Programación paralela y computación de altas prestaciones

Task 1

Students: Alejandro Carmona Martínez, Álvaro Rubira García

Mails: alejandro.carmonam@um.es, alvaro.r.g@um.es

Academic year: 2024/25

Date: 04/10/2024

Contents

1 Questions

1.1 What kind of parallelism is exposed in the identified method?

We started by developing the following sequential code:

```cpp
void Ped::Model::seqTick() {
  // 1. Retrieve each agent
  for (std::vector<Ped::Tagent*>::iterator it = agents.begin();
       it != agents.end(); ++it) {
    // 2. Calculate next desired position
    (*it)->computeNextDesiredPosition();
    int desiredX = (*it)->getDesiredX();
    int desiredY = (*it)->getDesiredY();

    // 3. Set position to calculated desired one
    (*it)->setX(desiredX);
    (*it)->setY(desiredY);
  }
}
```

Listing 1: C++ code for the **seqTick** function in the Ped::Model class (**ped_model.cpp**)

We could refer to the parallelism found here as straightforward data parallelism. While we cannot execute several ticks of the model in parallel (as there is a dependence with previous ticks) we can however run the calculations for each agent in parallel as they are independent operations and one agent does not depend on the operations of another (for now). Therefore, we choose to divide the number of agents between available threads and compute their movement concurrently. Altogether, this kind of parallelism performs independent operations on a huge set of data objects in such a way that can be scaled by parallelizing. Finally, regarding scalability, this approach may also scale up with the number of agents and the number of CPU cores available.

1.2 List at least two alternatives for the OpenMP, and two alternatives for the C++ Threads implementation that you considered.

1.2.1 Alternatives to OpenMP

OneTBB

According to its webpage, "*Intel® oneAPI Threading Building Blocks (oneTBB) is a flexible performance library that simplifies the work of adding parallelism to complex applications across accelerated architectures, even if you're not a threading expert.*" [1].

As a high-performance multithreading tool, OneTBB differentiates itself from OpenMP as the user is asked to declare tasks that TBB will optimize dynamically instead of writing `#pragma` directives:

```cpp
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    // loop body
}
```

Listing 2: OpenMP Parallel Loop

```cpp
tbb::parallel_for(0, n, [&](int i) {
    // loop body
});
```

Listing 3: OneTBB Parallel Loop

This paradigm changes OpenMP's programmer-specified granularity and scheduling for TBB's work-stealing algorithm, which abstracts the programmer from the previous specifics, although it also allows the programmer to manually control granularity if needed.

Parallel Patterns Library (PPL)

The Parallel Patterns Library (PPL) [2] is a high-level, task-based programming framework for C++, similar to the previously introduced Intel oneTBB. By abstracting low-level thread management, it allows developers to focus on concurrency through tasks rather than threads directly. PPL provides generic parallel algorithms, such as `parallel_for_each`, making parallel programming more accessible and efficient by abstracting the end user from the specifics of the implementation. This approach reduces the complexity of implementing parallelism while improving scalability. As was the case with TBB, its task-based model is particularly well-suited for handling irregular workloads or problems with dynamic and unpredictable parallel patterns effectively.

1.2.2 Alternatives to C++ Threads

Boost.Thread

Boost.Thread [3] provides a level of granularity reminiscent of C++ threads by being able to create independent threads in a similar manner. Additionally, it is OS portable, and it bears a higher-level abstraction compared to `C++ threads` by simplifying thread creation, management, and synchronization.

Threads are executed independently and can be created, joined, or detached by using the `boost::thread` class.

```cpp
    boost::thread myThread([]() {
        // Thread logic
    });
    myThread.join();
```

Listing 4: Example: Creating a Thread

Boost.Thread [3] is ideal for scenarios requiring fine-grained synchronization or portability. It predates C++ threads and inspired many of its features, but it remains relevant due to its rich API and extended functionality, although Boost.Thread is heavier than native threads and lacks automatic thread reuse, making it less efficient than C++ threads or the previous task-based libraries.

C++ Coroutines

C++ Coroutines, introduced in C++20, allow easily programming of asynchronous and lazily evaluated operations. Unlike traditional threads, coroutines allow functions to be paused and resumed, enabling non-blocking operations without requiring a separate thread. They are implemented using `co_await`, `co_yield`, and `co_return` keywords, which provide a structured way to handle asynchronous events. Coroutines are especially useful for tasks like I/O operations, networking, or processing streams of data.

1.3 Once for OpenMP and once for C++ Threads, explain your chosen implementation. Include, amongst others, answers to the following questions:

1.3.1 OMP

The OMP implementation is really straightforward, as we only need to include `#pragma omp parallel for` before the main loop:

```cpp
void Ped::Model::ompTick() {
  // 1. Retrieve each agent
#pragma omp parallel for schedule(static)
  for (std::vector<Ped::Tagent*>::iterator it = agents.begin();
       it != agents.end(); ++it) {
    // 2. Calculate next desired position
    (*it)->computeNextDesiredPosition();
    int desiredX = (*it)->getDesiredX();
    int desiredY = (*it)->getDesiredY();

    // 3. Set position to calculated desired one
    (*it)->setX(desiredX);
    (*it)->setY(desiredY);
  }
}
```

Listing 5: C++ code for the **ompTick** function in the Ped::Model class (**ped_model.cpp**)

How is the workload distributed across the threads?

The processing of each agent will take similar amounts of time, so the total number of agents can be divided equally between all threads to effectively distribute the work. This can be done with static scheduling, which also provided the best results. An equal number of agents is assigned to each thread at the beginning (or as close to equal as possible, if the number of agents is not divisible by the number of threads).

Which number of threads gives you the best results? Why?

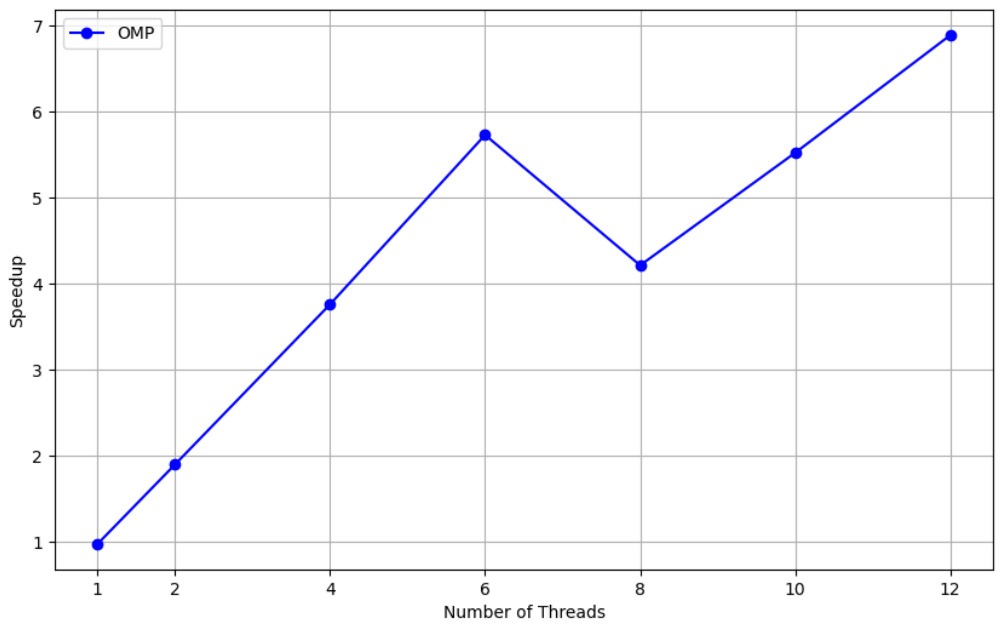


Figure 1: Speedup OMP vs Sequential for `commute_200000.xml`

We tested the OMP implementation on `commute_200000.xml` (as we think this scenario has enough agents to produce reliable results) and plot the observed speedup vs. the sequential version in figure 1. The best performance is achieved when 12 threads are used, as our testing has been done on a 6-core machine with Hyperthreading (as shown in appendix A), resulting in a total of 12 logical cores. Generally, when the number of threads increases, the speedup also increases, meaning that the parallel execution gains are overpowering the overhead due to the added thread creation, control, and synchronization mechanisms.

Nevertheless, the scaling in performance is not perfectly linear, as can be observed. Using 6 threads (without the need for hyperthreading) resulted in better core usage and therefore better performance than going to 8 cores, where hyperthreading is activated.

What are the possible drawbacks of this version?

OpenMP might have some drawbacks in comparison with the two other alternatives for dynamic workloads that require advanced task scheduling, or because of lack of integration with modern C++ idioms (lambdas, templates, RAII). We are using static scheduling and the code does not use any advanced idioms, so we still chose OpenMP.

Why did you choose this version over the other alternatives?

Mainly because of its ease of use and availability. Regarding ease of use, OpenMP excels in cases with simple parallel loops, and we saw a significant boost in performance by adding a single line of code. About availability, OpenMP is a widely recognized standard supported out-of-the-box by many compilers.

1.3.2 C++ threads

Initially, for the C++ threads implementation, we were creating and destroying `n_threads` threads for each simulation tick, which did not provide good results. We improved it by creating the threads only at the beginning and using barriers to ensure worker threads only start doing work when the main thread is ready.

We use a boolean flag that the main thread activates `freePthreads` to signal worker threads that there will be no more ticks. This could be avoided if the number of total ticks was known beforehand, but we assume `Ped::Model` does not have knowledge of this. The flag is of type `atomic<bool>` to restrict reordering of memory accesses.

```
void Ped::Model::setupPthreads() {
  exit_flag.store(false);
  pthread_barrier_init(&barrier, nullptr, nthreads);

  int start = 0;
  int remaining_agents = agents.size() % nthreads;

  // Create nthreads-1 threads
  for (int i = 0; i < nthreads - 1; i++) {
    // First remaining_agents threads will have 1 more agent than the rest
    int chunk_size = agents.size() / nthreads + (i < remaining_agents ? 1 : 0);
    int end = start + chunk_size;

    // Created threads will wait each tick at barrier in pthreadTask for main
    // thread
    workers[i] = std::thread([this, start, end]() {
        while (pthreadWorkerTick(start, end));
    });
    start = end;
  }
  main_thread_start = start;
}

bool Ped::Model::pthreadWorkerTick(int start, int end) {
  // Ensure workers wait for the main thread
  pthread_barrier_wait(&barrier);

  if (exit_flag.load()) {
    return false;
  }

  for (int i = start; i < end; i++) {
    // 1. Retrieve each agent
    Ped::Tagent* agent = agents[i];
    // 2. Calculate next desired position
    agent->computeNextDesiredPosition();
    int desiredX = agent->getDesiredX();
    int desiredY = agent->getDesiredY();

    // 3. Set position to calculated desired one
    agent->setX(desiredX);
    agent->setY(desiredY);
  }

  // Ensure main thread only exits after all workers have finished their work
```

```cpp
  pthread_barrier_wait(&barrier);

  return true;
}

void Ped::Model::pthreadTick() {
  pthreadWorkerTick(main_thread_start, agents.size());
}

void Ped::Model::freePthreads() {
  exit_flag.store(true);
  pthread_barrier_wait(&barrier);
  for (int i = 0; i < nthreads - 1; i++) {
    workers[i].join();
  }
}
```

Listing 6: C++ code for the C++ threads implementation)

How is the workload distributed across the threads?

