h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbmd
FACHBEREICH MEDIA

Bachelor of Arts Animation and Game
2022

GPU Accelerated Wavefunction Collapse using Compute Shaders

Lasse Foster
Matriculation Nr. 762434

First Supervisor: Prof. Dr.-Ing. Martin Leissler
Second Supervisor: Prof. Dipl.-Inf. Stephan Jacob

BA ANIMATION & GAME | Eigenhändigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Soweit ich auf fremde Materialien (Texte, Grafiken, Diagramme, Bildmaterial, Codes, Sound) oder Gedankengänge zurückgegriffen habe, enthalten meine Ausführungen vollständige und eindeutige Verweise auf die Urheber und Quellen. Dies gilt auch für Quellen, die ich selbst für andere Veröffentlichungen oder Prüfungen erstellt habe.

Alle weiteren Inhalte der vorgelegten Arbeit stammen von mir im urheberrechtlichen Sinn soweit keine Verweise und Zitate erfolgen.

Mir ist bekannt, dass ein Täuschungsversuch vorliegt, wenn die vorstehende Erklärung sich als unrichtig erweist.

Darmstadt | 12.02.22 | Foster, Lasse | Lasse Foster

Ort | Datum | Name, Vorname | Unterschrift

BA ANIMATION & GAME | Erklärung zur Archivierung

Bitte Zutreffendes ankreuzen:

- [x] Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek der Hochschule Darmstadt bin ich einverstanden.

- [ ] Mit der Archivierung der gedruckten Abschlussarbeit in der Bibliothek der Hochschule Darmstadt bin ich nicht einverstanden.
  Begründung:

  - [ ] Die Arbeit ist unterliegt der Geheimhaltung, da sie in einem Unternehmen durchgeführt wurde und ihr Inhalt ausdrücklich durch dieses gesperrt ist.

  - [ ] Persönliche Gründe

Darmstadt | 12.02.22 | Foster, Lasse | Lasse Foster

Ort | Datum | Name, Vorname | Unterschrift

BA ANIMATION & GAME | Declaration of Authorship

I hereby declare that the thesis I am submitting is entirely my own work except where otherwise indicated.

I have clearly indicated the presence of all material from other sources I have quoted, paraphrased or used in any way. This includes texts, presentations, graphics, diagrams, still or moving images programme code and sounds.

I have clearly indicated the presence of all material that I have created for other examinations or published elsewhere. I understand the rules and regulations regarding academic good practice and plagiarism.

Darmstadt | 12.02.22 | Foster, Lasse | Lasse Foster

Place | Date | Last Name, First Name | Signature

BA ANIMATION & GAME | Archiving Declaration

Mark with a cross where applicable:

- [x] I hereby give my consent to the storage of a printed copy of my thesis in the Library of Darmstadt University of Applied Sciences

- [ ] I do not give my consent to the storage of a printed copy of my thesis in the Library of Darmstadt University of Applied Sciences.

  Reason:

  - [ ] My thesis is confidential, as it was written in cooperation with a company and is subject to the restrictions of a non disclosure agreement.

  - [ ] Personal reasons

Darmstadt | 12.02.22 | Foster, Lasse | Lasse Foster

Place | Date | Last Name, First Name | Signature

Abstract

This work demonstrates and analyses multiple implementations of a parallel GPU Wavefunction Collapse solver, comparing them to current state-of-the-art sequential and parallel CPU implementations.

Wavefunction Collapse is a procedural generation algorithm that generates locally similar output to the input data. It can generate everything from terrains to buildings or even more abstract structures like text and saw wide adoption throughout the Indi-game-development scene. However, wavefunction Collapse does not scale well with an increase in output dimensions and patterns, limiting its use for large-scale applications.
Nevertheless, the current state-of-the-art approaches to parallelization show significant improvements to the execution time of part of the Wavefunction Collapse algorithm, improving performance by significantly over sequential implementations of the algorithm.

Based on these findings, a baseline of current general-purpose GPU compute functionality is established, discussing the different fields of application for compute environments like CUDA and compute shaders. I conclude that using compute shaders is the best solution to address the compute capabilities of GPUs due to their readily available pipelines in almost all game engines.

Six different solvers were implemented and tested, three CPU- and three GPU-solvers. The first GPU solver is based heavily on the previous work on parallel Wavefunction Collapse, only porting the propagation step to the GPU. In contrast, the other two GPU solvers use a fully compute shader based pipeline to minimize memory transfer times. One of the fully compute shader based solvers, also uses compute-buffers to schedule its graphics commands more efficiently.
The CPU solvers serve as a baseline for the GPU solvers and are tightly based on the previous work on parallelization of the propagation step. While one of the CPU solvers is a direct port of the method described in that work using a queue to track its open cells and solve open nodes in parallel, the second parallel CPU solver uses a hash-set for work tracking that reduces the number of thread creations, intending to reduce overhead and improve on the previous work.

The result show significantly worse execution times for any parallel implementation of the Wavefunction Collapse algorithm. The GPU solvers perform hundreds of times worse than the sequential CPU implementations. Moreover, while the parallel CPU solvers perform significantly better than the GPU solvers, none of the tests show better performance than the sequential solver. This difference in performance and the discrepancy to the previous work on parallel Wavefunction Collapse is the significantly improved performance and scaling of modern sequential implementations of the algorithm, compared to those that previous work based itself on.

The bad performance of the parallel implementations is primarily due to the bad parallelization properties of the algorithm, memory bottlenecks that affect the GPU solvers especially, and the overhead of API calls and thread creations. The results show that GPU implementations profit from minimal memory transfer operations and the use of compute-buffers as significant performance improvements were measured for the solvers that utilized these strategies.

It is concluded that the use of GPU compute shaders and multithreaded CPU implementations of the Wavefunction Collapse algorithm, in their current form, does not yield any benefit over the modern and optimized sequential version of the algorithm. Nevertheless, research on the topic of SIMD-based optimizations of the algorithm and further optimizations of the parallel code, since the potential for additional performance increases is seen in those areas.

Zusammenfassung

Diese Arbeit demonstriert und analysiert mehrere Implementierungen eines parallelen GPU Wavefunction Collapse Solvers und vergleicht sie mit aktuellen sequenziellen und parallelen CPU-Implementierungen.

Wavefunction Collapse ist ein prozeduraler Generierungsalgorithmus, der lokal ähnliche Ausgaben aus Eingabedaten erzeugt. Er kann alles von Terrains, über Gebäude, bis hin zu abstrakteren Strukturen wie Text generieren und fand in der Indi Spieleentwicklungsszene breite Anwendung. Allerdings skaliert Wavefunction Collapse nicht gut bei wachsenden Ausgabedimensionen und einer höheren anzahl an genuzten Mustern, was seine Verwendung für groß angelegte Anwendungen einschränkt. Nichtsdestotrotz zeigen die aktuellen State-of-the-Art-Ansätze zur Parallelisierung signifikante Verbesserungen der Ausführungszeit eines Teils des Wavefunction-Collapse-Algorithmus, wodurch die Leistung im Vergleich zu sequenziellen Implementierungen des Algorithmus deutlich verbessert wird.

Auf der Grundlage dieser Ergebnisse wird eine Basis für die derzeitige allgemeine GPU-Rechenfunktionalität geschaffen, wobei die verschiedenen Anwendungsbereiche für Rechenumgebungen wie CUDA und Compute Shader erörtert werden. Ich komme zu dem Schluss, dass die Verwendung von Compute-Shadern die beste Lösung ist um die Rechenkapazitäten von GPUs zu nutzen, da ihre Pipelines in fast allen Spiele-Engines bereits verfügbar sind.

Es wurden sechs verschiedene Solver implementiert und getestet, drei CPU- und drei GPU-Solver. Der erste GPU-Solver basiert stark auf der früheren Arbeit über parallele Wavefunction Collapse, wobei nur der Propagationsschritt auf die GPU portiert wurde. Im Gegensatz dazu verwenden die beiden anderen GPU-Solver eine vollständig auf Compute Shader basierende Pipeline, um die Speichertransferzeiten zu minimieren. Einer der vollständig auf Compute Shader basierenden Solver verwendet auch Compute-Buffer, um seine Grafikbefehle effizienter auszuführen.
Die CPU-Solver dienen als Grundlage für die GPU-Solver und sind eng an die frühere Arbeit zur Parallelisierung des Propagierungsschritts angelehnt. Während einer der CPU-Solver eine direkte Portierung der in dieser Arbeit beschriebenen Methode ist und eine Queue verwendet um ihre offenen Zellen zu verfolgen und offene Knoten parallel zu lösen, verwendet der zweite parallele CPU-Solver ein Hash-Set für die Arbeitsverfolgung. Dieses soll die Anzahl der Thread-Erstellungen reduzieren, mit der Absicht den mit Thread-Erstellungen einhergehenden Overhead zu reduzieren und die vorherige Arbeit zu verbessern.

Das Ergebnis zeigt signifikant schlechteren Ausführungszeiten für eine parallele Implementierung des Wavefunction Collapse-Algorithmus. Die GPU-Solver schneiden hunderte Male schlechter ab, als die sequentiellen CPU-Implementierungen. Und während die parallelen CPU-Solver deutlich besser abschneiden als die GPU-Solver, zeigt keiner der Tests eine bessere Leistung als der sequentielle Solver.
Dieser Leistungsunterschied und die Diskrepanz zu früheren Arbeiten zum parallelen Wavefunction Collapse, ist auf die deutlich verbesserte Leistung und Skalierung moderner sequentieller Implementierungen des Algorithmus im Vergleich zu denen zurückzuführen, auf denen frühere Arbeiten basierten.

Die schlechte Leistung der parallelen Implementierungen ist, in erster Linie, auf die schlechten Parallelisierungseigenschaften des Algorithmus, Speicherengpässe, die vor allem die GPU-Solver betreffen, und den Overhead von API-Aufrufen und Thread-Erstellungen zurückzuführen. Die Ergebnisse zeigen, dass GPU-Implementierungen von minimalen Speichertransferoperationen und der Verwendung von Rechenpuffern profitieren, da für die Solver, die diese Strategien verwenden, signifikante Leistungsverbesserungen gemessen wurden.

Die Schlussfolgerung lautet, dass die Verwendung von GPU-Compute-Shadern und

Multithreading-CPU-Implementierungen des Wavefunction-Collapse-Algorithmus in ihrer derzeitigen Form, keinen Vorteil gegenüber der modernen und optimierten sequentiellen Version des Algorithmus bringt. Nichtsdestotrotz sollte an dem Thema SIMD-basierte Optimierungen des Algorithmus und weitere Optimierungen des parallelen Codes geforscht werden, da in diesen Bereichen das Potenzial für zusätzliche Leistungssteigerungen gesehen wird.

List of Figures

List Of Abbreviations

WFC	Wavefunction Collapse
GPU	Graphics Processing Unit
CPU	Central Processing Unit
GPGPU	General-Purpose Graphics Processing Unit
UI	User Interface
GUI	Graphical User Interface
API	Application Programming Interface
JIT	Just-In-Time
AOT	Ahead-Of-Time
SIMT	Single Instruction, Multiple Threads
SIMD	Single Instruction, Multiple Data

Contents

1 Introduction

With an increase in computation power over the years, the complexity and scale of most AAA games increased along with consumer expectations. A major contributing factor that made this trend possible in the first place was the shift from manually produced game content and worlds to procedural generation and tooling, empowering smaller teams to deliver the same results that required large teams before.
Procedural tools are used to generate everything from terrains and buildings to weather systems, visual effects, and animations, offering a system of controls and constraints that deliver predictable and consistent results when applied to a large variety of assets.

Traditionally all areas of generation require a different and ideally use-case specialized tool-set and algorithm, ranging from noise-based height map generation for landscapes to mesh-based generation algorithms used to generate cities in games like *Assassin's Creed Syndicate*[1].

While these tools allow artists to create high fidelity assets with little effort and at high speeds, they are specific to the area they are used in, potentially requiring significant alterations to their underlying systems for use in another context. This process is labor-intensive, complicated and costly, making a more generally applicable, easy to tweak and adaptable algorithm desirable.

One such algorithm, promising wide adaptability, diverse results from simple inputs and an easy-to-use system of constraints is WFC, a generative algorithm developed by Maxim Gumin[13]. Since it was first published in 2016, WFC has seen wide adoption throughout the Indie game development community, finding applications for everything from traversable terrain[24], to infinite city generation[16] and has seen a lot of research aiming to expand its feature set.
More information on modern uses of WFC for terrain and terrain feature generation, as well as a more in depth comparison to other modern procedural generation algorithms, can be found in my report on *Terrain Feature Generation Using Wavefunction Collapse*[12].

Before we look at why the WFC algorithm could benefit from Graphics Processing Unit (GPU) acceleration, we first have to understand what the WFC algorithm is and how it works.

1.1 Wavefunction Collapse

WFC is a procedural generation algorithm which, in its original implementation by Maxim Gumin, is used to create bitmaps that are locally similar to the provided input bitmap[13].
It is based an algorithm called Model Synthesis[21] which's constrained system works basically the same, with the main difference of the two methods being that they pick the next collapseable node using a different method[20].

It allows to easily create large bitmaps with only small input data-sets with the ability to specify how restricted the randomness of the output will be.
The name Wavefunction Collapse references to a concept of quantum mechanics which shares conceptual parallels with the way WFC achieves its results. More specifically a possibility space of all possible combinations of patterns is created which, after being observed at one point, collapses further and further until only one result remains for the whole space.

Maxim Gumin originally provided a C# implementation of his algorithm, which is rather difficult to understand, since it isn't backed by a paper explaining the details. There is how ever basic documentation on the inner workings of WFC inside the Repository Read-Me, as well as an analysis by Kath and Smith[15].

For an easier and more modular code read I suggest looking into Fehr Mathieu's implementation called FastWFC[19] since it is quite precisely documented.

1.1.1 Algorithm

According to the original project Read-Me by Gumin[13] there are 5 main steps to the WFC algorithm.

1. Read the input bitmap and count NxN patterns.
   (a) (optional) Augment pattern data with rotations and reflections.
2. Create an array with the dimensions of the output (called "wave" in the source). Each element of this array represents a state of an NxN region in the output. A state of an NxN region is a superposition of NxN patterns of the input with boolean coefficients (so a state of a pixel in the output is a superposition of input colors with real coefficients). False coefficient means that the corresponding pattern is forbidden, true coefficient means that the corresponding pattern is not yet forbidden.
3. Initialize the wave in the completely unobserved state, i.e. with all the boolean coefficients being true.
4. Repeat the following steps:
   (a) Observation:
      i. Find a wave element with the minimal nonzero entropy. If there is no such elements (if all elements have zero or undefined entropy) then break the cycle (4) and go to step (5).
      ii. Collapse this element into a definite state according to its coefficients and the distribution of NxN patterns in the input.
   (b) Propagation: propagate information gained on the previous observation step.
5. By now all the wave elements are either in a completely observed state (all the coefficients except one being zero) or in the contradictory state (all the coefficients being zero). In the first case return the output. In the second case finish the work without returning anything.

The WFC algorithm supports two input modes by default

1. ***Overlapping***:
   This mode is used by the bitmap algorithm and as the name suggests uses overlapping tiles of

size NxN that are extracted from the input bitmap similar to how is shown in figure 1.
It is fully automatic and doesn't need input from the user to define it's constraints as they are described by neighbouring pixels of each pattern.

Figure 1: WFC Bitmap Sampling Example[13]

2. ***Tiled***:

   This mode can be used for input patterns of size 1x1, which can be treated as a tile.
   In tiled mode all constraints must be defined beforehand and by hand or by another algorithm that determines possible neighbours of each tile. This mode is much simpler conceptually then the overlapping model and is faster to solve using older versions of the algorithm. An example input tile-set and generated output of this method can be seen in figure 2.

