal population size that can be handled by a device, this is the only way we found to make the simulation efficient on modern GPUs. Since this copy is done in parallel with one separate thread for each agent, the copying thread does not know if the surrounding agents have died or advanced to another state. Because there is no efficient way to determine the new index of its agent during the copy, we calculate the new index of each agent before the copy operation using several prefix sums.

The indices are calculated separately for each mosquito state. To do so, we generate a temporal *data array* for each state. It has the same size as the agent array and is filled with 1 on the position of alive agents of the corresponding state and 0 on all other ones. The result of a prefix sum on this array will be the new index for each agent in that state, relative to the beginning of the state's block. Looking at the mosquito development cycle depicted in Figure 1, it is obvious, that agents of each state can only be found in a restricted area in the agent array. For example, all agents who are in larval state in the next generation, must be in either egg or larval state in the current iteration. Therefore, the prefix sum can be restricted to that area. To avoid having a separate prefix sum for each state, we use a *segmented prefix sum*.

The segmented prefix sum differs from a "standard" prefix sum as it restarts counting at 0 at the beginning of each segment. In our application we create one segment for each state. Each state's segment is set to the range where agents in that state can occur, as described in the previous paragraph. Those ranges overlap, as shown in Figure 3. Most segments span two state blocks, the one which's index is calculated and the previous one in the mosquito development cycle. The segment for the egg state covers only its own block, since there is no previous state. New eggs are simply put at the beginning of the new array. The segment of the blood meal seeking state covers four blocks. Other than mate seeking mosquitoes that develop to this state after finding a mate and blood meal seeking mosquitoes, also gravid mosquitoes go to blood meal seeking state after all their eggs have been laid. The segment also covers blood meal digesting mosquitoes although no agent in this block can become blood meal seeking in the next iteration. However, it is added to the segment, since all segments have to be contiguous in the used prefix sum implementation. Due to the overlapping of the segments, the new index of all agents cannot be calculated using a single segmented prefix sum, but we have to perform three of them. Figure 3 elucidates, which sates are handled by each of the three prefix sum.

To be able to calculate the new indices of the agents using the segmented prefix sum, two temporary arrays for each of the three prefix sums have to be created: The first is the *data array* consisting of 1 and 0 as described earlier. The data arrays of the states covered by the same prefix sum can easily be merged, since their segments do not overlap. The second is a *flag array*, determining the single segments. The result of the prefix sum holds the new index of each agent, relative to the start of its state's block. The start of the block can easily be determined by summing up the number of agents in all preceding states, which is equal to the last element of the corresponding state's prefix sum $+1$. On the GPU we use the parallel prefix sum implementation of Bolt [13]. When the program is executed on a CPU, we use a sequential implementation of the prefix sum, since it delivers higher performance on that kind of processors. While the sequential implementation of the prefix sum uses only a single kernel, the calculation of the prefix sum in parallel consists of four kernel invocations: one is generating the previously described temporal arrays, while the other three, taken from [13], are performing the actual prefix sums.

## 2.3. Random number generation

Random number generation is a crucial component of many agent-based simulations. Many parameters of our implementation (and similarly in AGiLESim) are influenced by randomness (e.g. mortality rate, delay to develop for some states, number of eggs generated, etc.). This means that many random numbers are needed in each iteration. However, GPUs do not have an available built-in pseudo-random number generator like CPUs, and OpenCL does not provide any means of generating random numbers on the device. Since this implementation aims at minimizing the communication between host and device, a common approach for random number generation involving generating all random numbers on the host CPU and copying all of them to the device is not an option. Therefore, we use the a hybrid Tausworthe [14, 15]/LCG [16] generator as described in [17]. This algorithm uses four integer random numbers as seeds to generate a single floating point random number. Each of the four seeds is used as an argument for either the Tausworthe or the LCG algorithm with varying parameters. The four resulting floating point numbers are then combined to form a single random number. The main advantage of using four seeds instead of one is the increased period length, equal to approximately $2^{121}$ for the used implementation.

The hybrid Tausworthe/LCG generator produces uniformly distributed random numbers. However, for some variables in the simulation, we need random numbers following a Gaussian distribution (e.g. the number of eggs generated by a mosquito). To produce normally distributed numbers we use the Box-Muller [18] transform which takes two independent, uniformly distributed random numbers as an input and returns a single random number with a Gaussian distribution. The implementation proposed in [17] produces two normally-distributed random numbers at a time; however, we generate only one because a second random value is not useful at the point of generation.