We try to assign the same number of agents to each thread for the same reasons we chose static scheduling in OMP.

Which number of threads gives you the best results? Why?

We plot the speedup average for C++ threads in figure 2. We can see the best results are again achieved with 12 threads. Similarly to OMP, the activation of hyperthreading creates a dip in performance when going to 8 threads.

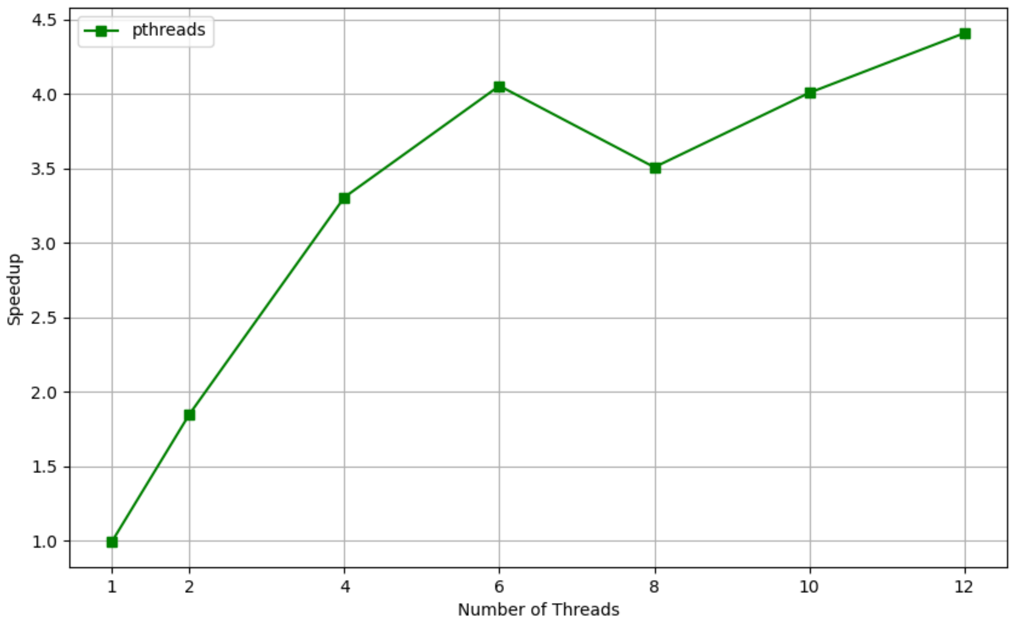


Figure 2: Speedup for C++ threads vs Sequential for `commute_200000.xml`

What are the possible drawbacks of this version?

C++ threads is not as feature-rich as many of the alternatives. For instance, we do not have access to a thread pool wich could have been used in this kind of task.

Why did you choose this version over the other alternatives?

`std::tread` is part of the Standard Libary, meaning no dependencies need to be managed. Moreover, although limited, we believe its features are enough for our simple use case.

1.4 Which version (OpenMP, C++ Threads) gives you better results? Why?

For C++ threads, the code is less easy to manage and understand than the OMP version, which only requires a pragma before the loop. Therefore, it is also more prone to coding mistakes. Moreover, it is difficult to manually achieve the same performance compared to using a really optimized framework for parallel computing such as OMP, resulting in higher overhead. We can see this reflected in figure 3, where OMP gives better results for all thread numbers.

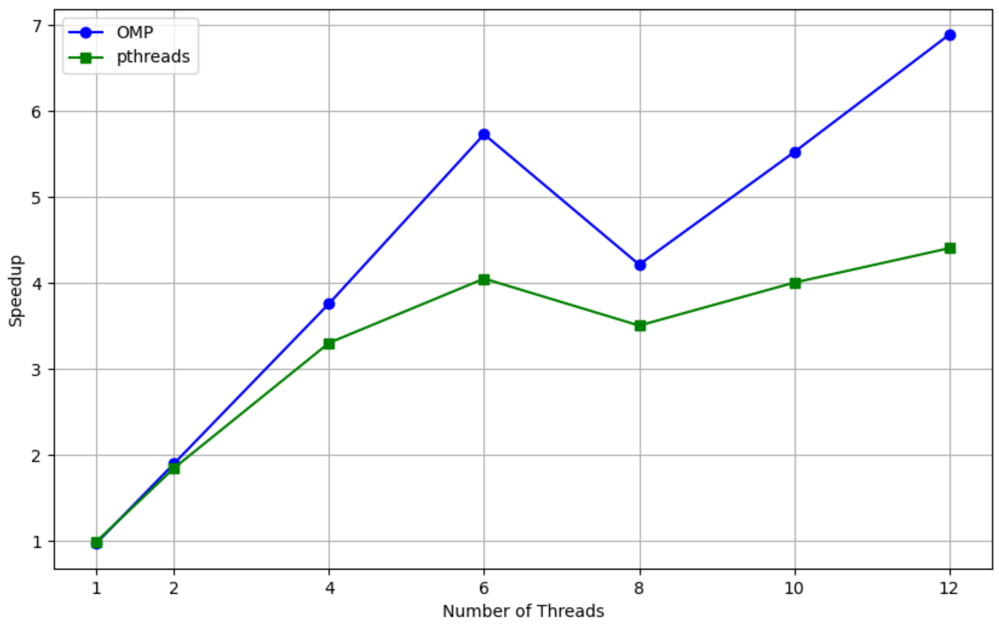


Figure 3: Speedup graph for OMP and C++ Threads for `commute_200000.xml`

1.5 What can you improve such that the worse performing version could catch up with the faster version?

We could try changing thread affinity settings, to check if manually pinning threads to specific CPU cores improves performance on the C++ threads implementation.

1.6 Consider a scenario with 7 agents. Using a CPU with 4 cores, how would your two versions distribute the work across threads?

Assuming one thread per core, the OMP version would assign 2 agents to the first three threads and 1 agent to the remaining thread. The C++ threads version would do the same.

1.7 For your OpenMP solution, what tools do you have to control the workload distribution?

The pragma `omp for` allows a `schedule` clause. We can set the scheduling policy in this clause.

A Apendix A: Test Machine Specifications

We are using an Intel Core i7-10750H CPU with x86_64 architecture, 1 socket, 6 cores per socket and 2 threads per socket (this is taking into account hyperthreading). This CPU has 12 MB L3 cache, 1.5 MB L2 cache, 192 KB data L1 and 192 KB instruction L1.

The result of running `lscpu` is the following:

```
Architecture: x86_64
  CPU op-mode(s): 32-bit, 64-bit
  Address sizes: 39 bits physical, 48 bits virtual
  Byte Order: Little Endian
CPU(s): 12
  On-line CPU(s) list: 0-11
Vendor ID: GenuineIntel
  Model name: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    CPU family: 6
    Model: 165
    Thread(s) per core: 2
    Core(s) per socket: 6
    Socket(s): 1
    Stepping: 2
    CPU(s) scaling MHz: 85%
    CPU max MHz: 5000,0000
    CPU min MHz: 800,0000
    BogoMIPS: 5199,98
    Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
       ↪ clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
       ↪  constant_tsc art arch_perfmon pebs
                        bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni
                            ↪ pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg
                            ↪ fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
                            ↪ popcnt tsc_deadli
                        ne_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
                            ↪ cpuid_fault epb ssbd ibrs ibpb stibp ibrs_enhanced tpr_
                            ↪ shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust
                            ↪ bmi1 avx2 sm
                        ep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt
                            ↪ xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts
                            ↪ hwp hwp_notify hwp_act_window hwp_epp vnmi pku ospke md_
                            ↪ clear flush_
                        l1d arch_capabilities
Virtualization features:
  Virtualization: VT-x
Caches (sum of all):
  L1d: 192 KiB (6 instances)
  L1i: 192 KiB (6 instances)
  L2: 1,5 MiB (6 instances)
  L3: 12 MiB (1 instance)
NUMA:
  NUMA node(s): 1
  NUMA node0 CPU(s): 0-11
Vulnerabilities:
  Gather data sampling: Mitigation; Microcode
  Itlb multihit: KVM: Mitigation: VMX disabled
  L1tf: Not affected
  Mds: Not affected
```

```
Meltdown: Not affected
Mmio stale data: Mitigation; Clear CPU buffers; SMT vulnerable
Reg file data sampling: Not affected
Retbleed: Mitigation; Enhanced IBRS
Spec rstack overflow: Not affected
Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl
Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Spectre v2: Mitigation; Enhanced / Automatic IBRS; IBPB conditional; RSB filling;
    ↪ PBRSB-eIBRS SW sequence; BHI SW loop, KVM SW loop
Srbds: Mitigation; Microcode
Tsx async abort: Not affected
```

This machine has 32 GB RAM and a 2060 RTX Mobile GPU.

References

[1] Intel Corporation. *Intel OneAPI OneTBB*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html. Accessed: 2024-12-29.

[2] Microsoft. *Parallel Patterns Library (PPL)*. https://learn.microsoft.com/es-es/cpp/parallel/concrt/parallel-patterns-library-ppl?view=msvc-170. Accessed: 2024-12-29.

[3] Boost. *Boost.Thread Library Overview*. https://www.boost.org/doc/libs/1_83_0/doc/html/thread.html#thread.overview. Accessed: 2024-12-29.

Programación paralela y computación de altas prestaciones

Task 2

Students: Alejandro Carmona Martínez, Álvaro Rubira García

Mails: alejandro.carmonam@um.es, alvaro.r.g@um.es

Academic year: 2024/25

Date: 17/10/2024

Contents

1 A. What kind of parallelism is exposed with your code refactorization?

This refactoring is particularly useful for exposing data-level parallelism. Data is placed next to values of the same kind and with the same purpose in memory, allowing easy SIMD parallelization where the same instruction is simultaneously applied on different data.

2 B. Explain your solution.

We are now storing all the agents together in a SoA (Structure of Arrays) class:

```cpp
namespace Ped {
class TagentSoA {
...
  int* agents_x;
  int* agents_y;

  int* destination_x;
  int* destination_y;
  int* destination_r;

  int* current_waypoint;
  int* waypoint_start_offset;
  Waypoint* waypoints;
...
};
```

In this new code, each agent can now be represented with the values in a specific index in each of the arrays (e.g. the first agent stores its current coordinates in `agents_x` ↪ `[0]` and `agents_y[0]`). For instance, we store the `x` and `y` coordinate of all agents the `agents_x` and `agents_y` arrays. The `destination` arrays store the coordinates and radius of the current destination of each agent. The `current_waypoint` array stores the index of the waypoint associated with the current destination in the waypoint array.

Finally, `waypoints` stores all of the waypoints for all agents. `waypoint_start_offset` tells each agent where in `waypoints` does its particular waypoint array start. This allows us to store all of the waypoints for all agents together in `waypoints`, so we can easily send all waypoints to the GPU at the start of the program.

All these modifications imply changing the initial `sequentialTick` code to account for the new code structure using SoA. Moreover, the code for displaying agents should now expect agents coordinates in SoA format, and has been modified to do so. All in all, these modifications already grant us a 2.54 speedup in sequential code over the sequential code presented in task 1.

3 C. Which parallelization has been most beneficial in these two labs? Support your case using plotted timing measurements.

We tested all implementations with `commute_200000.xml`. In figure 1 we plot speedups against the sequential SoA implementation. We currently get the best performance by running the vectorized implementation along with OMP parallelization (i.e. using multiple threads with OMP where each thread processes its agents with SIMD vector instructions). Each core in this implementation is processing 4 agents at a time using SIMD, so its speedup is roughly 4 times better than the speedup of OMP alone, which was the previous best.