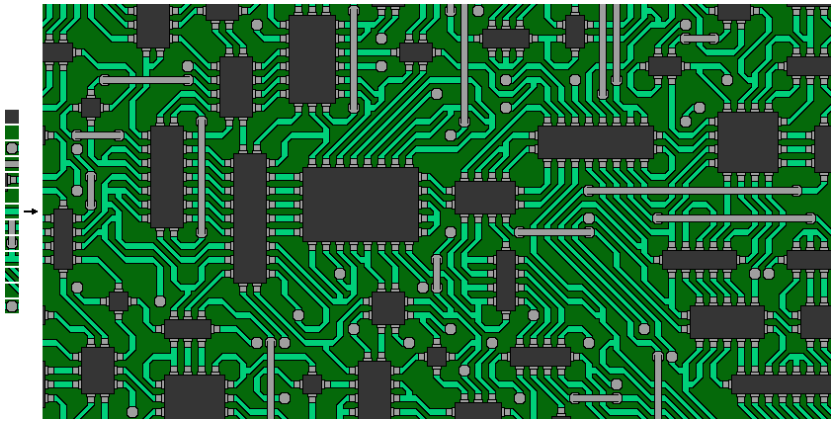


Figure 2: WFC Tilemap Example[13]

WFC has seen wide adoption mostly throughout the Indi game industry, is ported to dozens of different programming languages, expanded to more dimensions and data formats and integrated into industry standard applications like Houdini by SideFX.

1.2 Wavefunction Collapse Performance

WFC is a computationally highly complex algorithm, growing in complexity proportional to the wave's dimensions, as well as the number of patterns used.
While the complexity of parts of the algorithm, such as the observe step, follows a simple relation between both the number of patterns and the dimensions of the wave, the propagation step's complexity follows a lot more complicated relation based on the wave's dimensions, the pattern count and most importantly how restrictive the patterns are.
A pattern-set that reaches a stable wave-state quickly will spend less time performing the propagation step than highly restrictive pattern-sets that cause large amounts of collapses once a pattern is removed.
Additionally, an increased number of tiles will also lead to more iterations of the propagation step since more patterns will need to be removed from the wave, increasing its execution time.
Furthermore, while WFC in its original form is a two-dimensional algorithm, adding more dimensions to the solver is possible, with each added dimension or increase in tiles/patterns, leading to a steep increase in computation time.

An additional factor that can increase the computation time required to achieve a valid result using the WFC algorithm are so-called contradictions. Contradictions happen when the algorithm collapses the nodes so that one or multiple nodes end without a pattern that is compatible with all its neighbors, which is especially likely at large wave dimensions, or when using tile-sets that allow for only a few possible configurations.

An example of a large-scale use of the WFC algorithm is the *Infinite City Demo* by Kleinberg[16], which was done in Unity and features, as the name suggests, endless city generation. Since the WFC algorithm is not designed to allow for endless generation by default, a specialized solver is implemented that generates the city in a circle around the position of the player, generating new structure where needed and discarding distant parts(see fig 3).

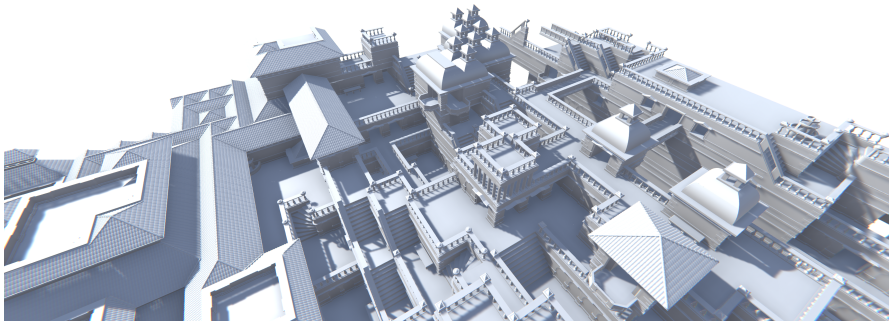


Figure 3: Area generated by Kleineberg's *Infinity City* demo using WFC[16]

On a moderately fast Central Processing Unit (CPU) and moderate player movement speeds, the city can be generated without the player reaching its borders, but at faster movement speeds the generation struggles to keep up, potentially even having to delete the floor beneath the player because of contradictions(see fig. 4). In addition to the limited generation distance, the limitations of the WFC generation speed become apparent quickly during usage.

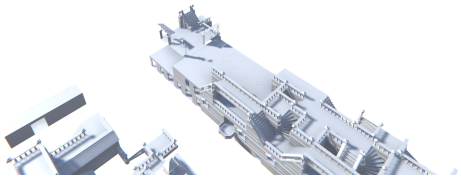


Figure 4: Player Catching up with WFC generated structures due to slow solver performance[16]

The *Infinite City* demo demonstrates WFC's biggest strength, the generation of continuous complex structures, while also revealing its biggest weakness: scaling. Due to repeated contradictions and a generally low generation speed, WFC can only be used to generate a very limited area, requiring much time to generate new parts of the structure.

This is a fundamental problem of WFC, preventing its use for many larger-scale applications, even though many games could profit from its flexible and highly varying results, greatly.

2 State of the Art

2.1 Optimized Wave Function Collapse

As mentioned before, the WFC algorithm's computational complexity grows significantly with an increase of the wave's dimensions as well as an increase in patterns used. While the original WFC implementation by Gumin[13] has seen drastic performance improvements since its initial release, implementing better scaling algorithms developed by the countless other projects portin and improving the algorithm, Gumins choice of language is still a limiting factor, with its C# implementation performing worse than direct ports in C++.
One such implementation is fast-wfc[19], a more modularized implementation with a focus on performance but essentially very similar, making more heavy use of object-oriented programming. Many of the optimizations introduced by this approach were ported to the original C# implementation in 2018, speeding it up by 20x compared to older versions, according to Gumin's commit history[13].

The accumulative improvements of WFC algorithm over the last couple of years resulted in an algorithm that scales significantly more efficiently than initially after its conception in 2016, with natively compiled ports like fast-wfc[19] delivering even better performance.

2.2 Parallel Wavefunction Collapse

Another more complex approach to speeding up the WFC algorithm is parallelization of the algorithm using multi-threading. One of the only works discussing this method is *Parallel Wave Function Collapse* by Orlowski and Lee[17]. They implemented multiple parallel versions of the tiled model for the CPU. Their implementation is based on a C++ implementation of the WFC algorithm by Emil Ernerfeld[10] and their results show a speedup of the WFC algorithm of up to four times compared to Ernerfelds sequential implementation, with varying levels of improvement depending on the parallelization method used.

2.2.1 Approach

Based on the profiling results of Ernerfelds WFC[10] implementation, done by Orlowski And Lee (see fig. 5), only the propagation step was parallelized since it takes up the majority of the computation time. The proposed models were written and tested in C++ and use parts of the boost libraries[3] for work management as well as OpenMP and C++ threads for the parallelization.

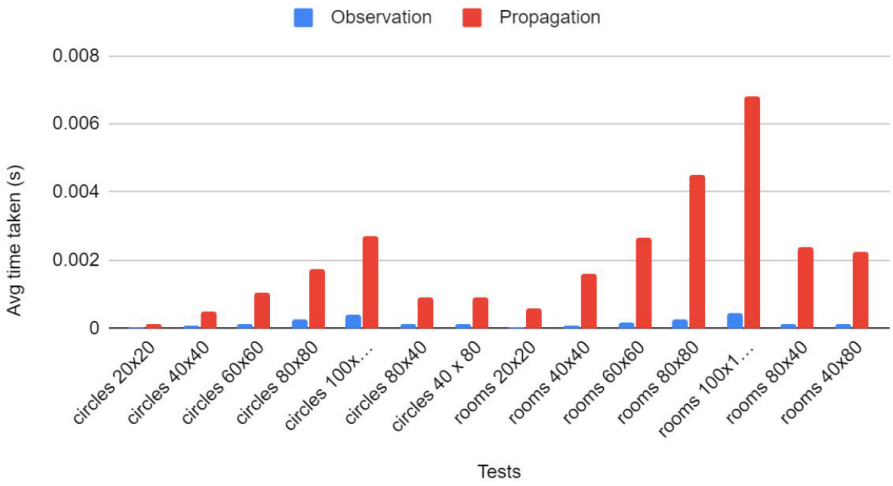


Figure 5: Sequential WFC Profiling[17]

In the C++ WFC implementation by Ernerfeld[10] the propagation step is implemented the follwing way[17]:

1. Mark the observed tile as changed
2. While there is any tile that is still marked as changed:
   - For each tile in the grid:
     - If the tile is marked as changed, check if any of its neighbors have their possibilities limited by the change in the tile. For each neighbor that does, eliminate those possibilities and mark the neighbor as changed. unmark the current tile.

According to Lee and Orlowski, the main difficulties parallelizing the WFC propagator were that:

- nodes directly depend on their neighboring nodes, meaning that the collapse of one node can affect multiple other nodes. However, this limitation is irrelevant since the propagation step only removes nodes from the wave, providing reliable propagation results even if multiple nodes are solved concurrently.
- the total number of iterations needed to collapse each node to their final state cannot be known beforehand.

Based on these limitations, they provide multiple approaches to parallelization, using different strategies for the solver and its work tracking.

2.2.2 Multi-core Parallel with OpenMP

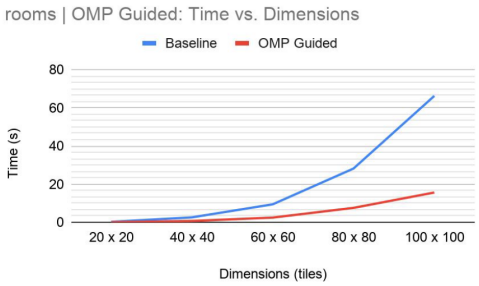
Their initial implementation parallelizes the propagate function using OpenMP; a multi-threading library used to automate the thread creation and assignment for the individual operations of the propagation step.

Each propagation loop works roughly like this:

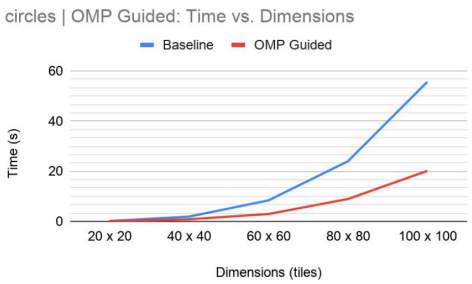
Algorithm 1 OpenMP

```
1:  for every node in the image do                          ▷ This loop is multi-threaded
2:      for each neighbouring node do
3:          if neighbour changed state then
4:              for each pattern possible in node do
5:                  check if pattern still compatible with any neighbour pattern
6:                  if pattern not compatible anymore then
7:                      remove pattern from node
8:                  end if
9:              end for
10:         end if
11:     end for
12: end for
```

Their initial approach of assigning one thread to each row of the wave resulted in speedups of roughly *2-4* times compared to the sequential version. Another test using assignment of one thread per node showed similar results with a slight speedup over the row-wise assignment method (see figure 6a and figure 6b).



(a) [17]



(b) [17]

Figure 6: Performance Test Results Mutli-core Parallel with OpenMP[17]

2.2.3 OpenMP Using Lock-Free Queues

This approach uses the same propagation strategy as the multi-core parallel implementation. However, instead of checking each node in the image during each iteration, a work queue is used to mark every node that can potentially change, drastically reducing the workload caused by unnecessary checks on nodes that would not collapse any further.

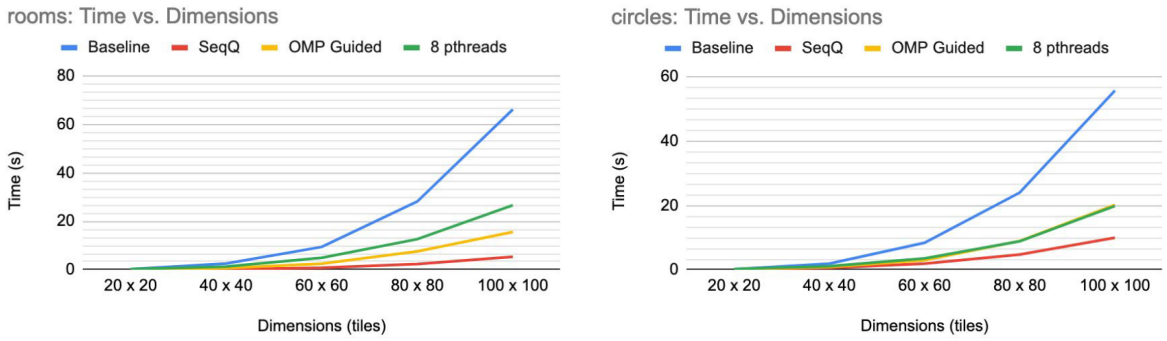
The approach uses four threads that solve the propagation step for the nodes, that are part of the work queue, in parallel. Every time a node is collapsed, its neighboring nodes are added to the work queue and thereby marked as potentially changed. After one iteration finishes, the next four open nodes are de-queued, assigned to the four threads and solved. These steps are then repeated until the algorithm finishes propagation.

The work-queue loop looks roughly like this:

Algorithm 2 OpenMP using Lock-Free Queues

```
1:  while work-queue is not empty do
2:      De-queue four nodes and assign them to the solver
3:      for the four nodes assigned to the solve do                ▷ This loop is multi-threaded
4:          for each neighbouring node do      ▷ Avoid loop/check by keeping track of collapsed node
5:              if neighbour changed state then
6:                  for each pattern possible in node do
7:                      Check if pattern still compatible with any of the neighboring node's patterns
8:                      if pattern not compatible anymore then
9:                          Remove pattern from node and add its neighbouring nodes to the work queue
10:                     end if
11:                 end for
12:             end if
13:         end for
14:     end for
15: end while
```

The resulting algorithm is significantly faster than the initial multi-core parallel implementation (see fig 7).



(a) [17] (b) [17]

Figure 7: Performance test results with OpenMP using lock-free queues[17]

2.2.4 CUDA

In addition to the CPU based parallel implementations provided in their work, Orlowski and Lee mention a CUDA-based implementation using similar strategies for parallelization. They mention promising initial results but do not elaborate further due to inconsistencies between their implementations. They suggested that further research on a General-Purpose Graphics Processing Unit (GPGPU) based approach, such as CUDA, could result in even better performance than is possible using their CPU parallel approach.

2.3 GPU Acceleration

While Orlowski and Lee provide strategies that allow for implementations of the WFC algorithm on parallel hardware, we first need to establish what compute capabilities modern GPUs provide and how the desired use-case will direct the environment, they are used in, before the details of the GPU implementations can be discussed.

A GPU is a specialized processing device that follows the concept of maximized parallelization in order to compute large data blocks as quickly as possible in parallel. As the name suggests, GPUs are very efficient at computing computer graphics and image processing and are separate units from the general-purpose CPU. While CPUs are made to deliver the lowest possible latency, GPUs are made for maximum data throughput[7], with this focus on throughput rather than latency, resulting in worse single-core performance on GPUs[5].

With the introduction of so-called stream processing units GPUs gained the ability to do be used more as general-purpose computation devices. This concept is called GPGPU with this work falling into its research field. It is used for machine learning, linear algebra and crypto mining. Depending on the environment used, it can be executed without a graphics pipeline present, as is needed for fragment

shader-based computation.

The programs executed on graphics cards are called shaders, traditionally describing algorithms to calculate shading, thus giving it its characteristic name. Shaders can be written in C-Style syntax using languages like HLSL or GLSL, supporting a limited subset of the language C. This subset is expanded regularly with new hardware releases and currently supports features like variables, varying loops and function definitions and calls.