The approach in [17] also proposes a separate set of seeds for each thread, where the current results of the Tausworthe/LCG steps are used as a seed for the next one. This requires a considerable amount of memory (equal to the number of agents) to store those seeds in addition to one read and one write operation to global memory every time a random number is generated. To avoid this overhead, we use an approach, similar to the *One-PRNG-per-kernel-call-per-thread* approach described in [19]: all threads use the same seed, which is mangled using the global id of the thread. Instead of incorporating the simulation time step in the mangling function, we generate a new set of seeds in every iteration on the host CPU, and transfer them to the device. Doing so, allows us to replace the hashing function to mangle the seed used in [19] with a single addition or multiplication operation on the individual seeds. In comparison to the approach proposed in [17], this approach allows SAMPO to handle larger populations due to the reduced memory overhead. Furthermore, our approach does not require write access to the device memory and all threads read the seeds from the same memory location, which means the read operations on the seed can be cached on most modern GPU architectures.

In our implementation, each thread needs to generate up to four random numbers per iteration. Therefore, the host writes fours sets of seeds to the device in each iteration, which each thread combines with its global id to generate four independent random numbers. Using the previously described method, the number of seeds generated and transferred from the host to the device in each iteration is constant and independent of the number of agents in the simulation. Due to the small amount of data to be transferred, the time needed for the data transfer can be neglected on most the the architectures used in our tests (see Table 3). A considerable overhead could be measured only on the AMD Radeon HD5870; however its impact is constant and does not scale with agent population size.

## 2.4. Simulation Output

The simulation output consists many measures of the state of the simulation at each time step including: number of mosquitoes in each development state, an age-adjusted larval biomass (a measure of how saturated the aquatic environment is - affecting both larval mortality and the number of eggs laid by a given mosquito), the number of female adult mosquitoes and how many of them are infective, the number of mosquito bites that have occurred (both infective and non-infective), the number of gonotrophic cycles[1] that have been completed, and the sum duration for all completed gonotrophic cycles.

[1]A gonotrophic cycle is completed when a gravid mosquito placed all its eggs and goes back into blood meal seeking state

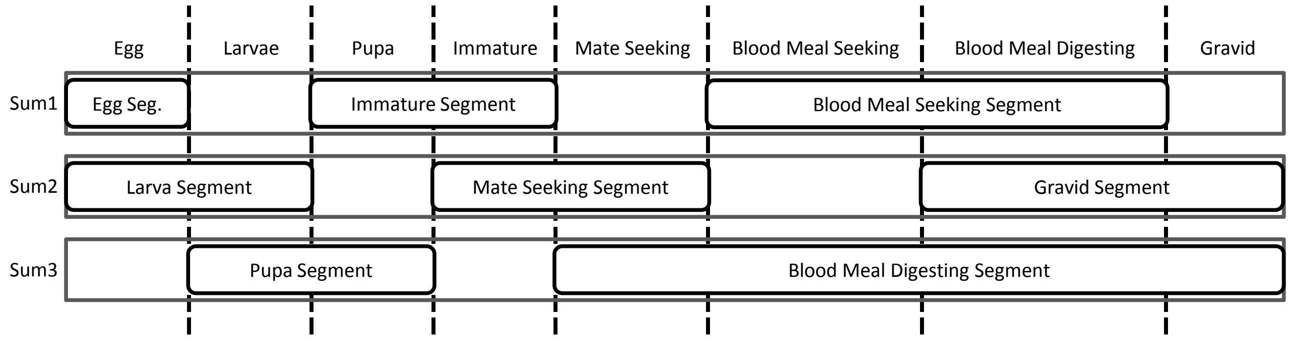

**Figure 3.** Segments for the three prefix sum calculations. Each prefix sum calculations considers all agents from the start of the first segment to the end of the last one.