For the vector+OMP implementation, we get peak performance with 6 threads. The benchmarks were run on a CPU with 6 cores and hyperthreading (this means 12 logical threads), so the best results are obtained without hyperthreading. This did not happen with the OpenMP (without SIMD) implementation: using vector instructions, the higher core utilization results in worse results with hyperthreading.

The CUDA and sequential implementations always run with 1 CPU thread, so we display them as horizontal lines for easy comparison with other implementations.

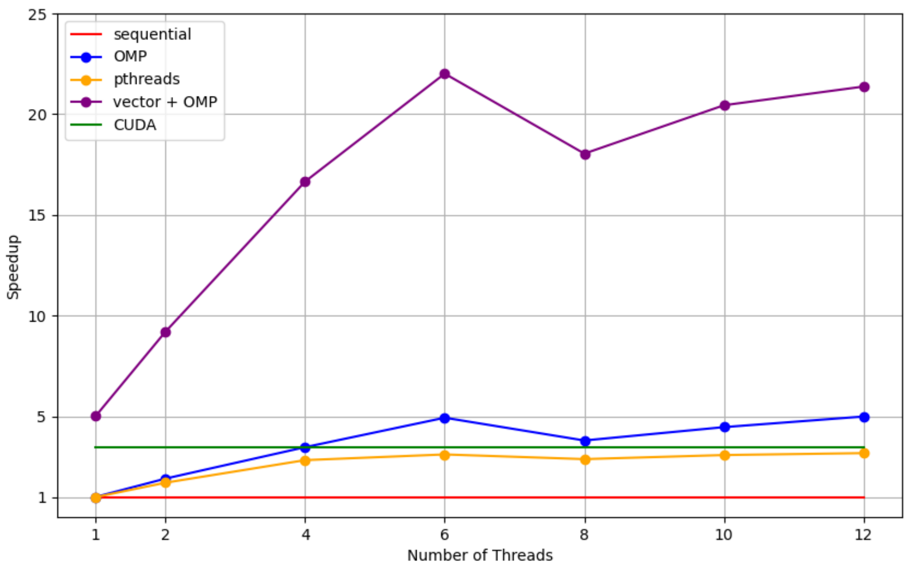


Figure 1: Speedup against sequential SoA for `commute_200000.xml`

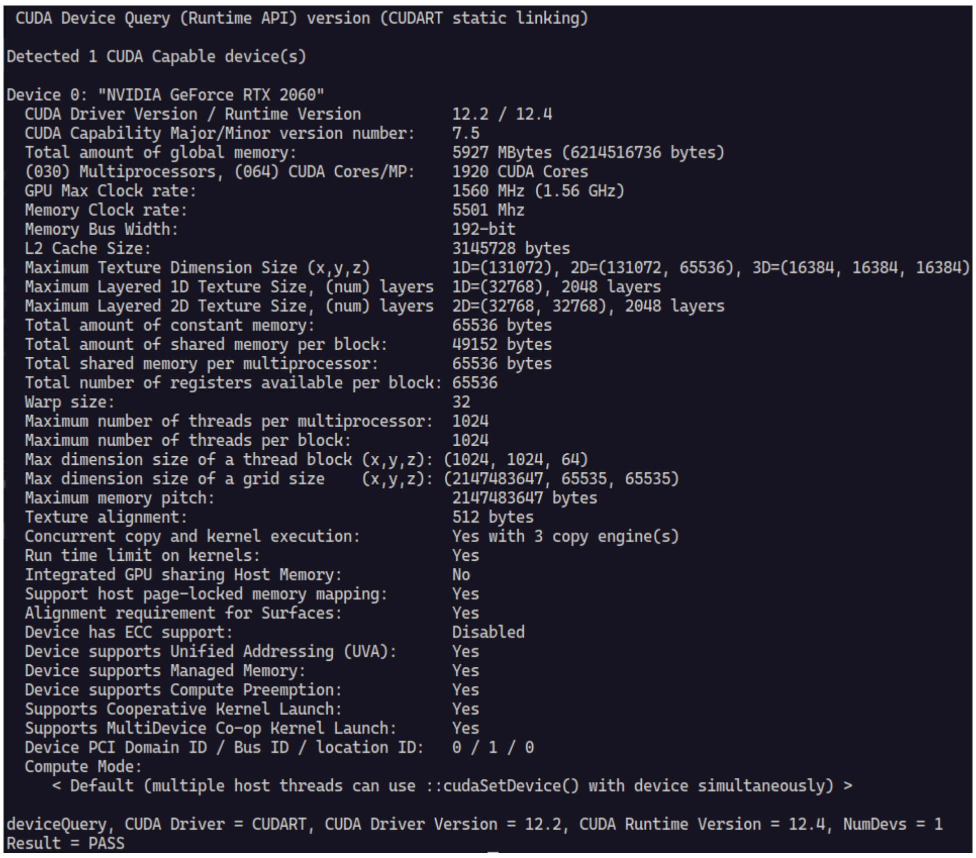
4 D. Compare the effort required for each of the three (four) parallelization techniques. Would it have made a difference if you had to build the program from scratch, instead of using given code?

Vectorization and CUDA were the most time-consuming parallelization techniques, both requiring alterations to the way program memory was handled. Using OMP was surprisingly easy (just adding a `parallel for` directive) and `pthreads` was also relatively easy to work with when creating a straightforward parallelization (although optimizing it further to get performance similar to OpenMP might be more complicated).

If we had created the program from scratch and knew we might want to use SIMD instructions, we could have used the SoA approach from the beginning. As a consequence, we would not have needed to refactor the codebase.

5 E. (Bonus:) List your GPU specifications.

It is an NVIDIA GeForce RTX 2060 Mobile with the following specs:



```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce RTX 2060"
  CUDA Driver Version / Runtime Version          12.2 / 12.4
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 5927 MBytes (6214516736 bytes)
  (030) Multiprocessors, (064) CUDA Cores/MP:    1920 CUDA Cores
  GPU Max Clock rate:                            1560 MHz (1.56 GHz)
  Memory Clock rate:                             5501 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 3145728 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.2, CUDA Runtime Version = 12.4, NumDevs = 1
Result = PASS
```

Figure 2: GPU specifications

6 F. (Bonus:) Is this code section suitable to be offloaded to the GPU? What makes it suitable / not suitable?

No, the parallelized section of the simulation is not particularly well suited to this GPU, and GPUs in general, mainly due to its lack of arithmetic intensity, and although not as directly responsible in this scenario, the use of double precision and the memory transfers needed are to be taken into account when using GPU acceleration.

Arithmetic intensity [1] refers to the ratio of operations to memory access in a computational task (ops:byte). It is a measure of how efficiently a computer algorithm utilizes the memory bandwidth available.

Although at first glance, a parallel code such as the one in the `tick()` function seems like a good candidate for GPU acceleration, in reality, as we process agents, each agent comes with its own data to load, but that data is only used once, which turns into a smaller ratio of operations per byte loaded as data re-utilization would allow for more operations per byte and better usage of the computational resources.

These programs or program segments are commonly referred to as memory bounded ones, as their performance is limited by memory rather than computational power.

Regarding memory transfers, although GPUs (and this one in particular) have a usually high bandwidth (336 GB/s) [2], this is limited by the bandwidth of the CPU when memory transfers between the two happen, so, this could be an issue compared to CPU execution where memory transfers are not necessary. However, due to the program design, there are only two memory transfers between CPU and GPU, so this is negligible.

Finally, albeit disguised by the low arithmetic intensity in this scenario, double precision performance in GPUs may cause a slowdown in respect to single precision of about 30x [2]. Yet it could be negatively impacting performance.

Therefore, the performance of the GPU is limited for this program due to its peak performance to bandwidth ratio both for double precision and single precision. If single precision was used, this issue would still be prevalent, although performance may improve.

7 G. (Bonus:) Give a short summary of similarities and differences between SIMD and GPU programming.

As already mentioned, **both SIMD (Single Instruction, Multiple Data) and GPU extract data level parallelism**. **SIMD**, supported by modern CPUs, allows for **parallel execution of the same operation across multiple data points** within a single CPU instruction. **GPU** programming, on the other hand, **leverages the massive parallel processing** capabilities of modern Graphics Processing Units (GPUs) to handle even larger-scale parallel tasks.

The differences between SIMD and GPU programming arise primarily from their distinct hardware architectures, levels of parallelism, typical use cases, and programming models. SIMD operates on the CPU using specialized vector instructions, such as AVX2, which allow parallel processing within the CPU's registers. In contrast, GPU programming (e.g., CUDA) leverages the GPU's thousands of cores, optimized for massively parallel computations.

SIMD achieves parallelism at the register level, processing a limited number of data elements within the CPU's vector registers. GPUs, however, support thread-level parallelism, allowing thousands of lightweight threads to run concurrently, making them ideal for highly parallel workloads. This ties in with the programming model, as SIMD programming typically involves the use of intrinsics or compiler-driven auto-vectorization to optimize performance. In contrast, GPU programming requires explicit use of APIs like CUDA or OpenCL, where developers must manage threads, memory hierarchies, and synchronization directly.

Finally, regarding its use, SIMD is effective for vectorized operations on relatively small datasets, such as in image processing or real-time systems. GPUs, on the other hand, are suited for large-scale parallel tasks, including deep learning, large matrix operations, and scientific simulations, where their massive parallelism can be fully exploited.

A Apendix A: Test Machine Specifications

We are using an Intel Core i7-10750H CPU with x86_64 architecture, 1 socket, 6 cores per socket and 2 threads per socket (this is taking into account hyperthreading). This CPU has 12 MB L3 cache, 1.5 MB L2 cache, 192 KB data L1 and 192 KB instruction L1.

The result of running `lscpu` is the following:

```
Architecture: x86_64
  CPU op-mode(s): 32-bit, 64-bit
  Address sizes: 39 bits physical, 48 bits virtual
  Byte Order: Little Endian
CPU(s): 12
  On-line CPU(s) list: 0-11
Vendor ID: GenuineIntel
  Model name: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    CPU family: 6
    Model: 165
    Thread(s) per core: 2
    Core(s) per socket: 6
    Socket(s): 1
    Stepping: 2
    CPU(s) scaling MHz: 85%
    CPU max MHz: 5000,0000
    CPU min MHz: 800,0000
    BogoMIPS: 5199,98
    Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
       ↪ clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
       ↪  constant_tsc art arch_perfmon pebs
                          bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni
                              ↪ pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg
                              ↪ fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
                              ↪ popcnt tsc_deadli
                          ne_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
                              ↪ cpuid_fault epb ssbd ibrs ibpb stibp ibrs_enhanced tpr_
                              ↪ shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust
                              ↪ bmi1 avx2 sm
                          ep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt
                              ↪ xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts
                              ↪ hwp hwp_notify hwp_act_window hwp_epp vnmi pku ospke md_
                              ↪ clear flush_
                          l1d arch_capabilities
Virtualization features:
  Virtualization: VT-x
Caches (sum of all):
  L1d: 192 KiB (6 instances)
  L1i: 192 KiB (6 instances)
  L2: 1,5 MiB (6 instances)
  L3: 12 MiB (1 instance)
NUMA:
  NUMA node(s): 1
  NUMA node0 CPU(s): 0-11
Vulnerabilities:
  Gather data sampling: Mitigation; Microcode
  Itlb multihit: KVM: Mitigation: VMX disabled
  L1tf: Not affected
  Mds: Not affected
```

```
Meltdown: Not affected
Mmio stale data: Mitigation; Clear CPU buffers; SMT vulnerable
Reg file data sampling: Not affected
Retbleed: Mitigation; Enhanced IBRS
Spec rstack overflow: Not affected
Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl
Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Spectre v2: Mitigation; Enhanced / Automatic IBRS; IBPB conditional; RSB filling;
    ↪ PBRSB-eIBRS SW sequence; BHI SW loop, KVM SW loop
Srbds: Mitigation; Microcode
Tsx async abort: Not affected
```

This machine has 32 GB RAM and a 2060 RTX Mobile GPU.