2.3.1 Stream Processing

"The concept behind stream processing is a very simple one. By connecting sequential (syntactically) compute kernels via streams so that each compute kernel can compute independently, a model to construct parallel applications emerges."[2]

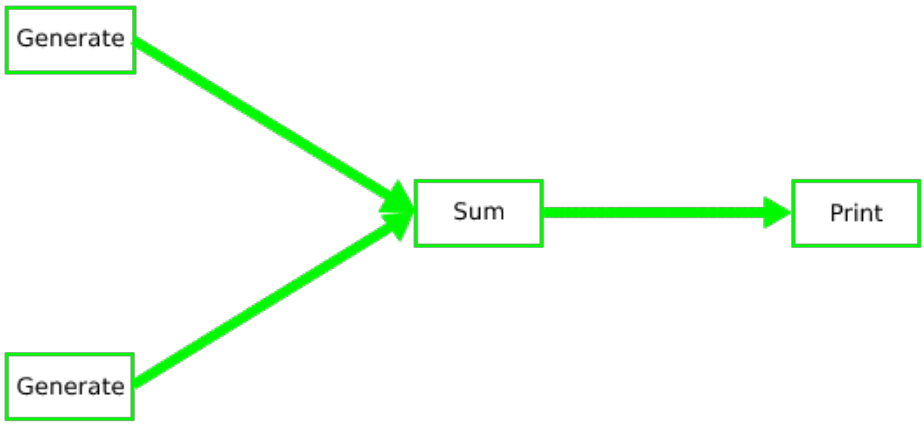


Figure 8: Simple sum application (toy app). The sum kernel takes a stream of random numbers, adds them and streams the output to a print kernel for human consumption. The exact nature of the streams depends on the run-time and hardware. [2]

This concept enables GPUs to schedule parallel tasks in large scales as any computation done on a compute unit is by definition independent of its neighboring units and does not need to access or wait for data from other threads to finish. Therefore no explicit allocation, synchronization, or communication among these units is required, consequently limiting the computations possible.

2.3.1.1 SIMT To execute its code as fast as possible across many threads, GPUs use a so-called Single Instruction, Multiple Threads (SIMT) execution model. The SIMT model executes batches of threads in so-called lockstep, with all threads executing the same instructions at the same time[14]. Using a SIMT execution model, however, limits the GPUs capability of executing branching algorithms, requiring sequential execution of the different execution paths once the control flow of a thread-batch

diverges[14]. This is a limitation of all SIMT based hardware and is described in more detail by Porter et al.[18].

To reach maximal performance, the SIMT execution model requires code that does not diverge so that as many threads can work in parallel as possible.

2.4 GPGPU Compute Environments

To use the GPGPU capabilities of a GPU a parallel computing platform and Application Programming Interface (API) is needed to build the application and schedule the compute kernels on the GPU. The most commonly used tools that enable the use GPGPU functionality are the *CUDA Toolkit*[9] and the *Direct Compute* API[8].

2.4.1 CUDA

Cuda is a parallel computing platform and programming model that is programmed in a special version of C/C++ called *CUDA C/C++* that compiles all relevant code into *PTX code* which is then compiled by the driver into hardware-specific code at runtime. It also handles data copying to and from the GPU as well as kernel scheduling details automatically for the user. This approach provides an easy-to-use method of interfacing with the GPU side code, making data exchange, scheduling and setup as easy as possible. Since CUDA targets the compute capabilities of GPUs directly, no graphics pipeline is needed. Furthermore, the CUDA toolkit provides multiple different libraries providing parallelized functionality for, e.g., scientific research[9].

2.4.2 Fragment Shaders

Fragment shaders are a distinct shader stage of the rendering pipeline, also called *Pixel Shaders*, calculating the final pixel colors of their corresponding render targets. Fragment shaders offer another method for GPGPU computations and have been used for different applications like fluid simulations[11](chapter 38) before the introduction of GPGPU platforms and APIs such as CUDA or compute shaders. Instead of dispatching kernels to operate on data directly, fragment shaders are used to write the resulting calculations into a render target using the rendering pipeline.
This approach requires using the whole render pipeline since a mesh has to be drawn with the shader applied to it. It is, therefore, oftentimes more complicated to use than its compute shader counterpart, in addition to a potentially higher overhead caused by the rendering pipeline.

2.4.3 Compute Shaders

Compute shaders are a part of all modern graphics APIs and allow for the execution of specialized shaders that expand beyond graphics programming[8]. Compute Shaders are written in the APIs corresponding shader language. These languages commonly use C style syntax, supporting a comparable feature set, with varying limitations depending on the generation on the GPU. It features very similar capabilities to the CUDA-based kernels but requires a fully functioning graphics pipeline to schedule its kernels.
Since it does not rely on a specialized C/C++ compiler and is integrated into every graphics API, most graphics-oriented applications can integrate compute functionality more easily than its CUDA counterpart. Most game engines already have compute capability integrated into their rendering pipeline, enabling easy use of GPGPU capable hardware.
Due to their wide adoption in most existing 3D engines and their superior flexibility compared to fragment shaders, this work will only discuss the use of compute shaders as its method of parallelization on the GPU.

2.5 GPU WFC

While previous efforts to execute the WFC algorithm do exist in the form of a Python-OpenCl implementation called gpWFC[23], no details on the implementation or its performance are given. However, the documentation mentions a GPU only rendering approach, using which the wave buffer never has to be copied back to CPU for rendering during propagation.

OpenCl is used at two points throughout the solver, going off of the source code. Apart from being used for the propagation step, an OpenCl kernel is also used to find the lowest entropy node in the wave for the observation step.
While the exact methods used in the propagation kernels are unknown, some memory optimizations quickly become apparent in the form of memory packing. It is done wherever information can be packed into bit-wise flags, reducing memory access overhead and increasing the likelihood of cache hits since more information can be loaded into the caches than the flags full-size representations would allow for.

3 Approach

The goal of this work is to accelerate the WFC algorithm by implementing it on the GPU based on the aforementioned parallelization strategies, using compute shaders. The resulting implementation performance measurements should deliver detailed information on the viability of using highly parallelized hardware to run the WFC algorithm by analyzing the CPU and GPUs relative performance, scaling across multiple wave dimensions and tile-set configurations. Additionally, more detailed timing data of the solver's individual steps should yield information on what the best parallelization strategy is, as well as reveal potential areas of improvement for the different solvers.

Due to its simplicity, only the tiled model will be implemented, but since modern WFC approaches use the same solver for both models, expanding the implementation for the use of the overlapping model is possible as well.
Both parallel CPU and GPU solvers will be evaluated to gain as detailed information as possible, with the CPU implementations serving as a baseline for the GPU implementations, as well as a validation of the results by Orlowski and Lee[17].

3.1 Sequential WFC

The sequential version of the WFC algorithm will implement Gumin's current solver[13] implementation closely, with elements taken from the fast-wfc[19] implementation for improved readability. In order to stay as comparable as possible, no significant changes to the implementation will be made and a simple 2D tiled model with the same heuristic method as Gumin's implementation will be used. The only changes introduced to the solver serve debug-functionality and should come with little to no overhead. Therefore the outputs and performance of this implementation should be very similar to the one by Gumin[13].

3.2 Parallel WFC

3.2.1 CPU Parallel Job

Based on the sequential WFC profiling done by Orlowski and Lee(see fig. 5) and the resulting approach they took for multi-threading, the algorithm, only the propagation step of the WFC algorithm will be parallelized. Their report provides multiple approaches one could take to parallelize the CPU implementation providing different levels of performance increase. However, for the sake of simplicity and this research focusing on parallelization on the GPU, I will only provide implementations based on their fastest, work-queue-based approach. Due to its similarities to the GPU solvers, they serve as a point of reference for the GPU implementation.

3.2.1.1 Work-Queue

The approach using a lock-free work queue allows for computations to be focused on nodes that are actually affected by the collapse of their neighbors. With each iteration, the targeted node's state will collapse as far as needed based on the patterns possible in its surrounding nodes. Every time a node is collapsed, it gets added to the work-queue. We add the collapsed node instead of its neighbors to avoid potential race conditions that could result from enqueuing nodes from multiple threads simultaneously.

For each iteration, one node will be dequeued from the work queue and the four neighboring nodes calculated and assigned to the solver. In addition, the collapsed node will be inputted into the solver so that we only need to check the constraints against the actually collapsed node instead of all the neighboring nodes.

This process is repeated until no more nodes are inside the work-queue, signaling that the propagation has reached a stable state that will not collapse any further by itself.

To avoid race conditions resulting from reading and writing to the same data, all read-write buffers that get accessed from other nodes than the one they correspond to are double-buffered. This means that the state that is used as an input for the solver and is read from is separated from the data that gets written to. From now on, these buffers will be referred to as *in-* an *out-buffers* respectively, using the method *swap* to swap the references of the in and out buffer.

Due to its particular setup, this approach is thread-safe even without double buffering since the open nodes do not lie directly next to each other. Therefore cells that get written to are not read in the same iteration by default. To avoid a potential conflict, the wave buffer will be double buffered anyways, as future changes to the scheduling method could lead to race conditions and avoid potential contingency that could result from read and write operations on the same container blocking each other.

3.2.1.2 Work-Hash Set and Batched Propagation

The work-queue-based approach requires a very high amount of thread dispatches, with every collapsed node resulting in a new iteration, which could result in another four iterations. Since dispatching a new thread always introduces latency caused by thread creation overhead, low workloads on individual threads can lead to inefficiencies using multi-threading, to the point that scheduling new threads can be less effective than doing calculations sequentially on a single thread. For this reason, I believe the proposed work-queue method to be sub-optimal and introduce changes to the work tracking that should reduce the overhead by reducing the necessary thread dispatches.

The work tracking is done differently from Orlowski's and Lee's method of using a work queue in that a hash-set is used to keep track of all potentially changed nodes. Hash-sets are used for this approach since they can only hold a specific value once no matter how often it is "added" to the hash-set.

To achieve this functionality, hash-sets create a hash of the value added, serving as its index inside the

data structure, resulting in linear access times when searching for elements. Furthermore, does this method of indexing ensure that an element can only exist once inside a hash-set Hash-sets can also easily be converted to arrays for later processing.
The same double buffering method was used to ensure thread-safe behavior, as in the work-queue approach.

The proposed approach works roughly like this:

Algorithm 3 Work Hash-Set And Batched Propagation

```
1: while open nodes in wave do
2:     observe node and add its neighbouring nodes to the work hash-set, duplicates will be avoided implicitly
3:     while work hash-set is not empty do
4:         copy input wave to output wave
5:         convert hash-set to array (work-array) and pass it to propagation job
6:         start propagation job; use as many threads as there are open nodes in the hash-set; batch nodes on threads if required
7:         for each node in the work-array do                    ▷ This loop is multi-threaded
8:             for each neighbouring node do
9:                 if neighbour changed state then
10:                    for each pattern possible in node do
11:                        Check if pattern still compatible with any of the neighboring node's patterns
12:                        if pattern not compatible anymore then
13:                            if pattern not removed in output wave then
14:                                Remove pattern from node, and add its neighbouring nodes to the work hash-set
15:                            end if
16:                        end if
17:                    end for
18:                end if
19:            end for
20:        end for
21:    end while
22: end while
```

The buffer swapping is done in-between the propagation steps and results are always written to the out-buffers, while reading will only be done from the in-buffer if not mentioned otherwise.

Using this method, the propagation iteration can potentially cover hundreds of nodes in one go offering optimization headroom for thread scheduling. A potential optimization would be to assign multiple open nodes to one thread, eliminating the overhead that would result from individual thread dispatches on a per-node basis. The threshold of when to group nodes onto a single thread would then be controlled by the overhead of thread scheduling and is depends on the speed and number of threads available to the system.

3.2.2 GPU Compute Shader

Since modern GPUs support a large feature subset of CPUs such as varying loop sizes and floating-point operations, most of the strategies used for the parallel CPU implementation can be used for this implementation too. The propagation strategy is based on Orlowski's and Lee's[17] propagation approach again. It works by going over each node, checking if it potentially changed, and collapsing the node as far as possible based on the neighboring nodes. It uses the same double buffering approach of the CPU-Parallel implementations, using separated buffers for input and output data of each iteration to avoid potential race conditions that could arise from reading and writing to the same buffers simultaneously.

One of the critical differences of using the GPU for the propagation step is that we do not use work queues like the one in the parallel CPU implementation. This is due to the limitation of GPU's not supporting the allocation of arrays with changing sizes during runtime, making data structures like lists limited, only offering structures like appendable buffers which have to be allocated beforehand.

Therefore a compute buffer is passed to the compute shader with enough space to hold information on the state of each node and if it needs to be collapsed further. This buffer will be called *work-buffer* from now on. Compute buffers have the advantage that their size does not need to be known during the compilation of the compute shader, allowing for more flexible code without the need for recompilation for varying output dimensions. Only possibly changed nodes are checked using this strategy, reducing unnecessary calculations.

Since the work-buffer does not give any information about whether or not there are open nodes without going over each entry and checking until an open node is found, a flag is used that is updated every time a node is marked as open. This eliminates the need for a costly check of the whole work buffer after each iteration.

The resulting approach works roughly like this:

Algorithm 4 GPU Compute Shader

```
1: while open nodes in wave do
2:     observe node and mark neighbours as changed in the work buffer
3:     while open nodes do
4:         start propagation; dispatch shader using the dimensions as the waves as thread sizes
5:         for each node in the wave do                    ▷ This loop is multi-threaded
6:             copy current state of node to output wave
7:             for each neighbouring node do
8:                 if neighbour changed state then
9:                     for each pattern possible in node do
10:                        check if pattern still compatible with any of the neighboring node's patterns
11:                        if pattern not compatible anymore then
12:                            if ( thenpattern not removed in output wave)
13:                                remove pattern from node and mark neighbours as changed in the work
   buffer
14:                            end if
15:                        end if
16:                    end for
17:                end if
18:            end for
19:        end for
20:    end while
21: end while
```

The buffer swapping is done in between the propagation steps and writing is always done to the out-buffers, while reading will only be done from the in-buffer, if not mentioned otherwise. After the algorithm has finished, the results can then be copied back to the CPU for processing of the output.

3.2.2.1 Memory Sharing All data needed by the solver needs to be stored one the GPU's memory, as sharing CPU memory would result in a massive bottleneck due to the lower transfer speeds of the PCI-E connection compared to the on-board memory. Therefore all data shared between GPU and CPU needs to be copied back and forth whenever either one requires access. This introduces latency to the dispatch and execution of shaders every time memory needs to be copied, making minimal data transfers desirable.

Therefore the observation step will also be ported to the GPU although it is not parallelized.

Having the observation step execute on the GPU should reduce the overhead that comes with copying the compute shader data to the CPU memory and back allowing for quicker consecutive execution of the kernels.

3.2.2.2 Command Queue All GPU instructions are handled via so called *command queues*, storing all GPU commands in a sequential fashion. Graphics APIs allow to access these queues and add elements to them, which are then worked on after another, if needed and possible, multiple queues in parallel. If commands get added from a different thread, adding commands to the command queues does not guarantee that they get executed immediately after another since other threads could add other render commands between the solver-related ones. Therefore adding rendering commands once by one can lead to pauses during the execution of the WFC related commands, introducing time losses. To circumvent breaks, custom commands lists can be submitted, holding multiple consecutive commands, which get executed immediately after one another, eliminating waiting times introduced by other tasks.

3.3 Evaluation-Method

The different solvers will be evaluated based on timing benchmarks, using different tile-set configurations of the *Knots* tile-set provided by Gumin[13].
The hardware used to run the benchmarks is a Ryzen 7 5800X, 32GB of ram and an Nvidia Geforce GTX 1080.
For each tile-set configuration, 20 runs will be done peer solver for five different output dimensions, taking the average time of each solver as the final result. If a solver encounters a contradiction, a new run is started with a different seed and the total time of all runs will be measured, including all steps needed to prepare the solver for the next run.
In addition to the average execution time, multiple average measurements of the internal function calls, namely *Observe* and *Propagate* will be taken. This timing information should help analyze the total execution times more detailed and will be analyzed related to their respective compute environment.

The tested resolution will be 8x8, 16x16, 32x32, 64x64 and 128x128.

The different tile-set configurations will be used to represent different levels of restrictions, starting with a highly restricted tile-set (see fig. 9, 10, 11, 12, 13) that results in immediate collapse of the whole wave, and adds nodes until a highly flexible tile-set is reached, that reaches stable propagation states quickly without collapsing many nodes. The different configurations should reveal the strengths and weaknesses of the sequential and parallel implementations. The resulting timings of these configurations will further be analyzed based on the internal function timings.