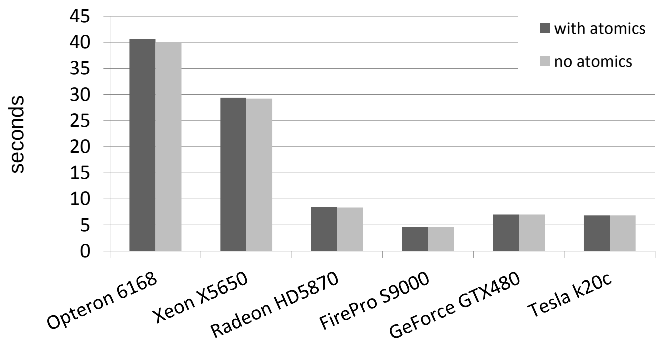

**Figure 4.** Execution time of the agent-state updating kernel of different processors with and without atomic operations for statistical counters. Times are taken from a one year simulation of a population with maximal 1.4 million agents.

To determine the number of agents in each state, the starting and end index of each state-block (as described in Section 2.1.2.) is copied to the host. The host will then use those values to calculate the actual numbers.

To calculate the number of bites as well as the number of gonotrophic cycles and the sum of their length we use *atomic operations* [9]. This means that there is a global counter for each of those values which is increased by each thread that handles the corresponding action (e.g. a bite occurred or a mosquito laid all is eggs). Although atomic operations are a potential performance bottleneck, especially on GPUs, in this simulation their effect is negligible, since they are not invoked that frequently. Figure 4 shows the impact of atomic operations on the execution time, comparing the run-time of the kernel updating the agent's state, with and without atomic operations. By omitting the atomic operations, some of results generated are no longer accurate, but the condition of the the simulation itself is not affected. The comparison clearly shows that the impact of the atomic operations on the execution time is minimal.

There are three numbers in the statistics which require performing a reduction over a large number of agents: The age-adjusted larval biomass, which requires a reduction over all larvae, the number of adult female/male, and potentially infective mosquitoes, which require a reduction over all adult mosquitoes. Since these numbers are affected by all agents in the corresponding states, summing then up using atomic operations would not be an efficient solution. Therefore, these numbers are calculated at the beginning of each iteration using a *parallel reduction in local memory*, based on the two-stage implementation in [20]. Since the counting of female and potentially infective mosquitoes is performed on the same data, we use only one parallel reduction to do both of them at once. The parallel reductions are executed in two steps:

1. Reducing the input array to `local_work_size` [9] elements using `local_work_size`$^2$ threads.
2. Using `local_work_size` threads to calculate the final result, based on the results of step 1.

This two-step approach is the only way to distribute the reduction to more than `local_work_size` threads since OpenCL does not provide the possibility to synchronize over several `work_groups` [9].

## 3. CORRECTNESS

To verify the correctness of our implementation we compared the generation of random numbers and the structure of the agent population –number of adult mosquitoes and their age structure (see [21])—to these metrics from a similarly configured Java version of AGiLESim.

### 3.1. Random number generator

To verify our approach to generate random numbers, described in Section 2.3., we compare the number of eggs produced by each mosquito in a one year simulation with a maximum of approximately $40,000$ agents. For the mosquitoes that we are simulating, the number of eggs is calculated based on a normal distributed random number as shown in Algorithm 1. To spawn these numbers, SAMPO uses the Box-Muller transformation described in Section 2.3., while the

**Algorithm 1** Formula calculating the number of eggs produced for a single mosquito developing from blood meal digesting into gravid state. *numEggBatches* is the count how often a specific mosquito already spawned new eggs.

```
function GENERATEEGGS(int:numEggBatches)
    $eggBatchSize \leftarrow 170$
    $stdDev \leftarrow 30$
    $nEggs \leftarrow \lfloor \text{RANDNORMAL}(eggBatchSize, stdDev) \rfloor$
    if $eggs < 0$ then
        $eggs \leftarrow 0$
    end if
    $eggs \leftarrow \text{ROUNDTONEAREST}(eggs \cdot 0.8^{numEggBatches})$
    return $eggs$
end function
```

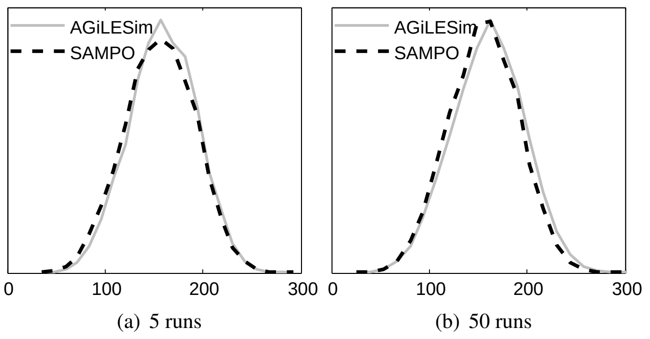

**Figure 5.** Histogram of the number of eggs generated per mosquito for SAMPO and AGiLESim. The left picture shows the histogram for five runs, the right picture for 50 runs. Each run simulates one year with a maximal population size of approximately $40,000$ mosquitoes.