B Running the program

The binary can be run with the following arguments (after running `make` in the root of the project):

```
Usage: ./demo/demo [-h|--help] [--timing-mode] [--implementation=(CUDA,vector,OMP,
    ↪ pthreads,sequential)] [-n(NUM_THREADS)] SCENARIO_FILE
e.g.: ./demo/demo --timing-mode --implementation=OMP -n12 commute_200000.xml
(NUM_THREADS must be > 0 and < 16)
```

More information about the options can be displayed by running `./demo/demo -h`.

References

[1] ScienceDirect. *Arithmetic Intensity*. 2024. URL: https://www.sciencedirect.com/topics/computer-science/arithmetic-intensity.

[2] TechPowerUp. *GeForce RTX 2060 Mobile Specs*. 2024. URL: https://www.techpowerup.com/gpu-specs/geforce-rtx-2060-mobile.c3348.

Programación paralela y computación de altas prestaciones

Task 3

Students: Alejandro Carmona Martínez, Álvaro Rubira García

Mails: alejandro.carmonam@um.es, alvaro.r.g@um.es

Academic year: 2024/25

Date: 11/11/2024

Contents

1 Questions

1.1 Is it possible to parallelize the given code with a simple omp parallel for (compare lab 1)? Explain.

While it was possible for the `tick` function of lab 1 (shown in listing 1), it is not possible to parallelize the `move` function (shown in listing 2) using a simple `#pragma omp parallel ↪ for` due to data dependencies and potential race conditions between agents. The `move` function involves interactions between agents and their neighbors to determine available positions. When multiple agents are processed in parallel, they may simultaneously read and write to shared data structures, such as the positions of agents or the quadtree. This can lead to inconsistencies and conflicts, as agents might move to the same position or fail to correctly detect occupied positions.

These problems did not exist in task 1, where there were no conflict-related issues. Therefore, the `ompTick` 1 function could be parallelized with an OpenMP `parallel for loop ↪` , as each agent's computation was independent of others. In `ompTick`, agents compute their next desired positions and update their positions without considering the states of neighboring agents. No shared data structures are being modified that could cause race conditions, so each iteration of the loop can safely execute in parallel without the need for synchronization.

Therefore, while `ompTick` benefits from straightforward parallelization due to the lack of inter-agent dependencies, the `move` function cannot be parallelized in the same manner without introducing proper synchronization mechanisms or changes in the shared data usage to ensure correct agent interactions.

```cpp
void Ped::Model::ompTick() {
  // 1. Retrieve each agent
#pragma omp parallel for schedule(static)
  for (std::vector<Ped::Tagent*>::iterator it = agents.begin();
       it != agents.end(); ++it) {
    // 2. Calculate next desired position
    (*it)->computeNextDesiredPosition();
    int desiredX = (*it)->getDesiredX();
    int desiredY = (*it)->getDesiredY();

    // 3. Set position to calculated desired one
    (*it)->setX(desiredX);
    (*it)->setY(desiredY);
  }
}
```

Listing 1: C++ code for the `ompTick` function in the `Ped::Model` class (`ped_model.cpp`) from Task 1

```cpp
void Ped::Model::move(Ped::Tagent* agent) {
   //Search for neighboring agents
  set<const Ped::Tagent*> neighbors;
   = getNeighbors(agent->getX(), agent->getY(), 2);
   //Retrieve their positions
  std::vector<std::pair<int, int>> takenPositions;
  for (std::set<const Ped::Tagent*>::iterator neighborIt = neighbors.begin();
       neighborIt != neighbors.end(); ++neighborIt) {
    std::pair<int, int> position((*neighborIt)->getX(),
    (*neighborIt)->getY()); takenPositions.push_back(position);
  }
   //Compute the three alternative positions that would bring the agent closer to his
       ↪ desiredPosition, starting with the desiredPosition itself
  std::vector<std::pair<int, int>> prioritizedAlternatives;
  std::pair<int, int> pDesired(agent->getDesiredX(), agent->getDesiredY());
  prioritizedAlternatives.push_back(pDesired);
  int diffX = pDesired.first - agent->getX();
  int diffY = pDesired.second - agent->getY();
  std::pair<int, int> p1, p2;

  if (diffX == 0 || diffY == 0) {
     Agent wants to walk straight to North, South, West or East
    p1 = std::make_pair(pDesired.first + diffY, pDesired.second + diffX);
    p2 = std::make_pair(pDesired.first - diffY, pDesired.second - diffX);
  } else {
     Agent wants to walk diagonally
    p1 = std::make_pair(pDesired.first, agent->getY());
    p2 = std::make_pair(agent->getX(), pDesired.second);
  }
  prioritizedAlternatives.push_back(p1);
  prioritizedAlternatives.push_back(p2);

   //Find the first empty alternative position
  for (std::vector<pair<int, int>>::iterator it =
          prioritizedAlternatives.begin();
       it != prioritizedAlternatives.end(); ++it) {
     If the current position is not yet taken by any neighbor
    if (std::find(takenPositions.begin(), takenPositions.end(), *it) ==
        takenPositions.end()) {
       //Set the agent's position
      agent->setX((*it).first);
      agent->setY((*it).second);
       //Update the quadtree
       (*treehash)[agent]->moveAgent(agent);
      break;
    }
  }
}
```

Listing 2: C++ code for the original `move` function in the `Ped::Model` class (`ped_model.cpp`) from Task 3

1.2 How would the global lock solution (sec 1.1) perform with an increasing number of threads? Would the second simple solution perform better in this regard? Can the scenario affect the relative performance of the second simple solution?

The first global lock solution would serialize the code. Very little computation would be made in parallel, as all threads have to synchronize (acquire a lock) before moving each agent. The movement of agents would be done completely sequentially. This would quickly reach a point where adding more threads would not benefit performance at all if it even increases performance at first.

The second option seems better for scalability, provided that we can afford to have one lock per location. However, needing to acquire a lock for each agent movement would probably still result in bad performance. Moreover, in areas where agents managed by many different threads are colliding, the program would tend to serialize again, as threads may have to check if an empty position is still not taken by acquiring locks that are heavily contended.

1.3 Consider a scenario file designed to generate the worst-case load balancing for your solution. Describe and try to quantify how bad it could get. Worse than the sequential solution?

Given that we didn't implement dynamic region creation (see section 2.1), the worst case for our solution would start by spawning all agents in one corner of the scenario and having a few waypoints in the opposite region. This initial positioning of agents and waypoints is chosen because our regions are created to contain all waypoints and the initial positions of agents.

Once they move towards the waypoints, all agents would be inside the same region and managed by a single thread. This situation can be made worse if the waypoints are placed in such a way that the agents move near the region's border: collision avoidance for these agents moving near borders will need to use expensive CAS operations to protect against threads that manage adjacent regions. The resulting scenario would be similar to the one shown in figure 1, where red squares show how regions are divided. All the agents started in the lower right corner and are now in the upper-left region, where there are three waypoints. This can be worse than the sequential version because of the synchronization overhead and the unused spawned threads. Additionally, if some agents eventually cross to other regions (we can see some agents crossed to the upper-right region in figure1), they will do so for brief periods of time, but will still add the overhead of transferring ownership of agents between regions.

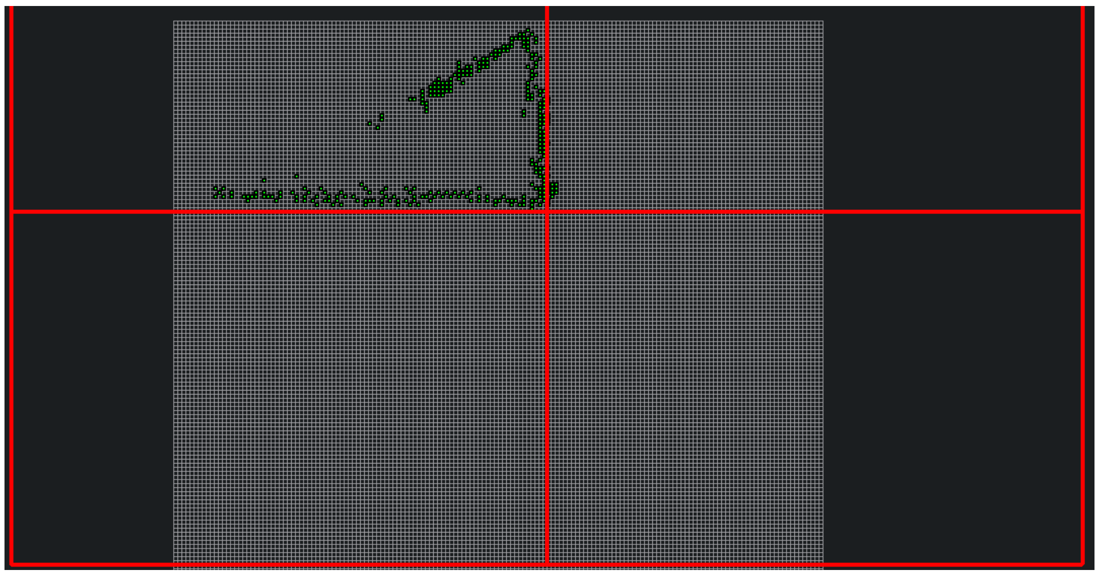


Figure 1: Program output for worst case scenario

1.4 Would your solution scale on a 100+ core machine?

As of this moment, the number of regions is statically set to 4, which, in turn, limits our peak scalability to 4 active threads, hence our solution's scalability is limited to 4 cores.

If our solution allowed N regions, and therefore, N threads, our solution may take advantage of 100+ core machines. However, this may not always introduce performance gains.

As explained in the evaluation section 3, depending on the size of the scenario, our sequential implementation may outperform the parallel one if the size is small enough. This is due to the overhead needed to create parallelism, both from the thread creation itself and the synchronization mechanisms needed to allow parallelism without race conditions. Hence, using too many regions and therefore, too many threads for managing only a few agents may cause performance to drop.

2 Bonus Assignment Questions

2.1 Briefly describe your solution. Focus on how you addressed load balancing and synchronization, and justify your design decisions.

We are using a boolean matrix to avoid collisions between agents. Each position has a corresponding byte that indicates if it is taken (we use bytes instead of bits because the atomic operations in our CPU operate on 1 byte as a minimum). Agents of the scenarios provided for this assignment move in a relatively small area, so this matrix easily fits in memory.

One benefit of using a matrix is that CAS operations can be easily integrated. For instance, when an agent is trying to move to a position near a border between regions (if it is not close to the border, synchronization is not needed), we first check if the position is empty by doing a regular memory read. If the position is empty, we then use CAS to check that it is still empty and set it to taken atomically.

