Profiling and Bottlenecks Analysis of an Agent-based Dengue Fever Simulation Model

Pablo A.S. Hugen¹, Guilherme Galante², Rogério L. Rizzi¹, Eduardo A.A. Cunha¹

¹ Colegiado de Ciência da Computacao
Universidade Estadual do Oeste do Paraná (UNIOESTE)
85819-110 – Cascavel – PR – Brazil

{pablo.hugen,guilherme.galante,rogerio.rizzi,eduardo.argou}@unioeste.br

Abstract. Agent-Based Model simulations plays a critical role in computational epidemiology, usually employing GPU aceleration to account for large scale scenarios. In this work, we profile and analyze an agent based model for dengue fever disease spreading simulation as part of a bachelor thesis work, to find bottlenecks and performance issues. We have discovered a bottleneck in two of the model kernels, and inefficient host-to-device memory transfers. The use of a unified memory architecture as also running more kernels in parallel using ISO C++17 Parallelism are proposed as future solutions to these challenges.

1. Introduction

The spread of infectious diseases, notably exemplified by the SARS-Cov-2 and H1N1 outbreaks, has been significantly accelerated by global interconnectivity, highlighting the critical role of computational technologies in public health management. Thus, mathematical models and simulations, as highlighted by [Rachah and Silva 2024], have proven indispensable for formulating effective health strategies. Moreover, the necessity for simulating complex epidemiological scenarios and advancements in High Performance Computing have propelled the use of Agent-Based Models (ABMs) in computational epidemiology simulations, as discussed by [Elsheikh 2024]. These models offer insights into epidemic dynamics by simulating disease progression on an individual level, emphasizing the spatial and temporal interaction between agents and their environment [Cunha et al. 2022]. However, the implementation of ABMs is not without challenges: They require significant computational effort, both in terms of memory storage and processing time, which becomes a limiting factor as the number of individuals or spatial resolution increases [Rosenstrom et al. 2024].

As a result, Graphics Processing Units (GPUs) acceleration have been used to deal with the agent scale problem, as shown by [Kitson et al. 2024, Thomopoulos and Tsichlas 2024], usually employing the CUDA programming model. Despite this, the model proposed by [Cunha et al. 2022], even though implemented using CUDA, still suffers from the scale problem, leading to impractical simulation times in real-world scenarios. Furthermore, recent studies have demonstrated the effectiveness of ISO C++ parallel algorithms in enhancing computational performance across a range of scenarios, using GPUs and other accelerators to offer a standardized, cross-platform approach to parallelism [Brown et al. 2019]. In view of this, to address these challenges, the authors are currently working on implementing a model based on the work of [Cunha et al. 2022] utilizing the ISO C++ parallel algorithms for potential performance improvements.
improvements. Therefore, the primary objective of the present work is to **profile the Cunha model to identify and analyze bottlenecks**, aiming to aid in significantly reducing simulation times for real world scenarios.

2. Methodology

In the simulator, each individual agent is characterized by attributes including age, location, and health status, within a spatial environment modeled as a mobility graph, as shown by Figure [1]. Agents interact within the surroundings through a series of time steps or cycles, enabling movement within the environment, contact with other agents, being prone to environmental forces and other control measures, and also transitions in health status. These status changes adhere to the **SEIRS** model (as illustrated in Figure [2]), cycling agents through predetermined health states: Susceptible, Exposed, Infected, and Recovered; thus dynamically simulating the spread and impact of diseases over time. This conceptual disease spreading model is based on the work by [Cunha et al. 2022], incorporating the Monte Carlo method to account for the stochastic nature of disease spreading, running multiple simulations with random parameters within a predetermined range. Specifically, the model is adapted for Dengue fever, incorporating both human and mosquito agents in the simulation. The [Cunha et al. 2022] original model implementation is available online at Github.

![Figure 1. SEIRS Model](image1)

![Figure 2. Mobility Graph](image2)

The new implementation leverages the **ISO C++17 Parallel Algorithms** specification, facilitating portable parallel programming with the Standard Template Library (STL). This is achieved through the **Nvidia HPC SDK**, which supports full C++17 on CPUs and allows offloading of parallel algorithms to Nvidia GPUs, thereby enabling GPU programming without the need for directives, pragmas, or annotations. Such an approach ensures that programs utilizing C++17 parallel algorithms are easily portable across most platforms. The full implementation can be found at [this repository](https://github.com).