used version of AGiLESim uses the random number generator provided by RepastJ [22]. Figure 5 shows the histograms of the number of eggs created by the individual mosquitoes. It can be seen, that the histogram curves are very similar, especially with a high number of runs.

## 3.2. Population comparison

In order to show that SAMPO produces correct populations we perform two validity checks on the produced population of adult mosquitoes. The first one compares the average number of adult mosquitoes on each day of a one-year-long simulation to the ones produced by the Java version of AGiLESim. Figure 6 shows how the number of adult mosquitoes evolves during the simulation. The picture clearly elucidates that the populations generated by the two different implementations can be considered as equal.

The second validation compares the ages of all adult agents at the end of a one-year-long simulation with the hypothetical age structure calculated using the formula presented in [21].

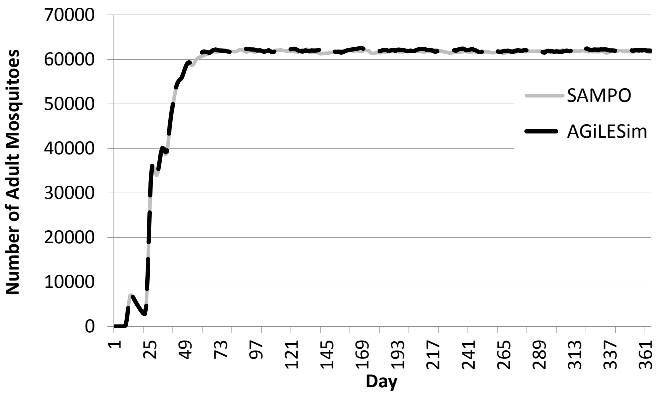

**Figure 6.** Average number of adult agents on each day of a one-year-long simulation, created with SAMPO and AGiLESim, respectively. The maximum population size for this experiment was approximately 1.4 million agents, the environment temperature was a constant 30°C.

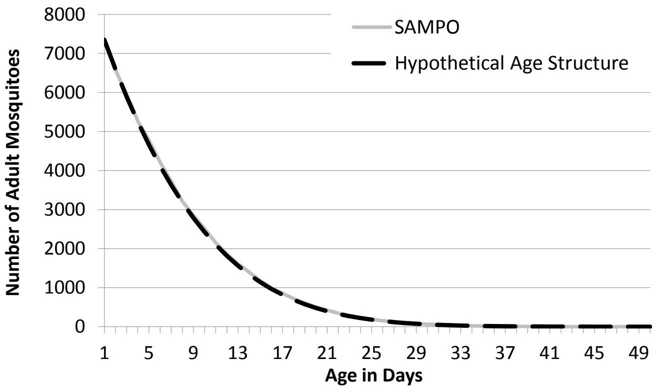

**Figure 7.** Histogram showing the age in days of the adult mosquitoes at the end of a one-year-long simulation as well as the hypothetical age structure calculated using the formula presented in [21]. The maximum population size for this experiment was approximately 1.4 million agents, the environment temperature was a constant 30°C.

Figure 7 shows a histogram of the adult mosquitoes' age distribution as well as the curve calculated based on the daily mortality rate of agents, as described in [21]. Also in this case, the values produced by SAMPO match the expected ones, thus we consider it as correct.