We only implemented the CAS bonus assignment, and not the dynamic region adjustment one. That is, our solution always uses 4 predefined quadrants for assigning agents to threads. The area assigned to each thread is only modified to make it bigger if an agent tries to get out of the area of the matrix. We use `pragma omp task` to create one task per region.

2.2 Can your solution be improved? Describe changes you can make, and how it would affect the execution time.

The first point of improvement could be the introduction of support for N regions instead of 4. Then dynamic region resizing would allow for a better load balancing of the agents. Finally, we could look for a way to reduce the size of the matrix, too, as in some specific low-agent scenarios, this matrix can be sparse (really big and populated with few agents), so a huge amount of memory space is to be consumed.

3 Evaluation

In this section, we will perform a timed evaluation of our parallel and sequential implementations. To accurately profile the performance of each implementation and how they relate to each other, we will use different scenarios:

1. **Generic Low Agent Workload** 3.1
2. **Generic High Agent Workload** 3.2
3. **Specific Worst Case Workload** 3.3

3.1 Scenario 1: Generic Low Agent Workload

This is the original `lab3-scenario.xml` file (listing 3), generating around 400 agents. It doesn't have enough agents to allow for speed-up when running with multiple threads, and its waypoints only span across 2 quadrants. The parallelization overhead caused by both thread creation and synchronization mechanisms along with not ideal waypoint positioning makes it go slower than the sequential implementation as shown in figure 2 (we are plotting the speedup of the parallel version against the sequential one).

```xml
<welcome>
  <waypoint id="w1" x="20" y="60" r="10" />
  <waypoint id="w2" x="160" y="60" r="10" />

  <agent x="0" y="60" n="200" dx="40" dy="60">
    <addwaypoint id="w1" />
    <addwaypoint id="w2" />
  </agent>

  <agent x="160" y="60" n="200" dx="50" dy="30">
    <addwaypoint id="w2" />
    <addwaypoint id="w1" />
  </agent>
</welcome>
```

Listing 3: lab3-scenario.xml

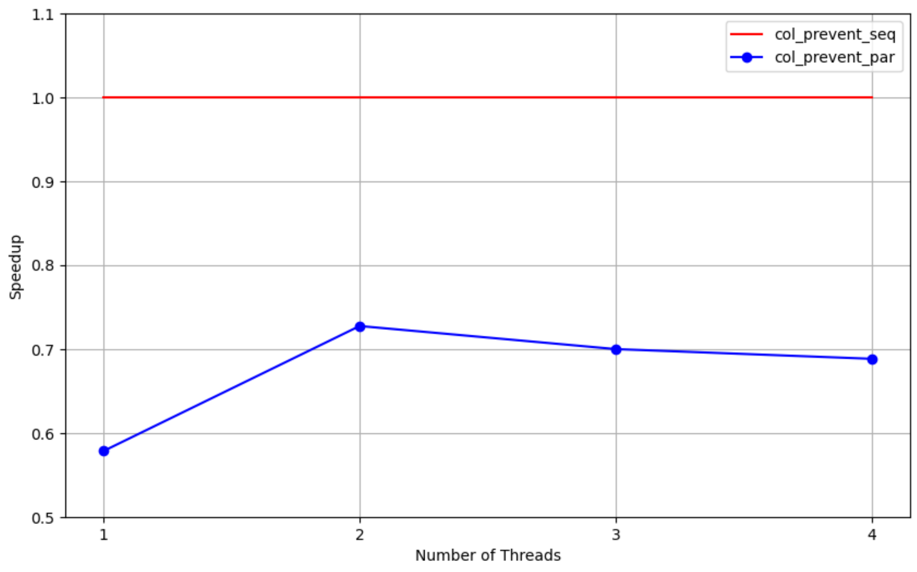


Figure 2: Speedup for scenario with few agents

3.2 Scenario 2: Generic High Agent Workload

We use a new file named `big_scenario.xml` (listing 4) for this scenario. This scenario creates around 125800 agents (even though each `agent` element in the `.xml` specifies 100000 agents, most are removed to avoid overlapping positions and the total ends up being 125800), and they are evenly distributed between regions. A noticeable speedup (1.93x) can be seen in figure 3 when using 4 threads against the sequential implementation.

```xml
<welcome>
  <waypoint id="w1" x="0" y="150" r="20" />
  <waypoint id="w2" x="100" y="0" r="20" />
  <waypoint id="w3" x="200" y="0" r="20" />
  <waypoint id="w4" x="400" y="100" r="20" />
  <waypoint id="w5" x="400" y="200" r="40" />
  <waypoint id="w6" x="250" y="300" r="20" />
  <waypoint id="w7" x="100" y="350" r="40" />

  <agent x="0" y="150" n="100000" dx="150" dy="150">
    <addwaypoint id="w2" />
    <addwaypoint id="w3" />
    <addwaypoint id="w4" />
    <addwaypoint id="w5" />
    <addwaypoint id="w6" />
    <addwaypoint id="w7" />
    <addwaypoint id="w1" />
  </agent>
  <!-- And so on for a total of 7 agent tags, one per waypoint -->
[...]
```

Listing 4: big_scenario.xml

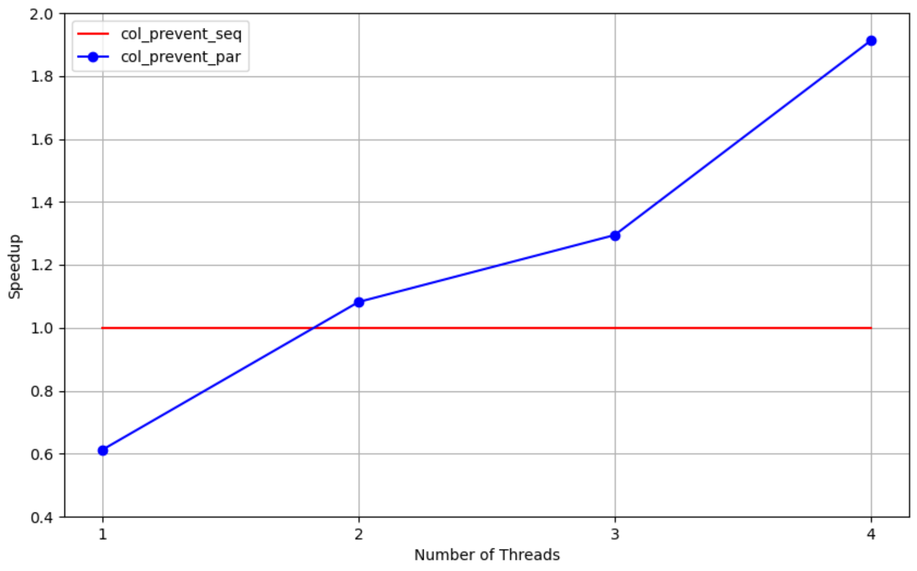


Figure 3: Speedup for scenario with many agents

3.3 Scenario 3: Specific Worst Case Workload

This scenario follows the description given in section 3 (listing 5). In figure 4 we can see our implementation cannot get a speedup and even gets worse with the number of threads.

```xml
<welcome>
  <waypoint id="w1" x="-5" y="45" r="15"/>
  <waypoint id="w2" x="100" y="-5" r="15"/>
  <waypoint id="w3" x="100" y="55" r="15"/>

  <agent x="160" y="70" n="300" dx="50" dy="50">
    <addwaypoint id="w1"/>
    <addwaypoint id="w2"/>
    <addwaypoint id="w3"/>
  </agent>
</welcome>
```

Listing 5: worst_scenario.xml

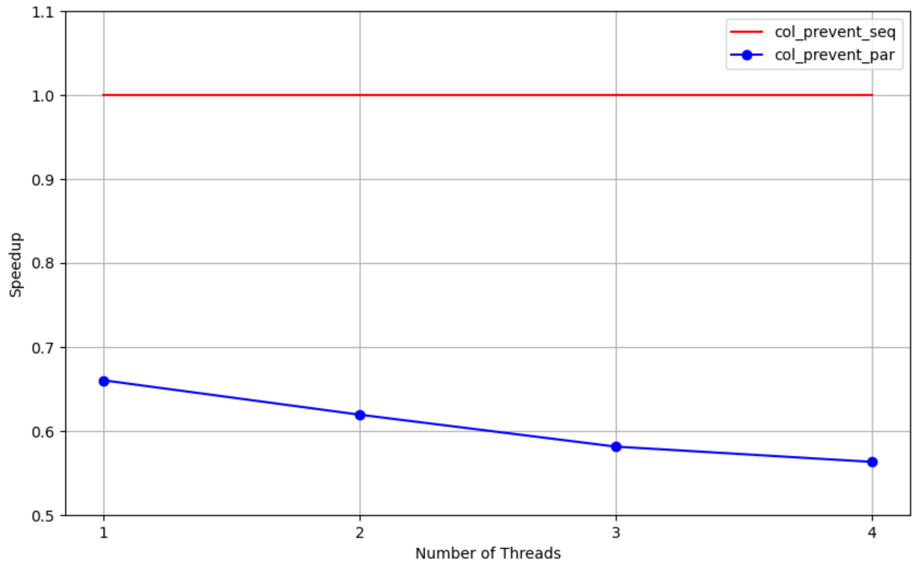


Figure 4: Speedup for worst case scenario

A Apendix A: Test Machine Specifications

We are using an Intel Core i7-10750H CPU with x86_64 architecture, 1 socket, 6 cores per socket and 2 threads per socket (this is taking into account hyperthreading). This CPU has 12 MB L3 cache, 1.5 MB L2 cache, 192 KB data L1 and 192 KB instruction L1.

The result of running `lscpu` is the following:

```
Architecture: x86_64
  CPU op-mode(s): 32-bit, 64-bit
  Address sizes: 39 bits physical, 48 bits virtual
  Byte Order: Little Endian
CPU(s): 12
  On-line CPU(s) list: 0-11
Vendor ID: GenuineIntel
  Model name: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    CPU family: 6
    Model: 165
    Thread(s) per core: 2
    Core(s) per socket: 6
    Socket(s): 1
    Stepping: 2
    CPU(s) scaling MHz: 85%
    CPU max MHz: 5000,0000
    CPU min MHz: 800,0000
    BogoMIPS: 5199,98
    Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
        ↪ clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
        ↪  constant_tsc art arch_perfmon pebs
                        bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni
                            ↪ pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg
                            ↪ fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
                            ↪ popcnt tsc_deadli
                        ne_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
                            ↪ cpuid_fault epb ssbd ibrs ibpb stibp ibrs_enhanced tpr_
                            ↪ shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust
                            ↪ bmi1 avx2 sm
                        ep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt
                            ↪ xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts
                            ↪ hwp hwp_notify hwp_act_window hwp_epp vnmi pku ospke md_
                            ↪ clear flush_
                        l1d arch_capabilities
Virtualization features:
  Virtualization: VT-x
Caches (sum of all):
  L1d: 192 KiB (6 instances)
  L1i: 192 KiB (6 instances)
  L2: 1,5 MiB (6 instances)
  L3: 12 MiB (1 instance)
NUMA:
  NUMA node(s): 1
  NUMA node0 CPU(s): 0-11
Vulnerabilities:
  Gather data sampling: Mitigation; Microcode
  Itlb multihit: KVM: Mitigation: VMX disabled
  L1tf: Not affected
  Mds: Not affected
```

```
Meltdown: Not affected
Mmio stale data: Mitigation; Clear CPU buffers; SMT vulnerable
Reg file data sampling: Not affected
Retbleed: Mitigation; Enhanced IBRS
Spec rstack overflow: Not affected
Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl
Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Spectre v2: Mitigation; Enhanced / Automatic IBRS; IBPB conditional; RSB filling;
    ↪ PBRSB-eIBRS SW sequence; BHI SW loop, KVM SW loop
Srbds: Mitigation; Microcode
Tsx async abort: Not affected
```

This machine has 32 GB RAM and a 2060 RTX Mobile GPU.