Figure 9: Tile-Set 1

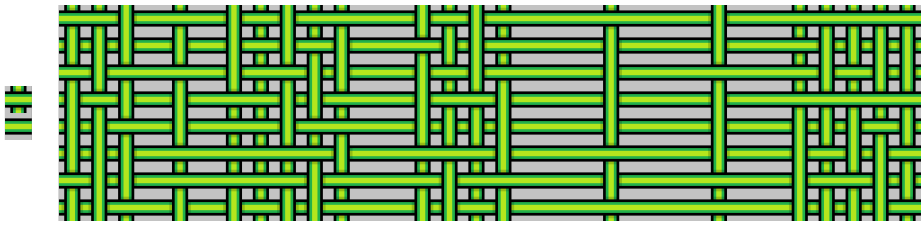


Figure 10: Tile-Set 2

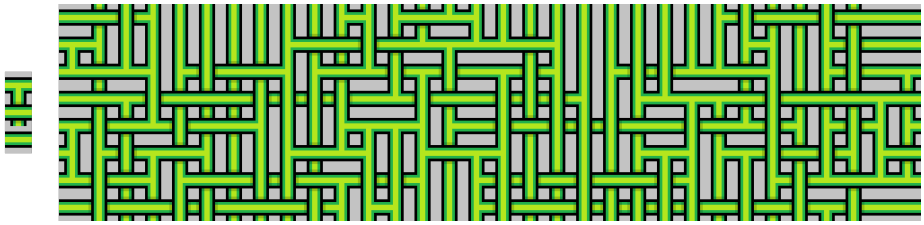


Figure 11: Tile-Set 3

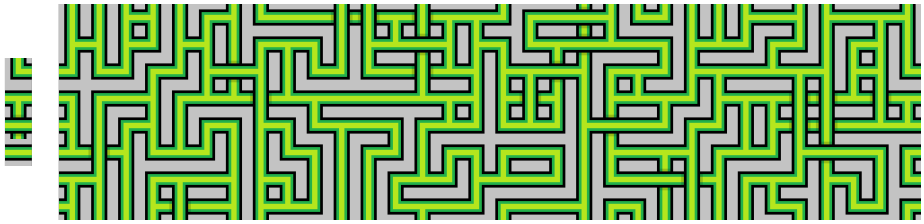


Figure 12: Tile-Set 4

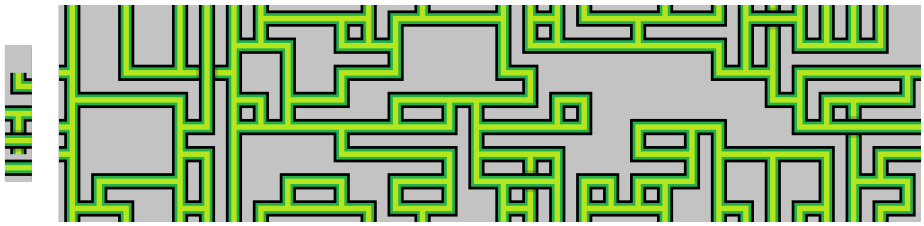


Figure 13: Tile-Set 5

4 Implementation

4.1 Platform and Environment

Most implementations of the WFC algorithm referenced throughout this work are standalone applications mostly lacking a User Interface (UI). They are also optimized to generate output files instead of displaying them directly and therefore do not come with a graphics back-end. While this approach reduces overhead and provides clean and concise code focusing on the WFC algorithm itself, it does not represent a representative experience one would get when using WFC inside a larger framework or game engine. Therefore, results collected in such an environment only deliver best-case scenario results that do not need to account for the inefficiencies introduced by such a software package.
To provide data that is more representative of a common use case, all implementations will be done using the *Unity Game Engine*[25].

Unity is a well-established game engine offering a diverse tool-set and allows for quick iterations thanks to its C# API. In addition to an easy-to-use, thread-safe multithreading system called *Jobs*[6], Unity provides Graphical User Interface (GUI) tools that allow for visual real-time debugging and inspection of the WFC generation progress. Furthermore, the job system supports compilation into highly-optimized native code using Unity's *Burst* compiler[4], offering significant performance speedups without much additional work.
The implementation of the different models will be done using regular C# for the sequential WFC model, *Burst Compiled Parallel Jobs* for the multi-threaded- and *Unity HLSL Compute Shaders* for the GPU implementations. Unity's job system is proprietary, but its functionality can easily be reproduced using regular C# and the *Parallel* class, providing similar tools to OpenMP[22]. It is chosen for this particular implementation because of its inherently thread-safe implementation and Burst compiler compatibility.
Burst is proprietary technology and produces significantly faster than regular C# could, through the use of Ahead-Of-Time (AOT) compiled native code. Therefore, all results collected represent the best-case scenario using C#, which is not directly representative of the performance expected from a regular parallel C# implementation. It only serves as a point of reference.

4.2 Base Framework

All implementations are based on a hybrid approach combining fast-wfc's[19] setup code and Gumin's WFC's solver[13]. Gumin's solver is used, since it, at the time of writing, contains the performance optimizations introduced in fast-wfc, while being optimized for use with C#, ensuring a clean implementation of the solver. Nevertheless, Gumin's implementation lacks documentation and is generally

more difficult to understand than fast-wfc. Furthermore, fast-wfc makes more robust use of object-oriented patterns, making it easier to follow in many cases. Therefore, most of the code surrounding the *Propagation* and *Observation* step, is ported from fast-wfc.

Gumin's implementation and fast-wfc are centered around the use of bitmaps, with most of the code using fixed types whenever dealing with supply data. Moreover, both implementations do not support dynamically changing the solver during runtime, which is required for the proper collection of timing data.

Therefore, changes were introduced to the algorithm, creating a generic implementation that can be expanded to use any input-data-type and support switching of the solver used at runtime. These changes allow for easier benchmarking and a standard interface for all solvers, providing a basic starting framework for all implementations.

Additionally, all implementations are based on the use of *Coroutines*, a feature of the Unity Game Engine that allows for stepped execution of functions using the *IEnumerator* interface provided by C#. It is used to support debug-functionality, allowing for GUI inspection of the current state of the WFC solver. Only minimal overhead is expected if debug functionality is disabled since all the debug-related code is separated from the actual solver, with little overhead potentially introduced by stepping through the enumerator manually and branching.

Data is supplied to the algorithm using Unity's *Texture2D* objects and a basic tile-matching system.

4.3 Sequential Implementation

The sequential implementation is a hybrid of Gumin's original WFC implementation[13] and fast-wfc[19] for optimal performance and readability, as explained above. It does not incorporate any changes to the solver itself other than the previously mentioned debug functionality, which does not change the functionality of the solver.

4.4 Parallelized Implementations

4.4.1 Burst Compiled Parallel Job Queue

The parallel job queue implementation makes use of the *IJobParallelFor* interface provided by the job system and a *NativeQueue* for work tracking. Since the job system provides a set of rules to make it thread-safe by default, no managed data types like C# lists or queues can be used. Instead, they are replaced by so-called *native collections* provided by the *Unity Collections* package.

As the name of the data structures suggests, this package provides wrapper structs for natively implemented data structures, such as arrays, lists, queues, hash-tables and hash-sets. Every data structure provides atomic access operations, which, together with the job system's safety systems, guarantee

data integrity and the avoidance of race conditions, as long as the restrictions are not explicitly deactivated.

Originally WFC uses multidimensional arrays, storing its data in an easy-to-access layout. *NativeArrays* do not support such functionality, requiring flattened data layouts using one-dimensional arrays and manual conversion of the multidimensional indices to one-dimensional ones. Therefore all data is flattened during the initialization phase of the algorithm and converted to multidimensional arrays when needed.

Both propagation and observation steps, including the *Next Unobserved Node* function, are written as Burst compiled jobs in hopes of achieving close to native code level performance.

4.4.2 Burst Compiled Parallel Job Hash-Set

The work-queue is replaced with a work-hash-set to reduce work and overhead introduced by repeated, unnecessary thread scheduling for nodes that were already collapsed during a previous iteration.

The implementation using a hash-set is almost identical to the queue-based one, but instead of scheduling four threads at a time, one for each neighbor of a collapsed cell, as many threads are scheduled as there are open nodes. The contents of the work hash-set are copied to an array before each iteration, serving as a lookup for each thread to determine the node it is supposed to work on.

Again both propagation and observation steps, including the *Next Unobserved Node* function, are written as Burst compiled jobs in hopes of achieving native code level performance.

4.4.3 Compute Shader Implementation

4.4.3.1 Naive Approach The naive compute shader implementation follows the approach presented by Orlowski and Lee[17] the closest, in that only the propagation step is run on the GPU.

Instead of performing all steps of the WFC solver on the GPU, all necessary data is copied back to the CPU after each iteration to check if the propagation has finished. The observation step is also performed on CPU using a very similar implementation as sequential and parallel CPU. However, instead of working on multidimensional arrays and double-precision entropy data, all arrays are flattened and doubles converted to floats. Floats are used instead of doubles as a consequence of using GPUs, with most not being capable of performing double-precision calculations in hardware. That being said, the use of floats should not cause any problems, as long as there are not multiple hundred different tiles in the input set.

This implementation is based on the assumption that the resulting overhead, introduced by copying the data back and forth, is smaller than the performance advantage that comes with using the CPU for single-threaded operations.

After the algorithm has finished, the resulting wave is then be converted back into a multidimensional array before executing the observation step using almost the same implementation as the sequential CPU solver, with the difference that open cells are marked as changed in the GPU buffer. Before execution of the next propagation step, all buffers are flattened again and copied to the GPU.

4.4.3.2 Full GPU solver with granular CPU checks For this implementation, all steps of the solver were ported to the GPU. It is based on the assumption that copying the solver data back and forth between the CPU and GPU introduces greater overhead than running the single-threaded operations on the GPU.
Instead of dispatching the kernel for the whole wave, like it is done for the propagation kernel, a single thread is scheduled to execute the observation kernel. Therefore, the observation kernel behaves the same as the naive implementation and can be ported almost one to one from C#, without introducing significant changes.
Since the GPU cannot be used to schedule kernels by itself, information on the current state of the solver gets copied back to the CPU after execution of each kernel to determine which kernel to schedule next. Furthermore, the CPU side will be used to swap the in- and out-buffers and to clear the work buffers for the next iteration by copying a cleared array to the GPU side buffer. The data required to determine the state of the solver is only a couple bytes large and therefore significantly smaller than what is required for the naive implementation.
This implementation is based on the assumption that copying smaller data sets than the naive approach reduces overhead introduced by the memory transfers, far enough to make it the faster approach in general.

4.4.3.3 Full GPU solver using compute buffers and coarse CPU checks While the fully GPU based solver does use less memory bandwidth than the naive solver, all kernel dispatches are still entirely dependent on the CPU-GPU data transfer speed. Therefore kernel execution is potentially bottlenecked by the CPU waiting for data from the GPU, causing unwanted overhead. Another weakness of the granular CPU-side checks is that other threads with access to the GPU command-queue can push commands in between WFC-kernel dispatch calls. This could cause periods of inactivity during which other kernels, than the WFC related ones, are executed on the GPU.
To circumvent this problem, a command-buffer will be used to schedule all relevant GPU command immediately after one another. Command-buffers have the advantage of pushing all commands to the command-queue consecutively, ensuring all commands in it will be executed as quickly as possible.
A downside of using command-buffers is that there is no way for the CPU to check the current state of the solver during the execution of the list. In the naive and granular implementation, clearing the work

buffer was done by copying the cleared data to the GPU-buffer when needed. However, without the ability to synchronize the GPU execution with the CPU this method becomes impossible. Therefore, in addition to the *Observe* and *Propagate* shaders, another compute shader is needed that clears the work output-buffer for the next iteration, which will be executed prior to each propagation iteration. Swapping the in- and out-buffers also becomes difficult since the bind-commands stored on command-buffer hold a fixed reference to one of the buffers on the GPU, that is not synchronized with the buffer references on the CPU side.

The ratio between observation and propagation steps required by the WFC algorithm varies drastically from tile-set to tile-set, with highly-restrictive tile-sets requiring far fewer observation steps than less restrictive ones. To execute the solver using a compute-buffer, multiple iterations of observations and propagations will be added to the buffer, instead of only one, with the goal of checking the output as little as possible, reducing downtime caused by unnecessary memory transfers.

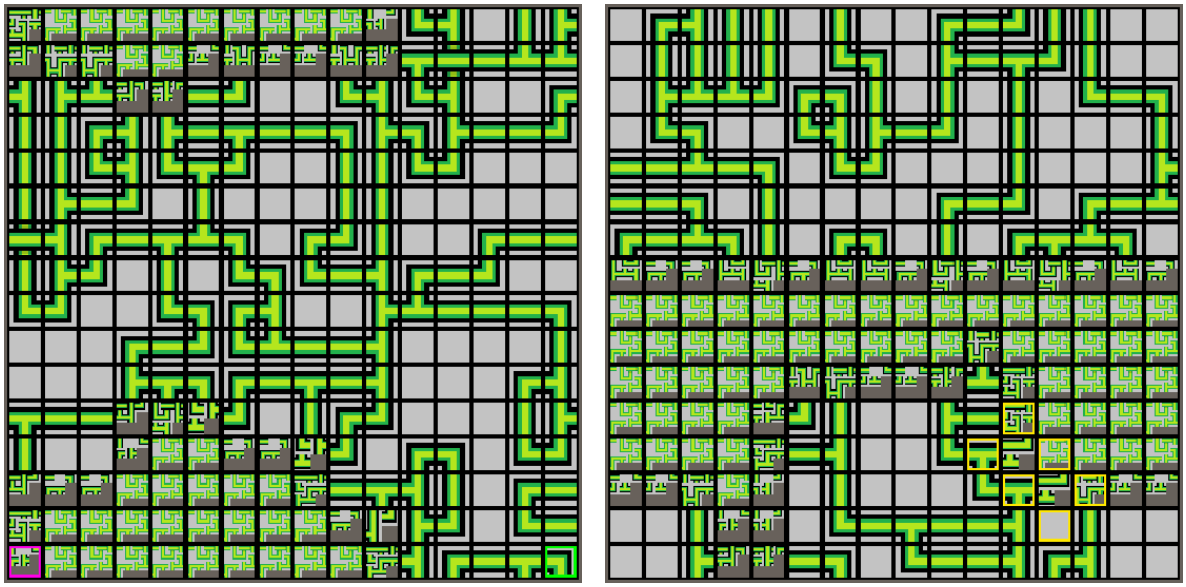
With each command-buffer dispatch, a fixed number of observations followed by a fixed number of observations are executed, with the clear-shader as part of the observation loop. Since the optimal number of propagation- to observation steps is defined by the restrictiveness of the tile-set and the wave's dimensions, fine-tuning has to be done for each solver configuration to achieve optimal performance and minimal unnecessary kernel executions. The dimensions of the wave affect this number since large waves require many more executions of the observation and observation kernel. An optimal configuration balances the ratio between observation and propagation steps, as well as the total amount of batched iterations of both before the results get copied back to the CPU for its checks. Misconfiguring the command-buffer will cause too much time spent waiting for data to be pushed around or computation cycles wasted on unnecessary kernel execution once stable states are reached.

4.5 Debugging

To better assess the functionality and correct execution of each WFC solver, a visual debugging feature was implemented as part of every solver, allowing for runtime inspection of the solver's state. This feature is essential for debugging of the GPU solver since it either does not return most of the data during execution, or does so in following the data layout required by the flattened multidimensional arrays, making a manual inspection of the raw data very time consuming and complicated.
For more detailed information, two debug modes were implemented for all solvers, allowing for fine or coarse inspection of the intermediate result of the propagation function, with the fine mode triggering the visual output from inside a more deeply nested loop than the coarse one allowing for inspection of more individual steps. Since the GPU compute-buffer solver does not execute its loops controlled by the CPU, the fine debugging option instead splits apart the individual computer buffers for observation

and propagation and triggers the visual feedback in between those.
The debugger also gives direct control over how long each solver snapshot gets displayed to allow for a more detailed inspection of each step when needed.