## 4. PERFORMANCE

The main goal of implementing this mosquito simulation in OpenCL was to reduce the simulation time using modern accelerator hardware such as GPUs. In the following paragraphs we will demonstrate the performance of our implementation in comparison to the Java version of AGiLESim [21] as well as it scalability with increasing population sizes on various processors. The population size is controlled by adapting the

**Table 3.** OpenCL run-time-compilation time and random seeds generation and copy time for the used processors.

| | AMD Opteron 6168 | Intel Xeon X5650 | AMD Radeon HD5870 | AMD FirePro S9000 | NVIDIA GeForce GTX 480 | NVIDIA Tesla k20c |
|---|---|---|---|---|---|---|
| Compilaton Time | | | | | | |
| Uncached (s) | 1.10 | 2.03 | 5.23 | 1.85 | 5.83 | 9.93 |
| Cached[a](s) | N/A | N/A | N/A | N/A | 0.14 | 0.14 |
| Random Seeds Generation and Copy (s) | 0.34 | 0.29 | 4.59 | 1.70 | 0.19 | 0.24 |

[a] The NVIDIA OpenCL run-time compiler automatically caches compiled kernels in the user's home directory. If a cached kernel is executed, it can be directly loaded form the cache without recompilation resulting in much less overhead.

**Table 4.** Used hardware

| | AMD Opteron 6168 | Intel Xeon X5650 | AMD Radeon HD5870 | AMD FirePro S9000 | NVIDIA GeForce GTX 480 | NVIDIA Tesla k20c |
|---|---|---|---|---|---|---|
| # Processors | 2 | 2 | 1 | 1 | 1 | 1 |
| Frequency (MHz) | 1900 | 2670 | 850 | 900 | 1401 | 706 |
| Compute Units | 24 | 24 | 20 | 28 | 15 | 13 |
| # Parallel Ops | 96 | 48 | 1600 | 1792 | 480 | 2496 |
| FLOPS (SP) | 365 | 256 | 2720 | 3225 | 1345 | 3524 |
| Memory (GB) | 32 | 24 | 1 | 6 | 1.5 | 5 |
| Memory BW (GB/s) | 83 | 62 | 153 | 264 | 177 | 208 |

initial number of eggs put into the system, as well as the carrying capacity [21]. For all experiments, the carrying capacity is constant throughout the entire simulation and set to 5$\times$ the initial number of eggs. We chose a simulation length of one year with a resolution of one hour, leading to 8760 simulation steps. The temperature was constant at 30°C.

Figure 8 shows the execution times using different population sizes of the AGiLESim compared to SAMPO. The execution times of the Java implementation of AGiLESim were measured on a dual socket Intel Xeon X5650 (see Table 4 for more details). It can clearly be seen, that SAMPO's OpenCL implementation scales much better with increasing problem sizes than the Java version. While the smallest problem size with a maximal population size of approximately 40 thousand agents is almost equally fast in both implementations, for the biggest tested problem size, with a maximal population size of approximately 5.5 million agents, the OpenCL version outperforms the Java implementation by a factor of 46. There are several reasons for that:

- The AGiLESim uses double precision for all floating point numbers in the simulation. SAMPO uses only single precision. However, as shown in Section 3., this has no negative effect on the result of the simulation.
- While the Java version of AGiLESim is sequential, the OpenCL implementation is parallel and uses all cores available in the system.
- The Java version of AGiLESim has a considerably lower constant overhead than our implementation using OpenCL. Executing all 8760 simulation steps without any actual calculation (but including all necessary data transfers for the OpenCL implementation), the Java version takes 3.5 seconds, while the OpenCL implementation needs 24.2 seconds to execute this task.
- Java has an automated memory management. The memory allocation is hidden from the programmer. Occasionally, a garbage collector is invoked by the Java VM, which frees all unused memory. The bigger the population, the more often the garbage collector has to be invoked in order to limit the maximum memory requirements. Since one invocation of the garbage collector consumes linear time in relation to the population size and it is invoked more often with increasing population sizes, this process consumes a considerable amount of time of the Java version when simulating large populations.
- Many agents in AGiLESim are stored in Java `ArrayLists`. Dead agents are eliminated from those lists using the `remove` operation. Since `ArrayLists` internally store data in an array, this operation requires moving all elements positioned after the eliminated element. This results in an average of $\frac{n}{2}$ data movements for each remove operation, leading to above linear time complexity.