B Running the program

The binary can be run with the following arguments (after running `make` in the root of the project):

```
Usage: ./demo/demo [-h|--help] [--timing-mode] [--implementation=(CUDA,vector,OMP,
    ↪ pthreads,sequential,col_prevent_seq,col_prevent_par)] [-n(NUM_THREADS)]
    ↪ SCENARIO_FILE
e.g.: ./demo/demo --timing-mode --implementation=col_prevent_par -n4 demo/commute_
    ↪ 200000.xml
(NUM_THREADS must be > 0 and < 16)
```

More information about the options can be displayed by running `./demo/demo -h`.

Programación paralela y computación de altas prestaciones

Task 4

Students: Alejandro Carmona Martínez, Álvaro Rubira García

Mails: alejandro.carmonam@um.es, alvaro.r.g@um.es

Academic year: 2024/25

Date: 25/11/2024

Contents

1 Evaluation

We are using 4 separate kernels for the parallelized heatmap in GPU: one for fading all the cells, one for updating the heatmap according to the agents' desired positions, one for normalizing and scaling the heatmap and a final one for applying gaussian blur. In figure 1 we show the average time per tick spent on each kernel and memory copy operation as well as the average total time for GPU heatmap computation and sequential heatmap computation. In total, the sequential heatmap implementation takes 554.64 seconds to execute for 10000 ticks of `scenario.xml` with the parallel collision prevention implementation (4 threads are used for collision detection), and the parallel heatmap implementation takes 156.38 seconds with the same collision detection. We inspect the time spent on each kernel in more detail in section 2.2.

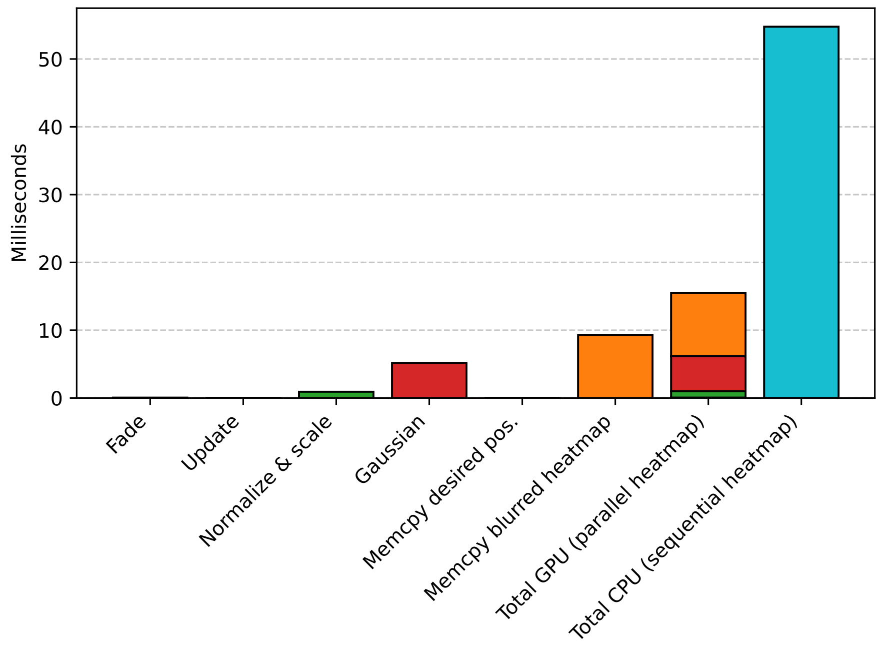


Figure 1: GPU kernels vs. sequential heatmap

2 Questions

2.1 Describe the memory access patterns for each of the three heatmap creation steps. How well does the GPU handle these access patterns?

2.1.1 Fade

```
#define BLOCK_LENGTH 16
#define THREADS_PER_BLOCK (BLOCK_LENGTH * BLOCK_LENGTH)
```

Listing 1: Kernel constants

```
__global__ void fadeHeatmap(int* heatmap, int elems) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < elems) {
    heatmap[idx] *= 0.80;
  }
}
```

Listing 2: `fadeHeatmap` kernel (heatmap creation)

```
Thread 0 -> heatmap[0]
Thread 1 -> heatmap[1]
Thread 2 -> heatmap[2]
...
Thread 31 -> heatmap[31]
```

Listing 3: `fadeHeatmap` kernel access pattern

Memory Access Pattern

In this kernel we create one thread per cell of the scenario, with blocks of 256 threads (this thread block yielded the best results in our testing). We use the same number of threads per block for all kernels. For this kernel we use one-dimensional thread blocks.

- Each thread accesses a single element in the heatmap array.
- No shared memory usage due to each thread only using the data they loaded.
- Access is coalesced since adjacent threads access adjacent memory locations, allowing memory transactions to be combined.
- No bank conflicts as each thread writes to different locations.

This pattern is simple and can be handled very well by the GPU due to the coalesced accesses and linear pattern, taking advantage of good cache utilization due to spatial locality. Shared memory is not useful here as threads from the same thread block do not need to share any data.

2.1.2 Update

```
__global__ void updateHeatmap(const int* pos_x, const int* pos_y, int* heatmap, int n_agents,
                              int rows, int cols) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (idx < n_agents) {
    int x = pos_x[idx];
    int y = pos_y[idx];

    if (x < 0 || x >= cols || y < 0 || y >= rows) {
      return;
    }

    atomicAdd(&heatmap[y * cols + x], 40);
  }
}
```

Listing 4: `updateHeatmap` kernel (heatmap creation)

Memory Access Pattern

We create one thread per agent in the scenario, with one-dimensional thread blocks.

- Threads access 3 arrays: `pos_x`, `pos_y`, and the heatmap.
- First two arrays (`pos_x`, `pos_y`) have coalesced reads.
- Heatmap access is scattered/random based on position values.
- Atomic writes to heatmap are necessary to avoid conflicts, as multiple threads might write to the same location.

For the update kernel, memory accesses are not fully coalesced as writes may follow scattered access as the memory accesses are based on the values from the position arrays. This alone causes memory divergence and poor cache usage, but the use of atomics hurts performance too due to serialization. Overall, this kernel is not really a good fit for GPU acceleration.

2.1.3 Normalize and scale

```
__global__ void normScaleHeatmap(int* orig_heatmap, int* scaled_heatmap, int elems) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (idx < elems) {
    int value = min(orig_heatmap[idx], 255);
    orig_heatmap[idx] = value;

    int x = idx % LENGTH;
    int y = idx / LENGTH;

    for (int cellY = 0; cellY < CELL_SIZE; cellY++) {
      for (int cellX = 0; cellX < CELL_SIZE; cellX++) {
        scaled_heatmap[(y * CELL_SIZE + cellY) * SCALED_LENGTH + (x * CELL_SIZE + cellX
            ↪ )] = value;
      }
    }
  }
}
```

Listing 5: `normScaleHeatmap` kernel (heatmap scaling)

Memory Access Pattern

We use one-dimentional thread blocks, with one thread per cell of the heatmap before scaling:

- One coalesced read and write per thread in `orig_heatmap`.
- For `scaled_heatmap`, each thread writes to `CELLSIZE` $\times$ `CELLSIZE` locations.
- Writes to `scaled_heatmap` are strided with stride $=$ `CELLSIZE`.

While the normalization part is actually coalesced for both reads and writes, the scaling is not coalesced but strided. Although the writes in the scaling part may be performed simultaneously since a certain position is only accessed by one thread, these may not be as performant. In the end, this kernel is not a perfect fit for GPU acceleration, but it is not a bad one either since this strided access would occur in CPU too.

2.1.4 Gaussian blur

Before delving into the Gaussian blur kernel and its memory patterns, it is important to clarify an implementation concept that affects the memory pattern itself for any 2D convolution.

For a 2D input matrix, a 2D convolution applies a filter to a position i by multiplying its neighbouring positions with filter weights and accumulating the results into i.

As seen in Fig. 2, we divide the input into tiles and each tile will be the input matrix of each threadblock. For every threadblock, we will need both the **elements of its tile, the core**, and the elements in a **halo around the input tile**, as these are the needed **neighbouring elements** to perform the compute for the elements closer to the edge of the tile. This is due to the fact that the filter will overlap with elements outside the input tile. The reach of these elements is determined by the filter radius.

In our CUDA implementation, the dimensions of each threadblock match the ones of the input tile, which means that we have fewer threads than elements needed as each thread maps to a position of the input tile, i.e. the core elements. Our approach is to divide the load of these elements (core) from the externally neighbouring ones (halo). We refer to **elements inside the input tile as core elements**, while the **neighbouring elements needed by one tile but outside the tile itself will be halo elements**.

At the beginning of the kernel, the core elements will be loaded into shared memory by each of the threadblocks threads. Meanwhile, the halo elements will be loaded from global memory when needed. Now we proceed with the full kernel implementation:

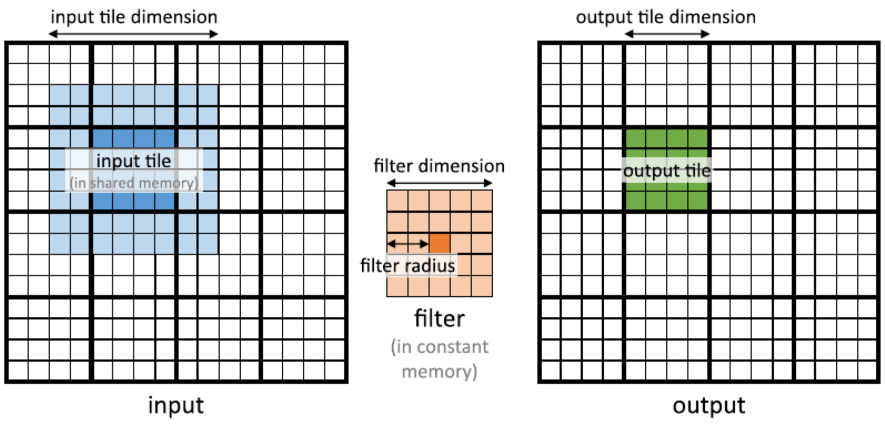


Figure 2: Input tile versus output tile in a 2D convolution. Core Region highlighted in dark blue, Halo Region in clear blue [1]

```
#define WEIGHTSUM 273
#define MASK_RADIUS 2

__constant__ int w[5][5] = {
    {1, 4, 7, 4, 1}, {4, 16, 26, 16, 4}, {7, 26, 41, 26, 7}, {4, 16, 26, 16, 4}, {1, 4, 7, 4, 1}};

__global__ void gaussianBlur(int* scaled_heatmap, int* blurred_heatmap, int rows) {
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

  __shared__ int shared_mem_tile[BLOCK_LENGTH][BLOCK_LENGTH];

  // Load data to shared memory (only load data for threads that will output a value)
  if (col < SCALED_LENGTH && row < rows) {
    shared_mem_tile[threadIdx.y][threadIdx.x] = scaled_heatmap[row * SCALED_LENGTH + col];
  } else {
    shared_mem_tile[threadIdx.y][threadIdx.x] = 0;
  }

  __syncthreads();

  // Calculate the output value
  if (col < SCALED_LENGTH && row < rows) {
    int sum = 0;
    for (int i_row = -MASK_RADIUS; i_row <= MASK_RADIUS; i_row++) {
      for (int i_col = -MASK_RADIUS; i_col <= MASK_RADIUS; i_col++) {
        // Check if cell to read is within the bounds of the shared memory array (tile)
        int shared_index_x = threadIdx.x + i_col;
        int shared_index_y = threadIdx.y + i_row;
        if (shared_index_x >= 0 && shared_index_x < BLOCK_LENGTH && shared_index_y >= 0 &&
            shared_index_y < BLOCK_LENGTH) {
          // 1) Core of the tile ->> Using shared memory
          sum += shared_mem_tile[shared_index_y][shared_index_x] *
                 w[i_row + MASK_RADIUS][i_col + MASK_RADIUS];
        } else {
          // 2) Halo of the tile ->> Using global memory
          int global_index_x = col + i_col;
          int global_index_y = row + i_row;
          // Cells outside scenario are not counted (treated as if they were 0)
          if (global_index_x >= 0 && global_index_x < SCALED_LENGTH && global_index_y >= 0 &&
              global_index_y < rows) {
            sum += scaled_heatmap[(global_index_y)*SCALED_LENGTH + global_index_x] *
                   w[i_row + MASK_RADIUS][i_col + MASK_RADIUS];
          }
        }
      }
    }
    int value = sum / WEIGHTSUM;
    blurred_heatmap[row * SCALED_LENGTH + col] = 0x00FF0000 | value << 24;
  }
}
```