Each debugger snapshot displays the whole wave and its possible states for every node. If a node still has multiple possible patterns, it will display each pattern in a scaled-down grid fitted into the node's display area(see fig. 14).
Except for the CPU sequential solver, the debug view will also highlight the cells that are currently part of the work queue with a yellow frame, making inspection of the work queue possible(see fig. 14b). The sequential solver will instead highlight the currently worked on cell in violet, while the neighbor that's being checked will be highlighted in yellow, allowing for a step-by-step inspection of the whole sequential solver(see fig. 14a).



(a) CPU Sequential Solver Debug View (b) GPU Granular Solver Debug View

Figure 14: Debugger Visual Output

5 Result

All tile configurations show similar results, in which the GPU solvers perform the worst by a considerable margin, in some cases performing multiple hundreds of times worse than the CPU solvers, while the sequential solver performed the best for all cases. The single tile configuration is the exception to this rule, using which the CPU parallel queue solver performs the worst. For every configuration using more than one tile, more detailed measurements of the individual solver steps show that parallelization comes with a high cost for both CPU and GPU solvers. Every parallelized function call performed significantly worse than the sequential implementations and while the Burst-compiled single-threaded functions did not perform as bad, higher execution times were still measured across the board.

Since no parallel solver matches or surpasses the performance of the CPU-sequential one, the individual benchmarks and timings will be compared using it as the baseline measurement.

5.1 WFC Profiling

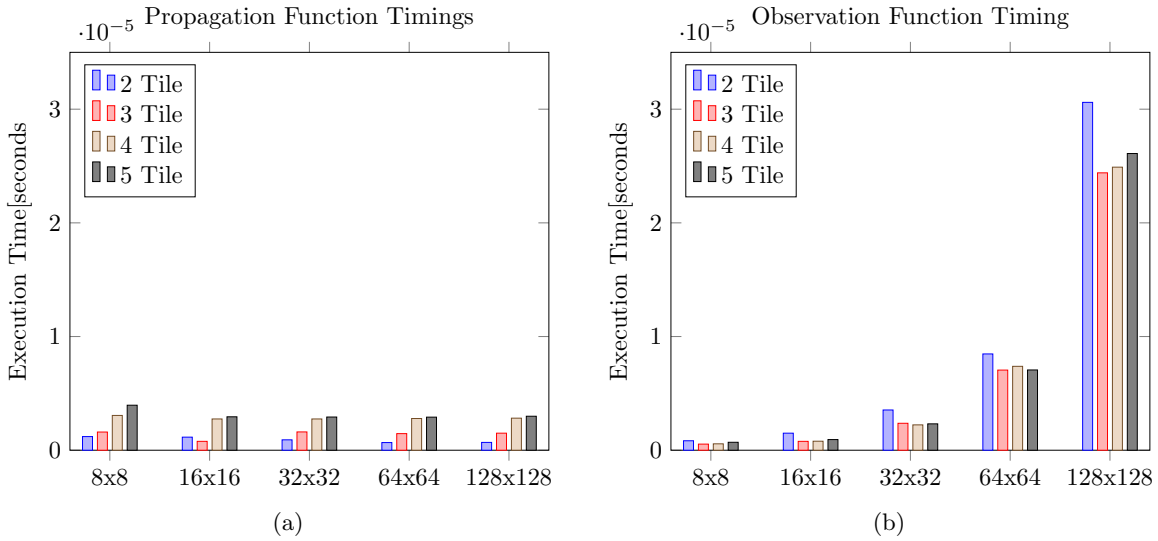


Figure 15: Sequential Solver Profiling

As part of the individual function timings, both the propagation and observation function were measured, with the results showing drastically different timings than those measured by Orlowski and Lee(see fig. 5), which measured significantly higher execution times for the propagation function at all sizes.

The measurements made using Gumins[13] propagation and observation code, in the context of the Unity implemented solvers, show completely different results, with the propagation function executing at almost the same speed across all wave dimensions and noticeably faster than the observation

function at sizes of 64x64 nodes and upwards.

5.2 Single Tile Performance

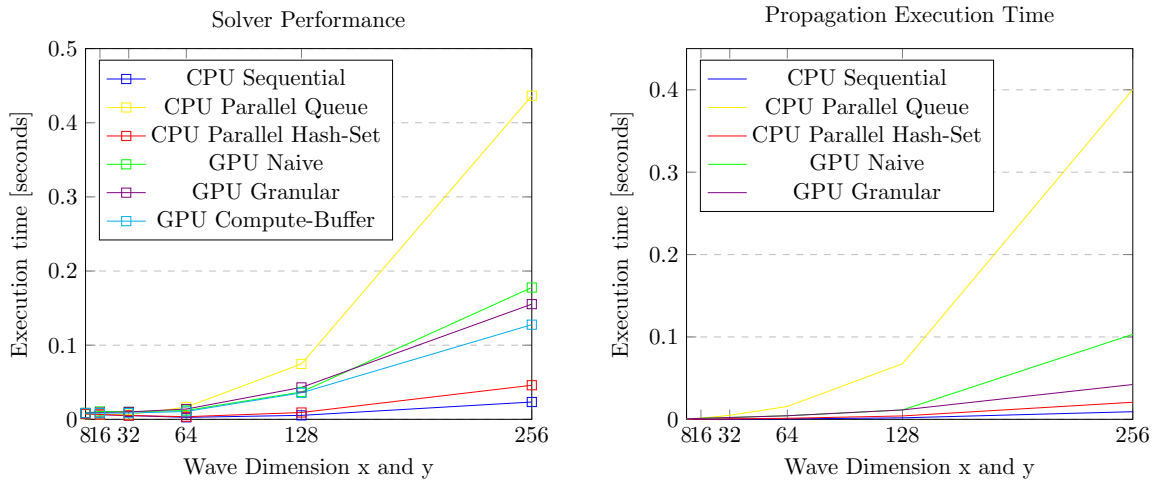


Figure 16: 1 Tile Solver Performance

5.2.1 CPU Parallel Solver

While the **CPU Parallel Queue** solver performs worse or equal to the hash-set solver at scales below 64x64 nodes, these differences are slight compared to the difference measured at large wave sizes. The queue solver performs the worst out of all solvers, with a significant margin, taking 510% - 1770
This trend is directly reflected in the propagation execution time, which executes 3400% slower on average, performing 4200% slower at 256x256 tiles.

The **CPU Parallel Hash-Set** solver is the second-fastest one performing its calculations with a time increase of 30% - 100% going from 64x64 to 256x256 nodes.
The cause of this performance advantage over the queue solver can be found in the propagation function execution timings (see fig 16b) as it only takes 120% - 140% longer to execute at this scale. At lower scales, this margin increases drastically to up to 1420% at 8x8 nodes.

5.2.2 GPU Solver

The GPU-solvers all perform very similarly to each other, with only small margins in between them. While initially, both the **GPU Naive** and **GPU Granular** solver perform almost equally fast in the range of 8x8 - 32x32, the naive solver performs faster at sizes 64x64 and 128x128, finally getting overtaken by the granular solver at 256x256 nodes.

Both solvers start at 20% - 100% slower at scales from 8x8 to 32x3. The naive solver then goes from 340% to 660% in the range from 64x64 to 256x256, with the granular solver initially performing worse at scales up to 128x128 tiles, going from 410% to 700% slower, only to drop below the naive solver with a scaling of 665% reaching 256x256 nodes.

The **GPU Compute-Buffer** solver performs the best out of all the GPU solvers showing relative slowdowns of 300% - 570%, in the range of 64x64 - 256x256, performing better than all other GPU solvers at every scale.

5.3 2 Tile Performance

Using two tiles and relaxing the restrictions of the tile-set increases the average generation time by 1007%, with the sequential solver needing 6736% more time to execute. These increases are especially noticeable at scales 32x32 and upwards and drastically increase previously measured differences between the GPU and CPU solvers.

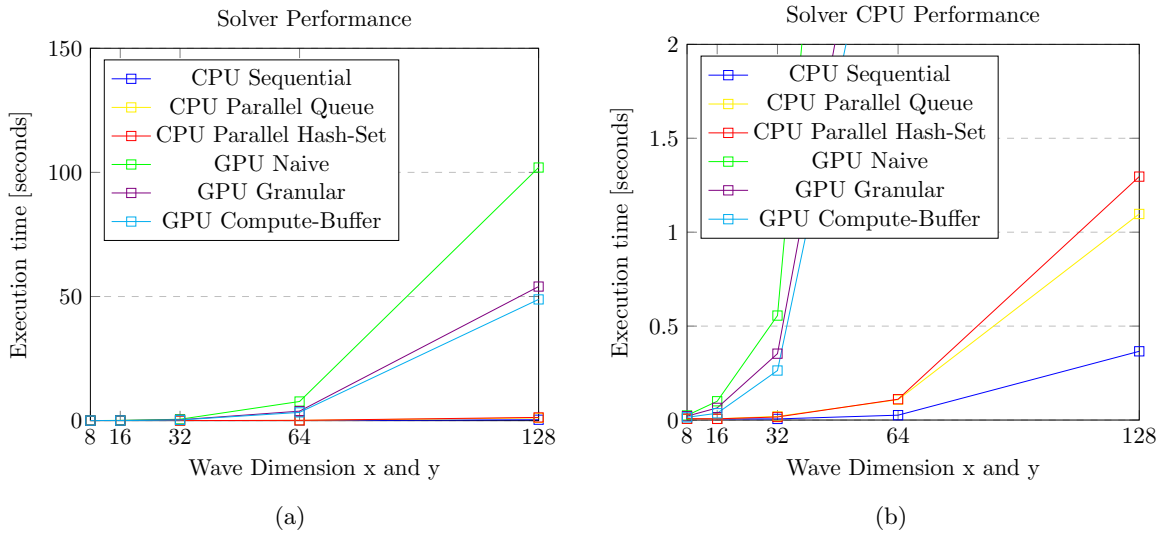
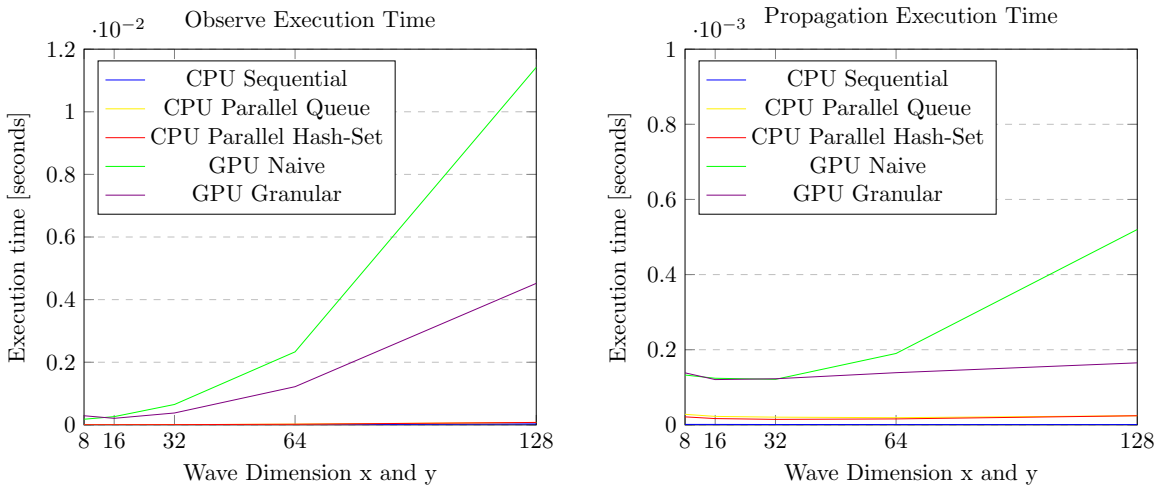


Figure 17: 2 Tile Solver Performance



(a) (b)

Figure 18: 2 Tile Function Timings

5.3.1 CPU Parallel Solver

While the **CPU Parallel Queue** solver performed significantly worse than all other solvers using the one tile configuration, this configuration puts it in an advantageous context compared to all other parallel solvers.

At scales up to 16x16 nodes, it performs its calculations within the margin of error to the sequential and hash-set solver, with a significant slowdown measured at 32x32, increasing the execution time to 320% and then 420% of that of the sequential solver. At 128x128 node, its ratio decreases to roughly 300%.

At 32x32 tiles, the **CPU Parallel Hash-Set** solver pulls ahead of the queue solver with a slowdown of 170% compared to the sequential solver but falls behind from 64x64 nodes upwards, going from a 330% - 250% slowdown.

The **individual function timings** reveal that the Burst compiled, single-threaded jobs used for the observation step perform significantly worse than the standard C# implementations. While it is expected that the parallel functions will take a little more time to execute, because of work-tracking related operations, the observed 160% increase in execution times over the sequential version are unexpectedly high.

The relative performance of the queue and hash-set solvers contradicts the individual function measurements, which show significantly higher execution times for the propagation steps of the queue solver. On average, it executes each function call 2480% slower than the sequential version, while the hash-set solver is only 2040% slower.

The performance difference of the two solvers also cannot be explained by the execution difference of the observation step since both solvers execute it in roughly the same amount of time.

Looking at the individual function timings(see fig. 18), reveals that the GPU solvers perform significantly worse than the sequential solver. Each propagation step takes roughly 23900% longer for the granular solver and 75400% longer for the naive implementation.
Compared to the sequential solver, both Burst compiled solvers take 150% more time to execute the observation step and 3400% longer to perform the propagation step, at a resolution of 128x128 nodes.

5.3.2 GPU Solver

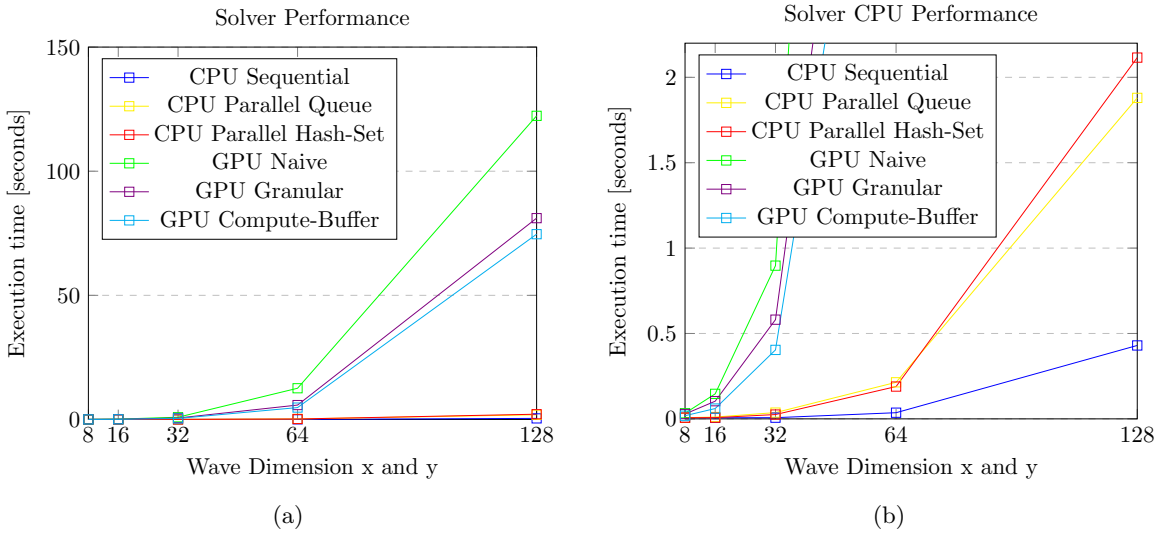
Reducing the level of restriction has a massive impact on the overall performance of all the GPU solvers, with the most extreme difference in execution time increasing from 0.037 seconds to almost 102 seconds, an increase of roughly 275600%.