We analyze the performance and scalability of the OpenCL implementation on the processors described in Table 4, including CPUs and GPUs. Figure 8 shows the performance of those processors with varying population sizes. The numbers shown in Figure 8 represent the execution time of the

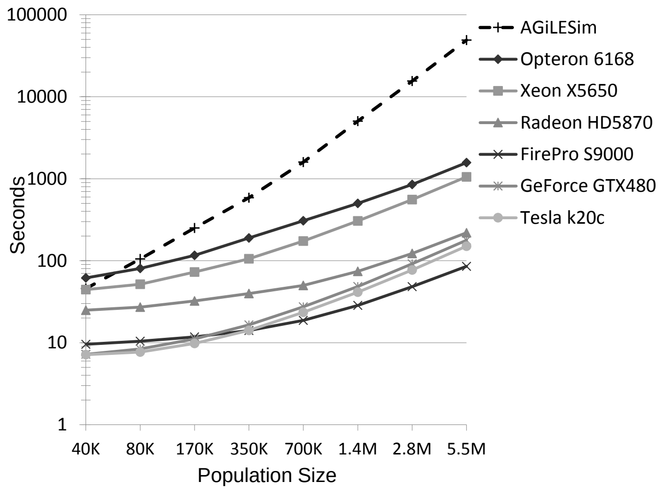

**Figure 8.** Execution time with varying population sizes for the Java implementation used in AGiLESim [21] and our OpenCL implementation on various processors. The execution times of AGiLESim were measured on an Intel Xeon X5650.

simulation only, excluding the time needed for the OpenCL run-time compilation. The compilation times for the various devices can be found in Table 3.

The performance measurements clearly show that GPUs suit this simulation better than CPUs. A NVIDIA Tesla k20c achieves a speedup of 6-7 and 8-11 compared to the tested Intel and AMD dual socket CPUs, respectively. The CPUs and the NVIDIA GPUs scale almost linear with the population size of the simulation. On the AMD GPUs, however, the execution time raises less than linear with increasing problem sizes. The downside of the AMD GPUs is that they show relative long execution times for small population sizes. According to our measurements, just invoking all the kernels and data transfers needed for the simulation without doing any actual calculations takes about 17 and 9 seconds on the Radeon HD5870 and FirePro S9000, respectively. In contrast, both NVIDIA GPUs need only 5 seconds for this task. The high overhead on the AMD GPUs seems to be related to the high kernel invocation overhead on such hardware [12]. The inefficient use of resources on the AMD GPUs is also underlined by the low GPU usage. Depending on the population size, we observed a GPU load of $35-95\%$ and $38-94\%$ on the Radeon HD5870 and FirePro S9000, respectively. On the Tesla k20c the GPU utilization ranges from 80 to 99% while NVIDIA does not provide any tool to remotely observe the GPU utilization on the GeForce parts. The overall performance of the NVIDIA Tesla k20c is disappointing. Despite the report in [23] we observed a maximal speedup of 1.19 over the previous generation part for our simulation. Furthermore, when simulating large populations, the Tesla k20c is outpaced by the FirePro S9000, despite its bigger overhead and lower GPU usage.

## 5. CONCLUSION AND FUTURE WORK

In this work we implemented SAMPO, a parallel OpenCL version of an existing agent based point model for simulating populations of the *Anopheles gambiae* mosquito. We described some of the more important challenges that arise when converting an ABMS to the parallel architecture necessary for OpenCL and how we addressed them. Furthermore, we demonstrated correctness of our implementation by comparing the output to the one generated by AGiLESim. The theoretical age structure of the population is verified similarly to the verification applied for the original AGiLESim model [21].

The main goal of this work was to investigate the use of OpenCL as a means of minimizing simulation time for large scale ABMS executions. Our implementation is very effective for larger population sizes, achieving a speedup of up to 46 over the Java implementation of AGiLESim when running on the same multi-core CPU. In some respects this same-CPU comparison illustrates the overhead of using nonnative code for simulation. Furthermore, we might expect the OpenCL version, running on the CPU, to have similar performance characteristics to a highly-optimized C version. Running the simulation on modern GPUs, we observed that the performance improved approximately 12 times compared to the same code running on the CPU, and a total speedup of 576 over the original Java implementation. These findings indicate that OpenCL is an attractive environment for building compute-intensive agent-based simulations.

In the future, we expect to extend the functionality of our simulation incorporating features to simulate the effects of various vector-targeted interventions like insecticide-treated mosquito nets and indoor residual spraying (IRS) in order to observe their impact on the malaria transmission cycle. SAMPO was designed from the beginning to be able to support this kind of interventions. Furthermore, we plan to extend our simulation to different mosquito species. In our OpenCL implementation, the species specific behavior is encapsulated in a single header file, which can easily be exchanged with one characterizing another species.