Listing 6: `gaussianBlur` kernel (blur filter)

Memory Access Pattern

In this kernel we create one thread per pixel of the final heatmap, using two-dimensional 16×16 thread blocks. Although this means that only half a warp can coalesce accesses to global memory (as we are using rows that are 16 elements long), we found that this thread block size is faster than 32×32.

- Coalesced reads from `scaled_heatmap` into shared memory as thread (x,y) reads position (x,y) in global memory.
- Each thread block loads `BLOCK_LENGTH` × `BLOCK_LENGTH` elements into shared memory, which are enough for the core tile to be read from shared memory. However, reads from global memory are needed for taking the halo into account.
- Coalesced writes to `blurred_heatmap`.

Finally, as the gaussianBlur kernel performs a 2D convolution, this aligns well with the GPU strengths both computation and memory wise.

Most of the data is computed using shared memory, and while there is a part computed with global memory reads, the penalty from these global memory reads is lessened because the halo region from one block is the core region from another one. This means that another block may very well have loaded the halo values into shared memory and therefore deposited the value into L2 cache, too, which is shared between blocks and allows for faster access than global memory [1].

2.2 Plot and compare the performance of each of the heatmap steps. Discuss and explain the execution times.

For `scenario.xml`, we plot the average execution time of each kernel in figure 3, including memory copy operations in figure 4. We also provide the exact value of the average time for each operation in table 1.

The average kernel time spent copying desired agent positions to the GPU and updating the heatmap is negligible compared to all other operations. This is due to `scenario.xml` having a small number of agents (around 2450) compared to the total number of cells that need to be displayed on screen.

For instance, when fading, normalizing and scaling the heatmap one thread is spawned per cell of the scenario, for a total of $1024 \times 1024 = 2^{20}$ threads, substantially more than the number of agents that need to be taken into account when updating the heatmap. Even if the heatmap update uses atomic operations which are relatively costly, there are so few agents that this kernel execution is dwarfed by the others.

Moreover, the gaussian blur kernel and the memory operation that copies the resulting heatmap image to CPU operate on each pixel of the scenario. There are 25 pixels per cell, so that makes 25×2^{20} pixels in total. This illustrates why the gaussian blur and final copy of the blurred heatmap to CPU take such a long time. In particular, the final copy is moving 4 bytes $\times(25 \times 2^{20})$ pixels $= 100$ MiB from GPU to CPU each tick.

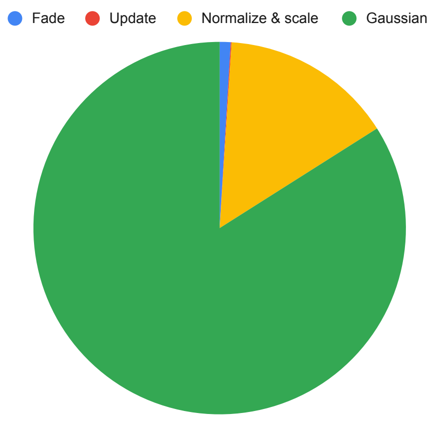


Figure 3: Average ellapsed time per kernel

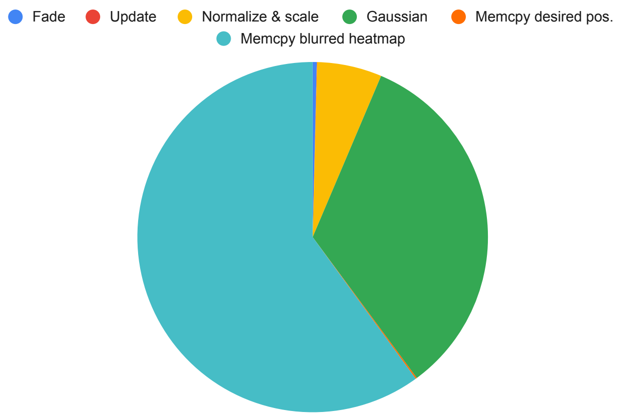


Figure 4: Average ellapsed time per kernel and memory operation

Operation	ms
Fade	0.0583
Update	0.0050
Normalize & scale	0.9239
Gaussian blur	5.1773
Copy desired pos. to GPU	0.0193
Copy blurred heatmap to CPU	9.2771
Total	15.4609

Table 1: Average time per operation

2.3 What speed up do you obtain compared to the sequential CPU version? Given that you use N threads for the kernel, explain why you do not get N times speed up?

For 10000 ticks, we get a $\frac{54.76}{15.46} = 3.54$ times speedup for heatmap calculation (not taking into account time spent on agent movement) when using the GPU implementation over the sequential heatmap. Although we spawn a different number of threads depending on the kernel, for the kernel with the least threads (the one that updates the heatmap with the agents' desired positions) we spawn around 2450. We do not get a 2450 times speedup for several reasons:

- GPU threads are different from CPU threads. They can be seen as slower, simpler versions of CPU threads. The GPU can run thousands of threads at the same time, but each thread's performance is lower than a CPU thread. GPU threads are also scheduled in groups (warps), so they cannot efficiently handle divergent paths inside each group.
- GPU time has to take memory movement operations between CPU and GPU into account. In fact, for this scenario, memory copy operations take up almost two thirds of the total execution time.

2.4 How much data does your implementation copy to shared memory?

As explained in Subsection 2.1, there is only usage of shared data for the gaussian blur kernel as it represents the only case of data sharing between threads of the same threadblock, therefore the only case when shared memory may be of interest.

For this kernel, the whole of the matrix found in `blurred_heatmap` is copied in tiles to shared memory. With every tile being of `BLOCK_LENGTH` (16) size, the dimensions of the tile copied per block is 1024 bytes (1 KiB).

However, not every memory access in the kernel is done through shared memory. Detailed in subsubsection 2.1.4, to perform the convolution both the core tile and the halo are needed, but only the core tile is copied to shared memory.

3 Bonus Assignment Questions

3.1 Describe and motivate what parts of the work you offloaded to the CPU.

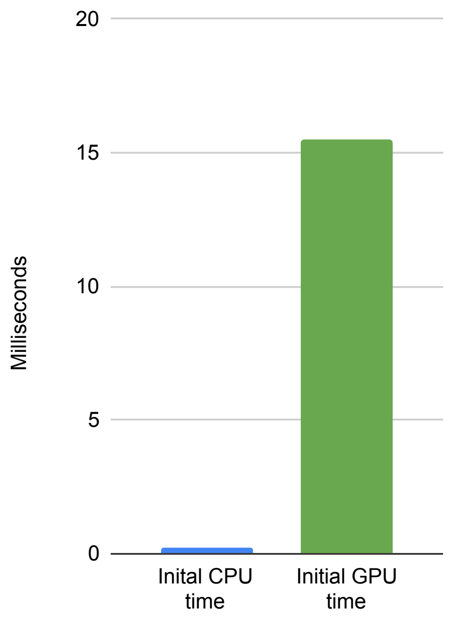
We are distributing some rows of the heatmap to be fully processed on the CPU. This allows us to have fine-grained control over what fraction of the work is assigned to the

CPU and GPU. We found that the number of rows that minimizes differences between the time the CPU and GPU finish each tick (for minimal imbalance) is 832 rows for the GPU and 192 for the CPU. It is important to remark that the number is a multiple of 16 because blocks are dimensioned as 16x16, so to maximize the GPU usage, its workload row count must be a multiple of 16.

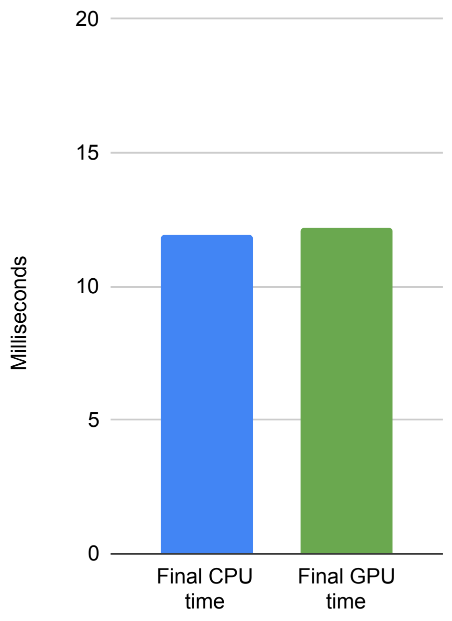
This distribution is specific for `scenario.xml`, using parallel collision prevention with 4 threads (which we found to be the fastest collision prevention algorithm for `scenario.xml`). This configuration can be run with the following command:

```
./demo/demo --implementation=col_prevent_par -n4 --heatmap_het --timing-mode demo/
    ↪ scenario.xml
```

3.2 Using your timestamp measurements, make a graph showing the workload imbalance before and after your adjustments. Additionally, measure the total runtime of the program. What speedup did you obtain? Explain why.



(a) Initial tick time



(b) Tick time after reducing imbalance

Figure 5: Average times for CPU tick vs. GPU tick

After our adjustments, we managed to bring the imbalance down from 98.59% to 2.16%. The final difference can be seen in image 5. After running the whole program for 10000 ticks, we get the speedup pictured in figure 6 when compared against the sequential version. The final version that minimizes imbalance obtains a 4.39 times speedup when compared to the version that uses a sequential heatmap and a 1.24 times speedup when compared to the version that only uses GPU for the heatmap.