The **GPU Naive** solver shows the worst performance of all, with an average increase in slowdowns of 3300%. It performs 270% - 29600% worse than the sequential solver, making it substantially slower than both of the other GPU solvers.
The individual function timings see an even worse slowdown, with the naive solver executing the propagation step anywhere from 10900% - 75500% slower than CPU sequential solver and the observation step takes 17400% - 37200% more time.

The **GPU Granular** solver already performs almost twice as fast as the naive solver and almost reaches the performance level of the compute-buffer solver, with execution times being 300% - 14700% slower than the sequential solver.
The relative speedup over the naive solver can also be seen in the individual function timings, where the granular solver shows slowdowns of roughly 10300% - 23900% over the sequential solver. The executing time of the observation method does increase with larger dimensions. However, it shows the biggest slowdowns at the smallest sizes, going from 23000% relative execution time at 8x8, to 16100% at 32x32 and back up to 19400% at 128x128.

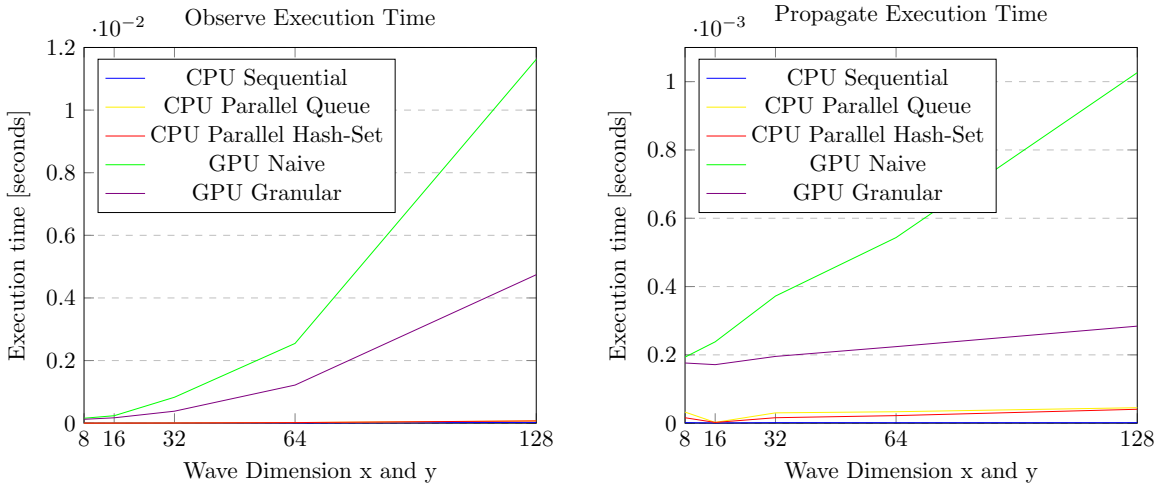
Again, the **GPU Compute-Buffer** solver performs the best out of all GPU solvers, performing roughly 17% faster on average than the granular solver. The measured timings show the execution taking 120% - 13200% longer than the sequential solver.

5.4 3 Tile Performance



(a) (b)

Figure 19: 3 Tile Solver Performance



(a) (b)

Figure 20: 3 Tile Function Timings

Using the 3 tile configuration, an average increase in generation time of about 52% can be observed, with the sequential solver needing 17% longer to execute, compared to the 2 tile configuration.
The average slowdown of all solvers increased by 25% compared to the previous generation, a considerably smaller increase than going from 1 to 2 tiles, which came with an average increase of 1370%.
The ranking of the solvers at 128x128 nodes stays the same with the sequential solver staying ahead, followed by the queue solver, the hash-set solver, and lastly, the GPU solvers, starting with the

compute-buffer, then the granular and at last the naive solver.

5.4.1 CPU Parallel Solver

The **CPU Parallel Queue** solver sees the most significant increase in computation time difference compared to the 2 tile configuration, increasing its average slowdown to 263%. Nevertheless, at 128x128 nodes, the queue solver stays ahead of all other solvers, showing a slowdown of roughly 340%.

Similar results can be seen for the **CPU Parallel Hash-Set** solver, which actually performs better on average than the queue solver, increasing its average slowdown to 316%. At 128x128 tiles, however, it still performs slower than the queue solver, with a slowdown of 390%.

These numbers stand in contrast to the propagation function timings, which show a significantly lower increase of slowdowns going from 2 to 3 tiles than previous generations. Both queue and hash-set solver increase their propagation time difference by roughly 26% and 39% respectively, with the queue solver needing 1800% and the hash-set solver 1200% longer to execute than the sequential version.
On the other hand, the observation function shows a decrease in performance, especially at larger scales, with the queue solver performing 190% and the hash-set solver 195% slower at 128x128 nodes.

5.4.2 GPU Solver

Relative to the 2 tile configuration, the GPU solvers show a less pronounced increase in overall slowdowns than the CPU solvers. However, due to their already higher execution time, the actual increase in execution time is a lot higher.

The **GPU Naive** solver shows the best scaling out of all the GPU solvers, increasing its average slowdowns by roughly 14%, compared to the 2 tile configuration. Using this configuration, it performs anywhere from 347% to 34759% worse than the sequential solver, with the slowest execution time measured at 64x64, dropping to 28454% relative execution time at 128x128.

The **GPU Granular** solver already performs substantially better than the naive solver but scales worse overall, with an increase in relative execution time of 23%, compared to the 2 tile configuration. It executes anywhere from 283% to 18764% worse than the sequential solver, scaling its worst at 128x128.

The **GPU Compute-Buffer** solver still shows the best performance numbers, with an average slow down of 7417%, roughly 20% slower compared to the 2 tile configuration. Execution time increases very similarly to the granular solver, starting with a slowdown of 151% at 8x8 nodes and increasing to 17261% at 128x128 nodes.

In addition to showing worse results than the previous configurations, the propagation execution times of all solvers switch from a more complex increase with size to a linear one, showing slight

fluctuations at sizes 8x8 and 16x16.

Contrary to the overall performance scaling of the naive and granular solver, the individual function timings scale more in favor of the granular solver.

The propagation function of the granular solver executes roughly equally as fast as it does using the 2 tile configuration, while the naive solver scales 23% worse than before.

Similar results were collected for the observation function, which scales roughly 46% worse for the naive solver, while the granular solver only increases its scaling by 10%, over the 2 tile configuration.

5.5 4 Tile Performance

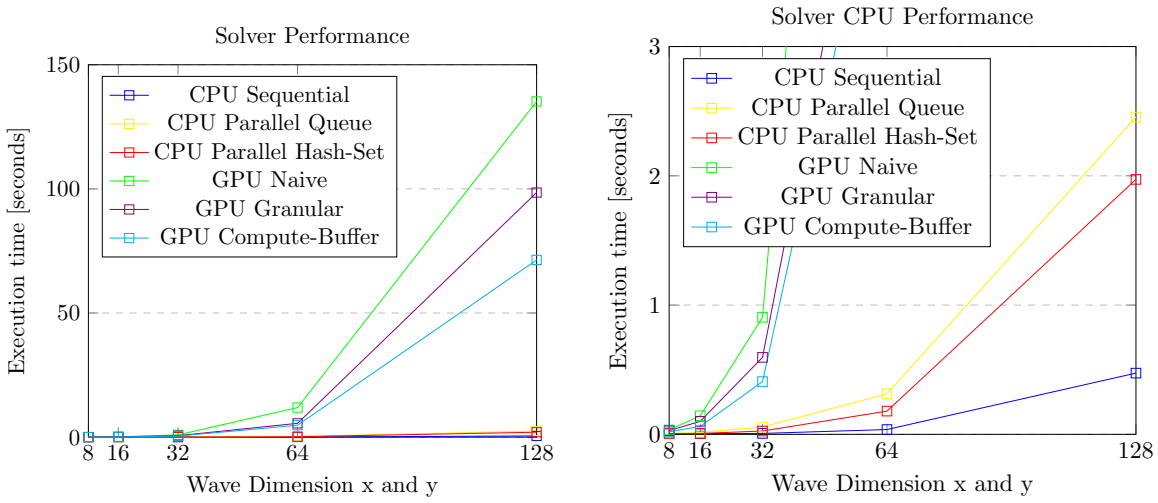


Figure 21: 4 Tile Solver Performance

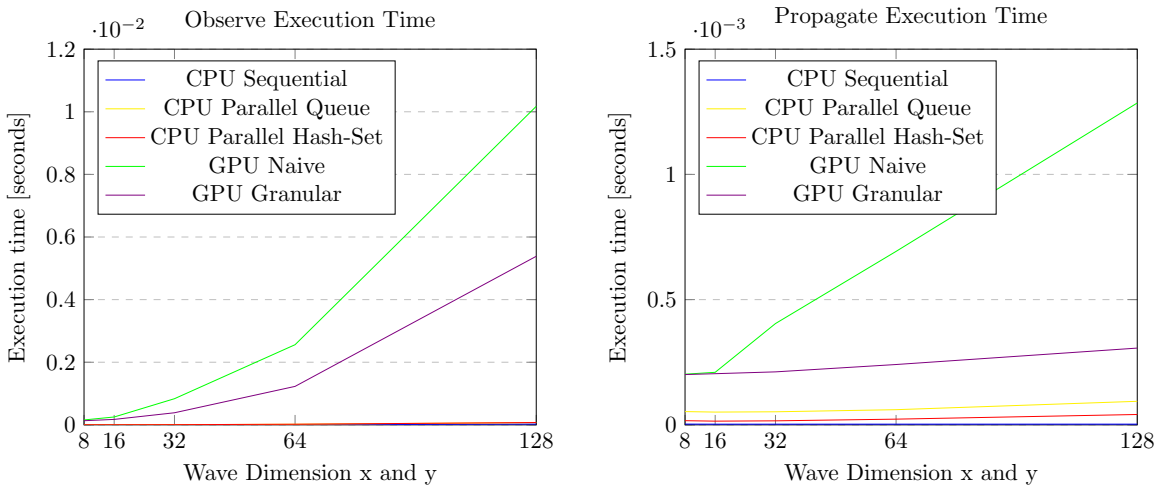


Figure 22: 4 Tile Function Timings

At four tiles, the point is reached at which almost all solvers improve their scaling relative to the sequential solver, with overall execution times increasing the least out of all configurations up to this point.

On average, all solvers increase their execution time by 10% over the 3 tile configuration and while not all solvers do, most improve their scaling noticeably.

5.5.1 CPU Parallel Solvers

The **CPU Parallel Queue** solver scales the worst out of all solvers compared to its previous generation, increasing its average slowdowns by 36% compared to the 3 tile configuration, reaching 395% on average. Its worst scaling remains at 64x64 tiles with a slowdown of 740%, which improves to 418% at 128x128.

These numbers are almost twice the once of the **CPU Parallel Hash-Set** solver, which sees an improvement in scaling of 7.5% over the 3 tile configuration. It also shows its worst scaling at 64x64 with a slowdown of 382% and improves to 310% at 128x128, making it the second-fastest solver after the sequential one.

The improvement of the hash-set solver can also be seen in the propagation function execution times, where it scales about 40% better than the 3 tile configuration, while the queue solver scales roughly 15% worse than before. Both solvers execute the observe method roughly equally as fast as they did for the 3 tile configuration.

5.5.2 GPU Solver

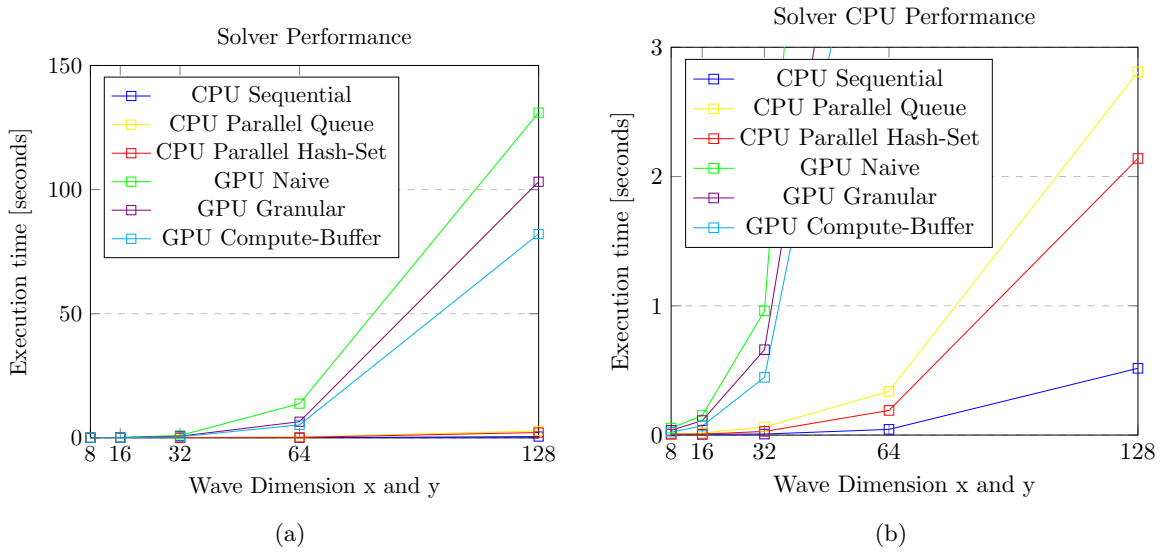
The **GPU Naive** solver shows an overall improvement in scaling of roughly 4%, lowering its average execution slowdowns for the 64x64 output from 34759% using 3 tiles, to 31788%. At all other sizes, the difference in scaling lies within the margin of error.

The **GPU Compute-Buffer** solver shows similar but overall more considerable improvements, scaling about 7.5% better than the 3 tile configuration. This improvement is especially noticeable at 128x128 nodes, where the relative execution times go from 17361% at 3 tiles to 15065%.

While the **GPU Granular** solver initially shows better scaling at sizes 8x8 - 66x64, 128x128 nodes show a significant slowdown, executing 20692% slower than the sequential solver.

Looking at the individual function timings, the naive and granular solver scale significantly better, improving their scaling by 42% and 48% on average. This leaves the naive propagation function at a slowdown of 6509% - 45439% and the granular solver at 6462% - 10461%.
The observation function sees the naive approach improving its average scaling by roughly 3.2% while the granular solver worsens by 3.3%. For the naive solver, the measurements show an overall scaling of 27900% to 40848%.

5.6 5 Tile Performance



(a) (b)

Figure 23: 5 Tile Solver Performance

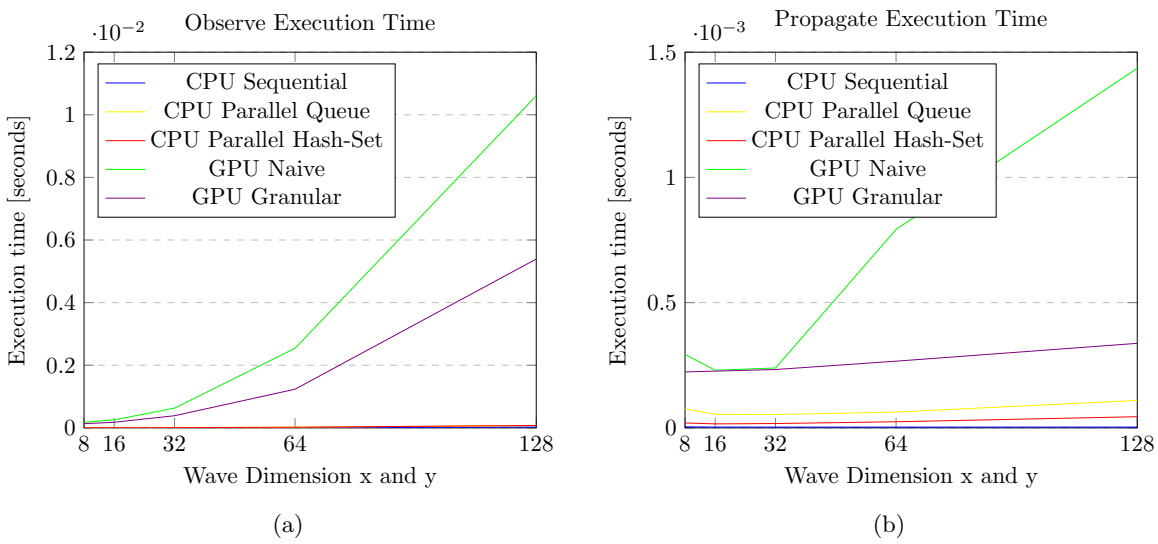


Figure 24: 5 Tile Function Timings

Increasing the number of tiles used to five increases the average execution time of all solvers by roughly 8% percent, that of the sequential solver by 9%.
With this lower overall increase in execution time, the scaling of all solvers, except the compute-buffer solver, improves by roughly 2% on average.