## REFERENCES

[1] C. Isidoro, N. Fachada, F. Barata, and A. Rosa, "Agent-Based Model of Aedes aegypti Population Dynamics," in *Progress in Artificial Intelligence, 14th Portuguese Conference on Artificial Intelligence, EPIA 2009, Aveiro, Portugal, October 12-15, 2009. Proceedings*, ser. Lecture Notes in Computer Science, L. S. Lopes, N. Lau, P. Mariano, and L. M. Rocha, Eds., vol. 5816. Springer, 2009, pp. 53–64.

[2] W. Gu and R. J. Novak, "Agent-based modelling of mosquito foraging behaviour for malaria control," in *Transactions of the Royal Society of Tropical Medicine and Hygiene*, vol. 103. Oxford University Press, 2009, pp. 1105–1112.

[3] Y. Zhou, S. M. N. Arifin, J. Gentile, S. J. Kurtz, G. J. Davis, and B. A. Wendelberger, "An agent-based model of the anopheles gambiae mosquito life cycle," in *Proceedings of the 2010 Summer Computer Simulation Conference*, ser. SCSC '10. San Diego, CA, USA: Society for Computer Simulation International, 2010, pp. 201–208.

[4] S. Coakley, M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough, "Exploitation of high performance computing in the flame agent-based simulation framework," in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, ser. HPCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 538–545.

[5] (2013, Oct.) Repast - recursive porus agent simulation toolkit. [Online]. Available: http://repast.sourceforge.net/repast_3/index.html

[6] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "Mason: A multiagent simulation environment," *Simulation*, vol. 81, no. 7, pp. 517–527, Jul. 2005.

[7] Eckhoff PA, "A malaria transmission-directed model of mosquito life cycle and ecology," *Malaria Journal*, vol. 2011;12:303. doi: 10.1186/1475-2875-10-303, 2011.

[8] P. Richmond, S. Coakley, and D. Romano, "Cellular level agent based modelling on the graphics processing unit," in *High Performance Computational Systems Biology, 2009. HIBI '09. International Workshop on*, 2009, pp. 43–50.

[9] Khronos OpenCL Working Group, "The OpenCL 1.2 specification," http://www.khronos.org/opencl, 2012.

[10] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "LibWater: heterogeneous distributed computing made easy," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 161–172. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465008

[11] J. Verschelde, "Memory Coalescing Techniques," http://homepages.math.uic.edu/$\sim$jan/mcs572/memory_coalescing.pdf, 2012.

[12] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, "Automatic OpenCL device characterization: guiding optimized kernel design," in *Euro-Par*, 2011, pp. 438–452.

[13] Advanced Micro Devices Inc., "Bolt," http://hsa-libraries.github.io/Bolt/html/index.html, 2013.

[14] R. C. Tausworthe, "Random Numbers Generated by Linear Recurrence Modulo Two," *Mathematics of Computation*, vol. 19, pp. 201–209, 1965.

[15] P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics of Computation*, vol. 65, pp. 203–213, 1996.

[16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1992.

[17] H. Nguyen, *GPU Gems 3*, 1st ed. Addison-Wesley Professional, 2007.

[18] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Annals of Mathematical Statistics*, vol. 29, pp. 610–611, 1958.

[19] C. L. Phillips, J. A. Anderson, and S. C. Glotzer, "Pseudo-random number generation for Brownian Dynamics and Dissipative Particle Dynamics simulations on GPU devices," *Journal of Computational Physics*, vol. 230, no. 19, pp. 7191 – 7201, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999111003329

[20] Advanced Micro Devices Inc., "OpenCL Optimization Case Study: Simple Reductions ," http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/, 2013.

[21] J. E. Gentile, G. J. Davis, and S. S. C. Rund, "Verifying agent-based models with steady-state analysis," *Computational and Mathematical Organization Theory*, vol. 18, pp. 404–418, 2012.

[22] M. J. North, N. T. Collier, and R. J. Vos, "Experiences creating three implementations of the Repast agent modeling toolkit," *ACM Transactions on Modeling and Computer Simulation*, vol. 16(1), pp. 1–25, 2006.

[23] NVIDIA Corporation, "What do K20 users think?" http://www.nvidia.com/docs/IO/122634/K20-testimonial.pdf, 2013.