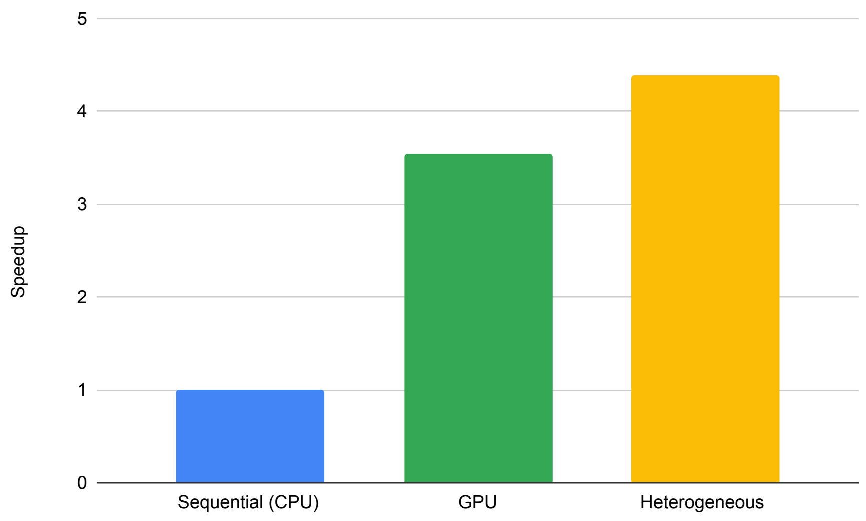


Figure 6: Speedup for 10000 ticks

The speedup we obtain is determined by the two most important operations time-wise, the gaussian blur kernel and the memcpy for the blurred heatmap 4. Combining CPU and GPU has a knock on effect on both, as the GPU has to process less tiles, hence less threadblocks are launched and the Device-to-Host transfer is now smaller.

As we already know, CPU computing speeds for 2D convolutions are poor when compared to GPU 2.3, so the CPU workload has to be selected accordingly, limiting simultaneous heatmap computations. This workload has to be big enough, so the memory transfer are sensibly faster but smaller enough so we do not increase GPU idle time, therefore obtaining the same execution time but less throughput. Also, the GPU workload has to be a multiple of 16 (threadblock size) to maximize threadblock usage.

With an 81-19 load balance in favour of the GPU, we are able to obtain the aforementioned 1.24x rooted in the time reduction on the gaussian blur kernel and the memcpy as shown on table 2.

Operation	**ms**
Fade	0.0454
Update	0.0053
Normalize & scale	0.6170
Gaussian blur	3.5152
Copy desired pos. to GPU	0.0234
Copy blurred heatmap to CPU	7.8504
Total	12.0593

Table 2: Average time per operation on the heterogenous heatmap computations

Heterogeneous gaussian blur performance achieves a 48% speedup from its all-GPU counterpart, which is bigger than the workload decrease percentually. Such an outcome indicates a better usage of GPU resources with a lesser amount of thread blocks. This may be the case if the number of threadblocks is not a multiple of the SM number [2] or if this is considerably smaller than the threadblock number. Both happen for the thread block count both when the heatmap is calculated entirely on GPU or CPU/GPU, but is less noticeable in the second scenario.

Furthermore, *Copy blurred heatmap to CPU* is faster by an 18%, which matches the decrease in data transferred from GPU to CPU. And as per the Amdahl law, its speed-up results determine the speed-up of the whole program, which mostly does with a 24% total speed-up even though other operations becomes even faster.

In the end, the speed-up for the combination of CPU/GPU is mostly affected by the workload differences, specially if memory transfers dominate the GPU time. However, it is important to understand that depending on our GPU capabilities, the decrease or increase on workload may not lead to faster or slower times as the GPU needs a certain amount of work to take advantage of its throughput but too much may lead to underperformance due to quantization and the number of available SM. In our case, the slightly smaller workload seems to better suit our GPU computing performance.

A Apendix A: Test Machine Specifications

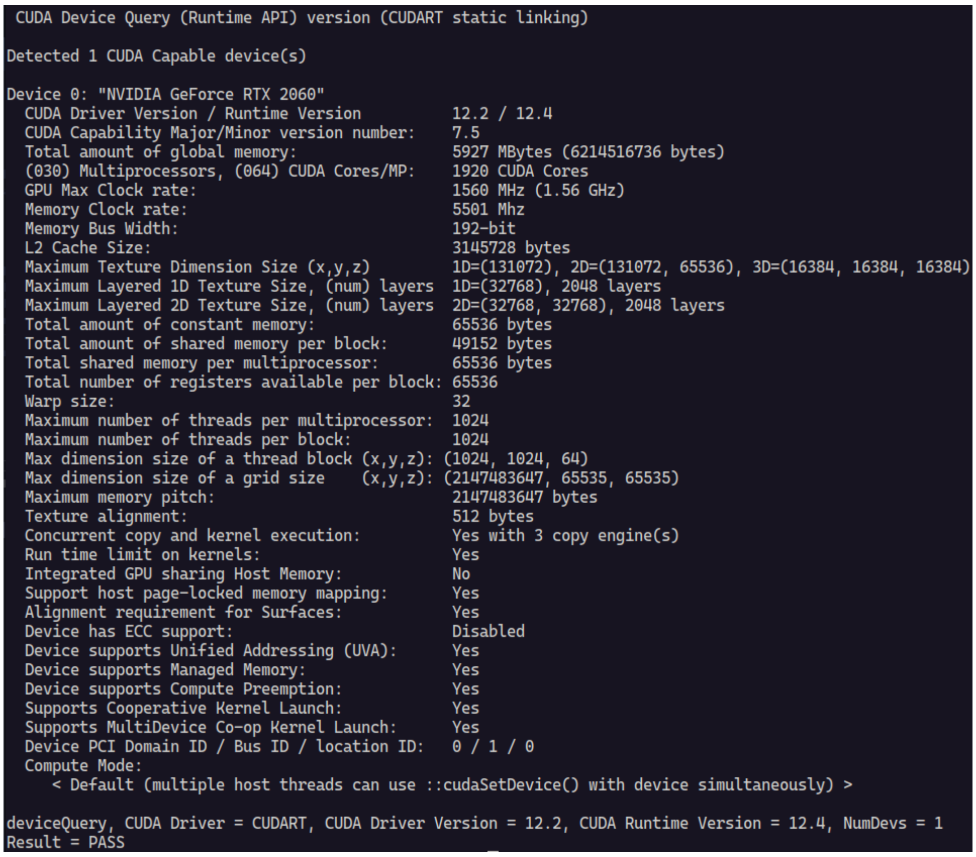
We are using an Intel Core i7-10750H CPU with x86_64 architecture, 1 socket, 6 cores per socket and 2 threads per socket (this is taking into account hyperthreading). This CPU has 12 MB L3 cache, 1.5 MB L2 cache, 192 KB data L1 and 192 KB instruction L1.

The result of running `lscpu` is the following:

```
Architecture: x86_64
  CPU op-mode(s): 32-bit, 64-bit
  Address sizes: 39 bits physical, 48 bits virtual
  Byte Order: Little Endian
CPU(s): 12
  On-line CPU(s) list: 0-11
Vendor ID: GenuineIntel
  Model name: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    CPU family: 6
    Model: 165
    Thread(s) per core: 2
    Core(s) per socket: 6
    Socket(s): 1
    Stepping: 2
    CPU(s) scaling MHz: 85%
    CPU max MHz: 5000,0000
    CPU min MHz: 800,0000
    BogoMIPS: 5199,98
    Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
        ↪ clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
        ↪  constant_tsc art arch_perfmon pebs
                          bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni
                              ↪ pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg
                              ↪ fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
                              ↪ popcnt tsc_deadli
                          ne_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
                              ↪ cpuid_fault epb ssbd ibrs ibpb stibp ibrs_enhanced tpr_
                              ↪ shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust
                              ↪ bmi1 avx2 sm
                          ep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt
                              ↪ xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts
                              ↪ hwp hwp_notify hwp_act_window hwp_epp vnmi pku ospke md_
                              ↪ clear flush_
                          l1d arch_capabilities
Virtualization features:
  Virtualization: VT-x
Caches (sum of all):
  L1d: 192 KiB (6 instances)
  L1i: 192 KiB (6 instances)
  L2: 1,5 MiB (6 instances)
  L3: 12 MiB (1 instance)
NUMA:
  NUMA node(s): 1
  NUMA node0 CPU(s): 0-11
Vulnerabilities:
  Gather data sampling: Mitigation; Microcode
  Itlb multihit: KVM: Mitigation: VMX disabled
  L1tf: Not affected
  Mds: Not affected
```

```
Meltdown: Not affected
Mmio stale data: Mitigation; Clear CPU buffers; SMT vulnerable
Reg file data sampling: Not affected
Retbleed: Mitigation; Enhanced IBRS
Spec rstack overflow: Not affected
Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl
Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Spectre v2: Mitigation; Enhanced / Automatic IBRS; IBPB conditional; RSB filling;
    ↪ PBRSB-eIBRS SW sequence; BHI SW loop, KVM SW loop
Srbds: Mitigation; Microcode
Tsx async abort: Not affected
```

This machine has 32 GB RAM and an NVIDIA GeForce RTX 2060 Mobile with the following specs:



```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce RTX 2060"
  CUDA Driver Version / Runtime Version          12.2 / 12.4
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 5927 MBytes (6214516736 bytes)
  (030) Multiprocessors, (064) CUDA Cores/MP:    1920 CUDA Cores
  GPU Max Clock rate:                            1560 MHz (1.56 GHz)
  Memory Clock rate:                             5501 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 3145728 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.2, CUDA Runtime Version = 12.4, NumDevs = 1
Result = PASS
```

Figure 7: GPU specifications

B Appendix B: Running the program

The binary can be run with the following arguments (after running `make` in the root of the project):

```
Usage: ./demo/demo [-h|--help] [--timing-mode] [--heatmap_(seq,par,het)] [--
    ↪ implementation=(CUDA,vector,OMP,pthreads,sequential,col_prevent_seq,col_prevent
    ↪ _par)] [-n(NUM_THREADS)] SCENARIO_FILE
```

```

e.g.: ./demo/demo --timing-mode --heatmap_het --implementation=col_prevent_par -n4
    ↪ demo/commute_200000.xml

--timing-mode: Reduce output to the terminal and don't show graphic representation.
--heatmap_(seq,par,het): Selects a heatmap implementation. If this option isn't
    ↪ specified, heatmap is not shown. Options are:
  - heatmap_seq: Heatmap is computed by a single CPU thread.
  - heatmap_par: Heatmap is computed in the GPU.
  - heatmap_het: Heatmap workload is divided between CPU (single thread) and GPU.
--implementation=: Selects an implementation for moving agents. If this option isn't
    ↪ specified, sequential is the default implementation. Options are:
  - sequential: A single CPU thread is used.
  - OMP: A number of threads specified by the -n option are used with OpenMP.
  - pthreads: Same as OMP but threads are managed with pthreads.
  - vector: Same as OMP option, but each thread uses SIMD instructions to process 4
      ↪ agents at a time.
  - CUDA: Agent movement is processed in the GPU.
  - col_prevent_seq: Same as sequential but with collision avoidance between agents.
  - col_prevent_par: Divides the scenario in 4 regions, allowing parallel agent
      ↪ movement (with collision avoidance) calculation with multiple threads. The
      ↪ number of threads to use can be specified with -n, but the fastest value is
      ↪ very likely to be -n4.
-n(NUM_THREADS): Sets number of threads for OMP, pthreads, vector and col_prevent_par.
    ↪  NUM_THREADS must be > 0 and <= 16.
```

More information about the options can be displayed by running `./demo/demo -h`.

References

[1] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. Page 169. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123814723.

[2] NVIDIA. *Deep Learning Performance Guide: Convolutional Networks*. `https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html`. Accessed: 2024-11-20. 2024.