5.6.1 CPU Parallel Solver

At small scales, of 8x8 nodes, both the queue and hash-set solver scale roughly 10% better than the sequential solver.

While the **CPU Parallel Queue** solver shows almost identical average scaling results as the 4 tile configuration, deviating less than 1% from the previously measured scaling, slight differences at each scale can be seen. The solver scales about 8% better at 64x64 nodes and 10% worse at 128x128 than the 4 tile configuration, resulting in slowdowns of 668% at 64x64 and 444% at 128x128, compared to the sequential version.

The **CPU Parallel Hash-Set** solver improves its average scaling by roughly 4% compared to the 4 tile configuration, with the main improvement seen at 8x8 and 64x64 nodes. At 64x64 nodes, it improves from its slowdown from 381% at 4 tiles to 333% at 5 tiles.

For the hash-set solver, the propagation function scales almost equally as well as it did for the 4 tile configuration, while the queue solver experiences worse scaling by roughly 4%, with slightly worse scaling at 64x64 and 128x128 nodes.
Additionally, the queue solver experiences worse scaling when executing the observe function at larger

scales, worsening its average scaling by roughly 2%. On the other hand, the hash-set solver sees an overall speedup for the observe function, improving its average scaling by 4%, only degrading its scaling noticeably at 16x16 nodes.

5.6.2 GPU Solver

At scales below 32x32 nodes, all solvers scale slightly worse than before, with the granular and compute-buffer solvers additionally performing worse at 32x32.

The **GPU Naive** solver improves its scaling by roughly 5% on average. However, this improvement mainly occurs at scales above 32x32 nodes, improving the naive solver's slowdowns from 31788% using 4 tiles to 31183% at 64x64 nodes. A more significant improvement can be seen at 128x128 nodes where its relative execution time improves from 28539% to 25369%.

The **GPU Granular** solver manages to improve its scaling at 64x64 slightly while improving it at 128x128, going from 20792% down to 19978% relative execution time. On average, the granular solver scales only 1% better than it did with the 4 tile configuration.

The **GPU Compute-Buffer** solver is the only one that scales worse almost across the board, with the only improvement recorded at 64x64 nodes going from a slowdown of 12750% to 11758%, while at 128x128 relative execution times worsens from 15065% to 15895%.

The individual observation function calls see improved scaling by roughly 9% for the naive and granular solver compared to the 4 tile configuration. This improvement is less significant for larger sizes, with waves below the resolution of 64x64 nodes seeing the most significant improvement. Similar improvements can be seen at 64x64 and 128x128 nodes for the naive solver, while the granular solver only improves at 128x128, worsening its scaling by roughly 5% at 64x64.

While the propagation function, on average, does not show a significant change in scaling for both the naive and granular solver, in practice, both solvers scale slightly worse at almost any size.
The only improvement of the naive solver can be observed at 32x32 tiles, improving its slowdowns from 14583% to 8097%, but if this measurement is ignored, the average scaling worsens by roughly 40%.
While the granular solver shows similar scaling, it is not nearly as pronounced, with the average increase in scaling of 1.5% remaining representative for almost all the timings measured.

6 Analysis

In combination with the data collected from the individual solver steps, the runtime data of the different tile configurations gives a detailed insight, not only into the individual performance of each solver, but their relative scaling to each other, as well as allowing for an estimated analysis of potential areas of overhead.

While the sequential CPU solver is always the fastest one for each tile-set and size, a multitude of factors impacts these results, an analysis of which will help understand the individual limitations of the methods used. Furthermore, understanding the underlying factors that limit each method will help estimate their behavior in different untested environments and grant insights into the best general optimizations when dealing with each method's specific environment and platform.

6.1 Data Discrepancies

Based on the data collected by Orlowski and Lee[17], significantly improved results were expected for the parallel CPU- as well as the GPU solver. If we compare this works data to that of Orlowski and Lee(see fig. 6a), the performance achieved using the parallel CPU solvers also surpasses that of the sequential solver, reproducing the improved performance recorded previously.

However, discrepancies quickly become apparent when comparing Orlowski and Lee's timings of the sequential solver to the ones in this work. This works solver performs magnitudes faster, even without the additional performance gains of running it in a natively compiled language.
Looking at the sequential implementation used by Orlowski and Lee[17][10], the cause of this discrepancy is easily identifiable.

The propagation function uses a method of propagation that scales significantly worse than the implementation by Gumin[13], which was used for this work. While Gumin's implementation mainly grows in complexity based on the number of tiles used, the implementation used by Orlowski and Lee additionally increases directly proportional to the wave's dimensions, as it iterates over each node of the wave to check for changes, instead of targeting changed nodes directly.

Keeping this in mind the profiling data collected by Orlowski and Lee(see fig. 5) is not representative of current, state of the art, WFC implementations, such as *fast-wfc*[19] and therefore also Gumin's WFC implementation[13].
The optimized propagation function used for both does not see the same increase in computation time that Orlowski and Lee measured, but instead noticeably higher execution times of the *observe* step, at sizes above 32x32 nodes(see fig. 15). Since parallelization of the WFC algorithm currently only aims to reduce the execution time of the propagation function, no improvements to the overall algorithm can be observed, since it is not the more costly operation anymore and introduces more overhead than

it requires for the execution of the sequential version.

6.2 GPU Solver

As the results show, no matter which tile-set was used, the GPU solvers perform the worst out of all solvers by a significant margin, most of the time generating the output magnitudes slower than any of the CPU solvers, especially the sequential one. The difference between the individual GPU solvers themselves is also considerable, with the compute-buffer solver performing almost twice as fast as the naive implementation and significantly faster than the granular solver in most cases as well.

The main causes of the differences in performance between the CPU and GPU, as well the GPU's solvers themselves, that can reliably be identified from the collected data, are:

1. Slower single-core performance on the GPU
2. Low Parallelization
3. Memory bottlenecks
4. Unnecessary thread dispatches
5. Graphics-command execution latency

6.2.1 Slower single-core performance

Based on the information we have about the differences of CPUs and GPUs published by Nvidia[7] and rough data tests on the single-core performance of a single CPU- versus GPU-core[5], we can infer a significantly slower single-core performance on the GPU.

Since the WFC algorithm's observation step consists of multiple operations, none of them parallelized, the performance of this function is directly connected to the single-core performance of the CPU and GPU.

Looking at figure 22a a significant margin from the CPU sequential execution times to the GPU can be seen, resulting in a 21516% performance difference at 128x128. While there is not enough data to make guaranteed assumptions about the scaling behavior of this difference, it seems to remain roughly constant once the 3 tile configuration is reached and upwards.

This suggests that the performance difference of the CPU sequential and GPU granular solver is in large part influenced by the single-core performance of the GPU, with the GPU cores, on average, performing anywhere from 177 - 202 times slower. And while this performance differences, changes from tile-set to tile-set, no definite trend could be observed from the data-set at hand, implying that the relationship between CPU and GPU single-core performance could be and most likely is constant, with the measured fluctuations being caused by API and driver overhead.

6.2.2 Low Parallelization

The slower performance of the GPU solvers compared to the CPU solver can not be explained by the GPU's slower single-core performance directly, since the GPU has a major advantage when it comes to its number of cores and therefore cells that can be worked on in parallel. While the single-core performance of the GPU is always lower than that of the CPU, it still has the advantage for raw data throughput, as it is what it is designed for, while the CPU is designed for minimal latency[7].
However, this only holds true if the task performed is sufficiently and efficiently parallelizable, since the execution of the sequential code has to be slower than factors on the GPU such as API overhead, slow single-core performance and thread scheduling overhead. Therefore, insignificant parallelization results in unnecessary overhead, in addition to only yielding benefits if the slower single-core performance can be overcome by running a sufficient number of them in parallel.
In addition to parallelizing poorly, the algorithm that runs on each thread is more computationally intensive than individual loops of the sequential propagation and ban methods. This increase in computational complexity stemmed from the requirement of each node to check each of its neighbor's patterns and compare them to the possibility of the current node. At the same time, the sequential propagation method can target the specific patterns of each node much more efficiently due to the use of a compatibility lookup. Overcoming this increase in computational complexity requires even more extensive parallelization and, therefore, reduces the effectiveness of the parallel solver even further.

Based on the measurements we have of the single-core performance of the GPU from the observed step timings, we can assume that at least a couple dozen or even hundreds of threads would need to be run in parallel to overcome the aforementioned factors of overhead. The exact number can not be known since multiple unknown factors such as the driver, GPU architecture and memory affect the actual performance of measured for a single-threaded kernel dispatch.

Using the debugging feature and stepping through the algorithm slowly reveals that for a tile-set with little restrictions, such as tile-set 3(see 11), 4(see fig. 12) and 5(see fig. 13, the wave stabilizes very quickly, mostly resulting in very phew open nodes in the range of 4-8(see fig14b).
Looking at highly restrictive tile-sets such as tile-set 1 (see fig. 9), we can see that the GPU solvers scale drastically better, which is due to the simple reason that the collapse of one cell will lead to the collapse of all its surrounding cells, leading to hundreds of open cells on a large wave(see fig. 25).

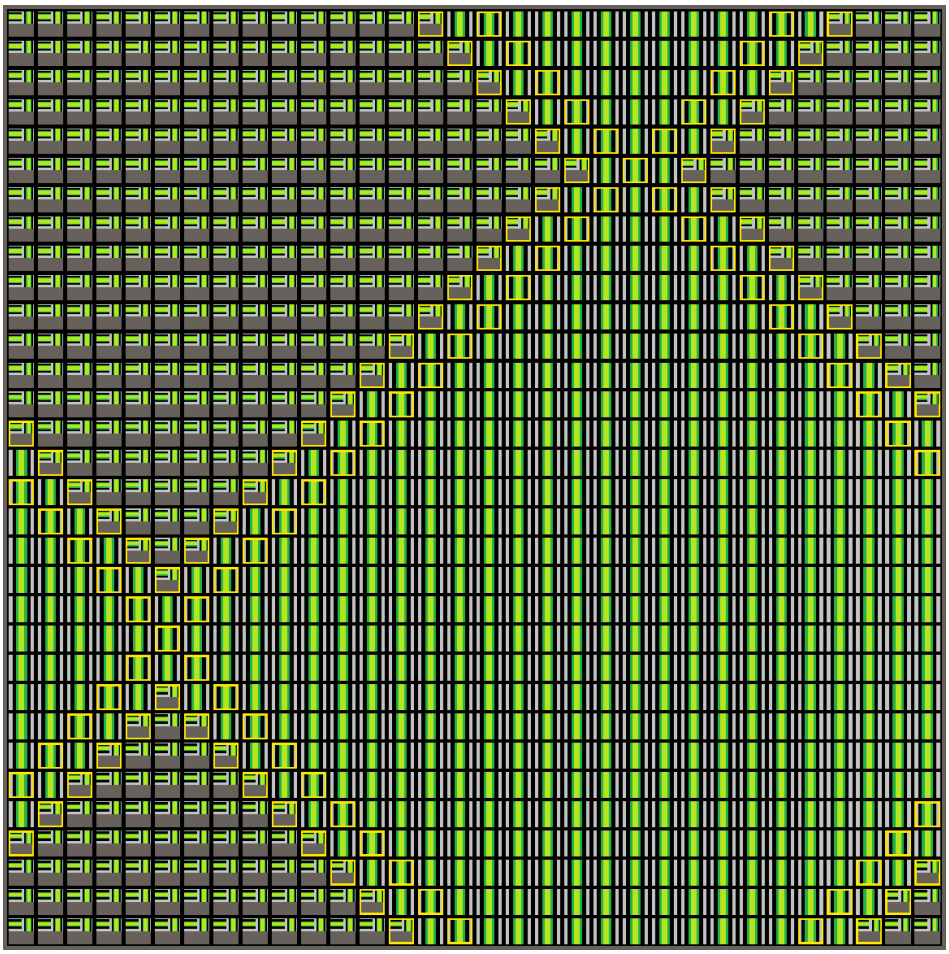


Figure 25: 1 tile propagation; open cells are marked yellow.

Based on these findings, the conclusion is made that the WFC propagation algorithm does not parallelize nearly well enough to make proper use of the GPU's highly parallel hardware.

6.2.3 Branching

The problem of slow single-core performance on the GPU is amplified further by the execution model used by GPUs called SIMT, which forces groups of threads, called a wave, to execute in lock-step(see section 2.3.1.1). The limiting factor of this model is that if a thread inside the same group diverges due to branching, all execution paths are executed sequentially until the control flows reconverges[14]. This is highly problematic since the propagation method used branches very often and is almost completely wrapped inside branching code, meaning that wherever there are a lot of open nodes inside a batch, all threads have to be executed sequentially, eliminating the advantage of parallelization.

Therefore, using the GPU to execute such a highly divergent algorithm is most likely highly ineffi-

cient and results in sequential execution of many nodes that should be calculated in parallel.

6.2.4 Memory Bottlenecks

Even though executing the observe function on the CPU is significantly faster than on the GPU (see fig. 18a,20a, 22a, 24a), this does not hold true for the GPU naive solver, which, even though the observe logic is executed on the CPU, performs far worse than any other solver, especially at larger wave sizes.
Furthermore, both the observe and propagate function scale is significantly worse for the naive solver than for the granular. Since the only additional operations executed for the naive observe function compared to the CPU sequential solver are the memory transfer and conversion operations required to set and get the data from and to the GPU, only these can be the cause of the relative slowdowns measured for the naive solver.

This assessment is further supported by the steeper increase in execution time of the individual function calls of the naive solver compared to the granular one. Figure 22 shows that the performance difference between the granular and naive shader grows proportionally to the wave's dimensions.
Since the observe function uses the same shader for every solver, the only differing factor is that the naive solver requires a transfer of the whole wave and the supplementary buffers before almost every propagation, since the wave stabilized very quickly using less restrictive tile-sets (see section 6.2.2). Therefore, the observe function is executed very frequently, requiring the wave, the entropy- and the collapse-data to be copied back to the GPU before the propagation shader can be executed.

The performance benefits of the compute-buffer- over the granular solver could also, in part, originate from its reliance on a separate compute-shader to clear its out buffers instead of the memory transfer operation required to clear the work buffers of the granular solver. In case the execution of the compute-shader is faster at clearing the memory of the out-buffers than a memory transfer, this would then lead to another speedup, resulting in the improved performance of the compute-buffer solver compared to the granular one.

Generally speaking, it seems beneficial to use the memory bus as little as possible, as it introduces a lot of latency, caused by factors such as API overhead and the data-transfer speed over the PCI-E bus.

6.2.5 Graphics-command Execution Latency

The compute-buffer solver was implemented to be run as independent of the CPU side process as possible. While it also replaces one of the memory transfers of the granular solver with the execution of a compute shader, its main difference is its scheduling of individual commands using a compute

buffer.
And while the compute-buffer solver requires the execution of another shader to clear its out-buffers, the overall performance of this approach surpasses that of the granular solver using any tile-set. In some instances, the compute-buffer solver even doubles the granular solver's performance advantage over the naive solver.