For the execution and experimental setup of the simulation, a computer equipped with an Intel Core i7-11390H processor, featuring 4 cores and 16 threads, was used. This system have 16GB of RAM and an NVIDIA GeForce MX450 graphics card with 2GB GDDR6 memory. The simulation scenario consisted of 365 cycles, designed to emulate the dynamics within a city block of Cascavel/PR, providing a comprehensive view of the environmental interactions over an annual period. To analyze the performance of CUDA kernels within the simulation, the Nvidia Visual Profiler (nvvp) tool was employed, offering detailed insights into the execution efficiency and potential bottlenecks within the GPU-accelerated processes.
3. Results

First, the utilization of the Nvidia Visual Profiler tool analysed the simulation kernels performance. The profiling data, as summarized in Table 1, revealed the execution time, frequency of invocations, and the relative importance of each kernel with an overall effect greater than 1% in the context of the entire simulation. The total time of the simulation was roughly 1139.54s, with 811.19ms of profiling overhead.

Table 1. Execution time and Invocations of kernels with Importance > 1%

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Total Execution Time (s)</th>
<th>Invocations</th>
<th>Importance (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MosquitosMovement</td>
<td>419.7169</td>
<td>4380</td>
<td>40.2</td>
</tr>
<tr>
<td>MosquitosContact</td>
<td>323.8985</td>
<td>4380</td>
<td>31</td>
</tr>
<tr>
<td>HumansContact</td>
<td>122.0778</td>
<td>4380</td>
<td>11.7</td>
</tr>
<tr>
<td>HumansMovement</td>
<td>27.2122</td>
<td>1095</td>
<td>2.6</td>
</tr>
<tr>
<td>Generation</td>
<td>20.16748</td>
<td>361</td>
<td>1.9</td>
</tr>
<tr>
<td>MosquitosControl</td>
<td>14.4168</td>
<td>365</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Notably, the MosquitosMovement and MosquitosContact kernels dominated the computational effort, accounting for over 71% of the total importance, highlighting the implementation complexity of the mosquitoes dynamics, and displaying them as the primary bottlenecks within the simulation. Next, the HumansContact and HumansMovement kernels, while less computationally intensive, still played a significant role in the simulation time. Finally, the Generation and MosquitosControl kernels, though having lower importance percentages, can also be considered as optimization targets.

Following, the memory transfers patterns between the CPU and GPU were examined. It was identified host-to-device transfers at 7.11ms for 55 transfers, with 38.63MB at a throughput of 5.429 GB/s, and device-to-host transfers at 35.97ms for 14,747 invocations, transferring 102.402MB at 2.847 GB/s. With those data, the profiler was able to identify some key points in the memory performance of the simulator, as shown in the Figure 3.

![Figure 3. Data movement profiler analysis.](image)

Particularly, as presented in the “Inefficient Memcpy size” field, the profiling tool revealed a significant use of small and inefficient device-to-host memory copies, primarily due to frequent state transfers for I/O operations. To mitigate this, strategies such as deferring simulation state presentation until completion or employing CUDA Unified Memory technology are suggested. Additionally, it can be seen in the field “Low Memcpy Throughput” that the analysis identified issues with low memory throughput, detecting
transfers at only 56.032 MB/s in average. Also, “Low Memcpy Overlap” at 0% display a lack of parallel transfers, which could be addressed by implementing asynchronous memory transfers between the host and GPU. Also, other critical observations are the absence of concurrent kernel executions (“Low Kernel Concurrency” value at 0%) and no execution/transfer overlapping, highlighted on the section “Low Memcpy/Kernel Overlap”. Those informations presents a complex challenge that necessitates a redesign of the simulator architecture, an aspect currently under development.

4. Conclusion and future work

In conclusion, we found that the main bottlenecks of the [Cunha et al. 2022] model were the MosquitosMovement and MosquitosContact kernels, which accounted for over 71% of the total importance. The memory transfers patterns between the CPU and GPU revealed a lot inefficient device-to-host memory copies, primarily due to frequent state transfers for I/O operations. Finally, the absence of concurrent kernel execution was identified as a critical observation, posing a complex challenge that implies the necessity of a redesign of the simulator architecture to enhance overall performance.

Thus, future work is being directed towards a fundamental redesign of the simulator architecture to optimize data access patterns and reduce complexity in MosquitosMovement and MosquitosContact operations. This redesign aims to enable concurrent kernel execution through the implementation using ISO C++17 Parallel STL algorithms. Concurrently, efforts are underway to refactor state I/O operations, adopting unified memory architecture as a first class approach of the simulator.

References