Since the compute-buffers solver's main discerning factor is its kernel scheduling method, a performance benefit of using compute-buffers over the individual, loose graphics commands can be inferred. However, the precise performance advantage of using compute-buffers can not be told from the data at hand, since the observed gains are partially caused by removing another memory transfer operation. Looking at the single tile performance (see fig. 16a) of the different GPU solvers, the compute-buffer solver sees more significant speedups over the granular solver than the granular over the naive solver, even though the amount of data that gets transferred is reduced drastically.

Therefore, one can assume that at least a portion of the measured improvements is due to using a compute-buffer, suggesting that submitting graphics commands in a manner that allows for execution immediately after each other is more efficient when dealing with a large number of commands.

6.3 CPU Solver

While the parallel CPU solver in Orlowski and Lee's work[17] did see substantial performance improvements compared to the sequential propagation method they used, no such improvements could be measured using a re-implementation of their work queue solver and the improved hash-set solver. The reason for this discrepancy was explained in section 6.1 and boils down to differences in complexity scaling between the sequential propagation methods used, making the sequential method used in this work significantly faster to execute at large wave sizes.

Nevertheless, the reason for the slowdown measured for the parallel propagation function and the Burst-compiled observe function requires further analysis.

6.3.1 Low Parallelization

As explained in section 6.2.2, low parallelization of the WFC algorithm is the main limiting factor for any multi-threaded implementation, since it does not create enough open nodes to justify the overhead of thread creation and syncing. Nevertheless, we do not see performance as bad as that of the GPU solvers, which most likely is due to the CPU being optimized for latency-critical applications[7], therefore reducing the overhead of parallel code execution drastically.
Still, the CPU parallel implementation uses the same strategies for propagation as the GPU solvers, which does seem to increase computation time enough that parallelization cannot make up for the lost

time anymore; even in cases where there are potentially dozens of open nodes(see fig. 16).

This problem is further amplified by the parallel queue solver's multi-threading strategy. It creates many unnecessary threads to check cells that have already been checked during the current propagation step, as described in section 4.4.2.
Using highly restrictive tile-sets (see fig. 16), this problem becomes especially apparent since nodes that get added from multiple neighbors will also get processed multiple times, even though they will only collapse once. This leads to the creation of so many threads that the collective overhead makes the performance of the queue solver even worse than the already much slower GPU solvers.

The hash-set solver successfully solves the majority of the overhead introduced by the queue approach by reducing the number of threads created, improving the parallel solver's performance slightly in cases of relaxed tile-sets (see fig. 19, 21, 23) and drastically when using highly restrictive tile-set(16). The reason for the hash-set's worse performance using the 2 and 3 tile configuration(see fig. 17, 19) is not known. However, it could be a random occurrence caused by the unpredictable generation results of the random seeds used for benchmarking.

6.3.2 Higher Complexity And Burst Overhead

While the worse performance of the parallel propagation function can be traced back to the overhead of the thread creation process, performance differences of the sequential and Burst compiled observe steps suggest that there are other factors harming performance, caused by the use of the Burst compiler and the operations needed for work tracking.

Measurements of the CPU parallel solver function timings showed that the observation step takes noticeably longer to execute when using the Burst compiled methods than when using the sequential ones.
This is most likely the cause of

1. Additional computations needed to add the open nodes for work tracking
2. Overhead caused by switching from managed to native code
3. Overhead caused by the job system's thread-safety system

While the sequential and parallel observation methods execute the same basic code that determines the next open node, pattern to collapse the node to and removes all other patterns from that node, the parallel solver requires extra work to determine the neighboring nodes and add them to the work tracking. Therefore, the execution of the removal step is slightly more complex for the parallel solvers' observation step.

Nevertheless, execution of the Burst AOT compiled code is expected to be faster than Just-In-Time (JIT) compiled code used for the sequential solver, therefore suggesting that more factors might play into the final measurements. From the documentation of the *Unity Job System* and its safety system, one prominent factor stands out that will most likely create much overhead for jobs that have a lot of data to work with.
Since the job system cannot check whether or not other threads of the application access or change the data of the containers passed to the jobs, a hard copy of all the data will be made for each job, ensuring thread-safety with other software components. Memory copy, just like memory transfers and allocations, introduces overhead, proportional to the amount of data used by the job, negatively affecting the dispatch speed of the job and introducing latency that would usually be avoided when working with manually managed containers.
Moreover, while Unity native-collections support active runtime safety checks for data access operations, these checks are deactivated for application builds, therefore not causing further overhead.

Another possible overhead factor, even though it is most likely a tiny one, is the context switch that occurs when switching from managed C# code to native code, as the Burst compiler generates it. In regular C# applications, invoking native code from external libraries has an overhead of between 10 and 30 x86 instructions per call, with marshaling creating additional overhead[26]. In addition to the overhead of calling external native code, Unity's job system will most likely introduce additional overhead when scheduling jobs and creating threads leading to further overhead.
However, the actual amount of overhead caused by the execution of individual Burst compiled jobs is not known, but if existent, likely very small.

6.4 Contradictions

Regardless of which solver is used, the most significant limiting factor, slowing down the WFC algorithm the most, are contradictions, which, despite not being the focus of this work, should be addressed when talking about the WFC algorithm's performance. No matter how fast a WFC solver executes, there will always be the chance for contradictions, forcing the algorithm to start over with a new seed. As explained prior, complex tile-sets with restrictive constraints will lead to contradictions increasingly often, especially at large wave dimensions, which is the use case where increased WFC performance is especially desirable.

Therefore, in addition to increased WFC solver performance, strategies are required that can deal with contradictions more efficiently than starting anew is. Some strategies that aim to resolve this limitation are further explored in my work on *Terrain Feature Generation Using Wavefunction Collapse*[12].

6.5 Conclusion

Even though the previous work by Orlowski and Lee on parallel WFC[17] showed significant performance improvement for the WFC algorithm using a parallel CPU solver, modern sequential implementations of the WFC algorithm deliver far superior performance compared to parallel CPU as well as GPU implementations.

Massive improvements of the WFC's propagation method have created an algorithm, the complexity of which grows significantly less than the sequential solver, that Orlowski and Lee's work was based on[10], resulting in propagation execution times that grow much less aggressively (see fig. 15). On the other hand, the observation step did not see the same improvements in execution time that the propagation step did, resulting in it being much slower at larger wave sizes than the propagation step, scaling significantly worse overall. Therefore, trying to optimize the propagation function further with the observation method in its current state is of little importance, since most tile-sets and wave-sizes shift the work balance from the, at small wave-sizes and tile-sets slower, propagation step to the observation step, increasing its overall execution time well past that of the propagation step.

Furthermore, while all parallel implementations of the WFC algorithm aimed at reducing the execution time of the propagation step, none succeed at creating a faster approach, even after granting the CPU solvers a theoretical performance benefit by running through AOT compiled native-, instead of JIT compiled C# code. The main limiting factor of all implementations, especially the GPU ones, is the low parallelization of the propagation step, which, except for the use of highly restrictive tile-sets, never leads to the creation of more than a couple dozen threads at max. Since efficient parallelization of an algorithm requires heavy workloads and high data throughput to overcome the overhead associated with scheduling the parallel hardware, using a parallel solver for the WFC algorithm does not yield any benefit using the methods described in this work.

This problem is further amplified for the GPU solvers because the code running on each thread is more complex than the individual iterations of the sequential propagation step, which, in combination with the significantly slower GPU single-core performance and hardware limitations, leads to substantially slower execution times. The SIMT execution model used on GPU will lead to sequential execution of many of the nodes that should execute in parallel, degrading performance even further. Similarly, the observation step executes much slower on the GPU than the already slower parallel CPU solvers.

Another major source of overhead, especially for the GPU solver, is the high amounts of memory transfers and copies, each inflicting a noticeable slowdown to the overall algorithm. And while this overhead can be reduced to a minimum by using the *compute-buffer* approach, factors like API and driver overhead still impacts performance severely, as can be seen at small wave sizes 16x16 and bellow,

where memory unrelated overhead causes the GPU solvers to perform worse, than at sizes of 32x32 nodes and above.

Overall, parallelization of the WFC's propagation function using the methods described in this work does not present a viable solution to speed up the algorithm, as current versions of the WFC algorithm already optimized the propagation algorithm to a point where parallelizations do not yield any performance increase. This holds especially true for GPU accelerated WFC, as there are simply too many factors of overhead and too little chance for parallelization of the workload to overcome them. Currently, improvements to the WFC algorithms execution speed are mainly that of contradiction avoidance, as it is still one of the most limiting factors for large-scale use of the WFC algorithm[12].

Therefore, the use of CPUs or GPGPU capable GPUs, to execute parallel implementations of the WFC algorithm is not suggested.

7 Future Work

7.1 Data Packing

All the data needed by the algorithm is currently laid out in the simplest but most memory inefficient manner, requiring full-sized 32-bit data types even for simple objects like booleans. While this format is easier to read and work with, most data does not require the full 32 bits per object, drastically increasing the amount of data that needs to be transferred for each memory access. This does not only have an impact on memory latency due to memory-speed limitations, but also reduces the likelihood of cache hits, because the information required for each propagation step is laid out too far apart in memory, additionally filling up the cache much quicker.
Packing the data more efficiently could result in significant speedups whenever memory operations are required while reducing the memory footprint of the application and increasing the maximum dimensions of the wave and the number of tiles used due to a reduction in buffer lengths.

7.2 Compatibility Lookup

While this work explored multiple approaches to the GPU based WFC solver, all of them use the same principle of checking all the still possible patterns against the neighboring ones. As mentioned before, this approach results in more computationally complex code, resulting in one of the most significant slowdown factors.
On the other hand, the sequential algorithm uses an approach that focuses the computations more efficiently on relevant patterns, using a compatibility data structure that keeps track of the number of compatible patterns of each node in all directions, allowing for more efficient elimination of patterns.
Initial tests showed that such a data structure could be used for the parallel solvers as well, with the tradeoff of requiring more memory to store the specific patterns removed from each node. Using a compatibility data structure could, however, result in faster execution times and is especially interesting for CPU solvers, as these already execute close to the sequential performance.

7.3 Adaptive Compute Shaders

In its current form, the propagation compute shader has to run for each node in the wave regardless of the nodes that are actually marked for change. A solver like the GPU granular could be adapted to dispatch a modified version of the propagation shader that targets the open nodes specifically, similar to the CPU hash-set solver.
Such a compute shader would require either a threadsafe stack to return an array with open cells or a step to retrieve the open cells from the work buffer in its current form.

The likelihood of improving the GPU solver past the sequential solver's performance is, however, quite slim since it would not address most of the main performance bottlenecks of the GPU solvers, like memory bottlenecks.

7.4 SIMD And Vectorization

Since the profiling of the WFC sequential solver in its current state showed that the observe step is the most costly in many situations (see fig. 15), further optimizations of that step could result in solid performance gains, especially at large wave sizes.

The observation step is highly sequential and would probably not allow for parallelization using traditional techniques. For similar reasons, the propagation function could not be parallelized successfully, requiring different methods to be used.

Another type of parallel processing is called Single Instruction, Multiple Data (SIMD), which, as the name suggests, can process the same computations on multiple data points in parallel, using single hardware-implemented instructions. Converting a sequential algorithm to make use of these instructions and process as many data points in parallel as possible is called *vectorization* and allows for significant improvements of numeric operations in many cases.
Since SIMD executes the same instructions for every supplied data point, branching algorithms are difficult to vectorize, making the propagation step a potentially bad candidate for such an implementation. This is also the reason why the SIMT execution model does not execute the propagation model efficiently, as can be seen with the GPU solvers.

SIMD vectorization could be used for multiple parts of the observe step where large data arrays have to be summed or averages calculated, without the additional cost of creating separate threads and syncing. Using SIMD could results in major improvements to the WFC algorithm and .Net allows for auto vectorization using the *vector* class, making implementation of multi-platform vectorization trivial.

References

[1] Damien Bastian. "Assassin's Creed Syndicate: London' Wasn't Built in a Day". In: GDC. 2016. URL: https://www.gdcvault.com/play/1023305/-Assassin-s-Creed-Syndicate (visited on 01/23/2022).

[2] Jonathan Beard. *A SHORT INTRO TO STREAM PROCESSING*. 2015. URL: https://www.jonathanbeard.io/blog/2015/09/19/streaming-and-dataflow.html (visited on 01/23/2022).

[3] *boost C++ Libraries*. URL: https://www.boost.org/ (visited on 02/10/2022).

[4] *Burst Compiler*. Unity Technologies. URL: https://unity.com/de/dots/packages#burst-compiler (visited on 01/26/2022).

[5] Huseyin Buyukisik. *How fast is a single GPU core compared to a CPU core?* 2021. URL: https://www.quora.com/How-fast-is-a-single-GPU-core-compared-to-a-CPU-core?share=1 (visited on 02/05/2022).

[6] *C#-Jobsystem*. Unity Technologies. URL: https://unity.com/de/dots/packages#c-job-system (visited on 01/26/2022).

[7] Brian Caulfield. *What's the Difference Between a CPU and a GPU?* 2009. URL: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ (visited on 02/05/2022).

[8] *Compute Shader Overview*. Microsoft. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader (visited on 01/24/2022).

[9] *CUDA Toolkit*. NVIDIA Corporation. URL: https://developer.nvidia.com/cuda-toolkit (visited on 01/24/2022).

[10] Emil Ernerfeldt. *wfc*. URL: https://github.com/emilk/wfc (visited on 01/31/2022).

[11] Randima Fernando et al. *GPU gems: programming techniques, tips, and tricks for real-time graphics*. Vol. 590. Addison-Wesley Reading, 2004.

[12] Lasse Foster. *Terrain Feature Generation Using Wavefunction Collapse*. Hochschule Darmstadt, 2022.

[13] Maxim Gumin. *WaveFunctionCollapse*. URL: https://github.com/mxgmn/WaveFunctionCollapse (visited on 01/23/2022).

[14] Axel Habermaier and Alexander Knapp. "On the Correctness of the SIMT Execution Model of GPUs". In: *Programming Languages and Systems*. Ed. by Helmut Seidl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 316–335. ISBN: 978-3-642-28869-2.

[15] Isaac Karth and Adam M. Smith. *WaveFunctionCollapse is Constraint Solving in the Wild*. Hyannis, Massachusetts, 2017.

[16] Marian Kleineberg. "Infinite procedurally generated city with the Wave Function Collapse algorithm". In: (2019). URL: https://marian42.de/article/wfc/ (visited on 01/23/2022).

[17] Amy Lee and Jan Orlowski. *Parallel Wave Function Collapse*. 2019. URL: https://amylh.github.io/WaveCollapseGen/ (visited on 01/23/2022).

[18] Adam Levinthal and Thomas Porter. "Chap - a SIMD Graphics Processor". In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 77–82. ISBN: 0897911385. DOI: 10.1145/800031.808581. URL: https://doi.org/10.1145/800031.808581.

[19] Fehr Mathieu. *fast-wfc*. 2018. URL: https://github.com/math-fehr/fast-wfc (visited on 01/23/2022).

[20] Paul Merrell. *Comparing Model Synthesis and Wave Function Collapse*. 2021. URL: https://paulmerrell.org/wp-content/uploads/2021/07/comparison.pdf (visited on 01/23/2022).

[21] Paul Merrell and Dinesh Manocha. *Continuous Model Synthesis*. Singapore, 2008. DOI: 10.1145/1457515.1409111.

[22] *OpenMP*. OpenMP ARB. URL: https://www.openmp.org/ (visited on 01/25/2022).

[23] sol. *gpWFC*. 2018. URL: https://github.com/s-ol/gpWFC (visited on 01/23/2022).

[24] Oskar Stålberg, Richard Meredith, and Martin Kvale. *Bad North*. 2018. URL: https://www.badnorth.com/ (visited on 01/23/2022).

[25] *Unity Real-Time Development Platform*. Unity Technologies. URL: https://unity.com/ (visited on 01/23/2022).

[26] Tyler Whitney et al. "Calling Native Functions from Managed Code". In: (2021). URL: https://docs.microsoft.com/en-us/cpp/dotnet/calling-native-functions-from-managed-code?view=msvc-170 (visited on 02/08/2022).