Towards a spatial model checker on GPU*

Laura Bussi[1], Vincenzo Ciancia[2], and Fabio Gadducci[1]

[1] Dipartimento di Informatica, Università di Pisa
[2] Istituto di Scienza e Tecnologie dell'Informazione, CNR

Abstract. The tool `VoxLogicA` merges the state-of-the-art library of computational imaging algorithms `ITK` with the combination of declarative specification and optimised execution provided by spatial logic model checking. The analysis of an existing benchmark for segmentation of brain tumours via a simple logical specification reached very high accuracy. We introduce a new, GPU-based version of `VoxLogicA` and present preliminary results on its implementation, scalability, and applications.

Keywords: Spatial logics · Model Checking · GPU computation

1 Introduction and background

Spatial and Spatio-temporal model checking have gained an increasing interest in recent years in various application domains, including collective adaptive [12, 11] and networked systems [5], runtime monitoring [17, 15, 4], modelling of cyber-physical systems [20] and medical imaging [13, 3]. Introduced in [7], `VoxLogicA` (*Voxel-based Logical Analyser*)[3] caters for a declarative approach to (medical) image segmentation, supported by spatial model checking. A spatial logic is defined, tailored to high-level imaging features, such as regions, contact, texture, proximity, distance. Spatial operators are mostly derived from the *Spatial Logic of Closure Spaces* (SLCS, see Figure 1). Models of the spatial logic are (pixels of) images, with atomic propositions given by imaging features (e.g. colour, intensity), and spatial structure obtained via adjacency of pixels. SLCS features a modal operator *near*, denoting adjacency of pixels, and a reachability operator $\rho\ \phi_1[\phi_2]$, holding at pixel x whenever there is a path from x to a pixel y satisfying ϕ_1, with all intermediate points, except the extremes, satisfying ϕ_2.

The main case study of [7] is brain tumour segmentation for radiotherapy, using the BraTS 2017 public dataset of medical images [2]. An high-level specification for glioblastoma segmentation was proposed and tested using `VoxLogicA`,

* Research partially supported by the MIUR Project PRIN 2017FTXR7S "IT- MaT-TerS" and by POR FESR Toscana 2014-2020 As. 1 - Az. 1.1.5 – S.A. A1 N. 7165 project STINGRAY. The authors are thankful to: Raffaele Perego, Franco Maria Nardini and the HPC-Lab at ISTI-CNR for a powerful GPU used in early development; Gina Belmonte, Diego Latella, and Mieke Massink, for fruitful discussions. The authors are listed in alphabetical order, having equally contributed to this work.

[3] `VoxLogicA`: see https://github.com/vincenzoml/VoxLogicA

$$\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathcal{N}\phi \mid \rho\,\phi_1[\phi_2]$$

Fig. 1. SLCS syntax. Atomic propositions p correspond to image properties (e.g. *intensity*, colour); boolean operators act pixel-wise; the *near* operator $\mathcal{N}$ denotes pixel *adjacency* (using *8-adjacency*: the pixels having a vertex in common with a given one).

resulting in a procedure that competes in accuracy with state-of-the-art techniques. In [6], also an accurate specification for nevus segmentation was presented. This paper introduces a novel development in the direction of taking advantage of Graphical Processing Units: high-performance, massively parallel computational devices. GPU computing differs from the *multi-core* paradigm of modern CPUs in many respects: the execution model is *Single Instruction Multiple Data*; the number of computation cores is high; the memory model is highly localised and synchronisation among parallel threads is very expensive. Each GPU core performs the same operation on different coordinates (a single pixel, in our case). The dimension of the problem (e.g. the size of an image) is provided to the GPU when the program (*kernel*) is launched, yet the number of threads does not scale with the problem size, being bounded by the number of computing units in the GPU. Currently, such a number is in the order of thousands, whereas the problem size may include millions of tasks. The problems that benefit the most of such architecture are the inherently massively parallel ones. In that case, the main issue is to minimise read/write operations from and to the GPU memory, and to turn a problem into a highly parallel implementation.

A substantial redesign is thus required to port existing algorithms to GPUs. `VoxLogicA-GPU` implements the core logical primitives of `VoxLogicA` on GPU, sharing motivation with a recent trend on implementing formal methods on GPU [8, 21, 22, 16, 18]. This paper aims to describe the tool architecture, including asynchronous execution of logical primitives on GPU and garbage collection, and to demonstrate a consistent efficiency improvement. In doing so, we had to overcome two major issues: implementing connected component labelling on GPUs and minimising the number of (computationally expensive) CPU $\leftrightarrow$ GPU memory transfers. Our current results are very encouraging, obtaining a (task-dependent) speed-up of one or two orders of magnitude.

2 Functional Description and Implementation

`VoxLogicA-GPU`[4] is a *global*, *explicit state* model checker, aiming at high efficiency and maximum portability. It is implemented in `FSharp`, using the `NET Core` infrastructure, and the *General-Purpose GPU computing* library `OpenCL`[5]. The choice of `OpenCL` is motivated by portability to different GPU brands.

[4] `VoxLogicA-GPU` is Free and Open Source software. Its source code is currently available at https://github.com/vincenzoml/VoxLogicA/tree/experimental-gpu.

[5] `FSharp`: see https://fsharp.org. `NET Core`: see https://dotnet.microsoft.com. `OpenCL`: see https://www.khronos.org/opencl. `ITK`: see https://itk.org.

`VoxLogicA-GPU` is a command line tool, accepting as input a text file describing the analysis, and a number of input images. The text file contains a set of logic formulas and parametrised, non-recursive macro abbreviations. As in [7], the tool expands macros, identifies the *ground* formulas (that is, without variables), and constructs a directed acyclic graph of *tasks* and *dependencies*. Such a graph is equivalent to the syntax tree, but it enjoys *maximal sharing*: no sub-formula is ever computed twice. In the CPU version, the tasks run in parallel on the available CPU cores, yielding a speed-up proportional to the degree of parallelism of the task graph and to the number of cores. In the GPU version, the tasks are currently executed asynchronously with respect to the main CPU execution thread, but sequentially: so-called *out-of-order execution* is left for future work.

The focus of this first release of `VoxLogicA-GPU` is on the *design* of a free and open source GPU-based infrastructure, with proven scalability. Thus, development has been narrowed to a core implementation that is powerful enough to reach the stated objectives, although not as feature-complete as `VoxLogicA`. In particular, the implemented primitives are those of SLCS plus basic arithmetics, and computation is restricted to 2D and integer-valued images. Implementation-wise, `VoxLogicA-GPU` is a command-line tool. It takes only one parameter, a text file containing the specification to be executed, i.e., a sequence of commands. Five commands are currently implemented: `let`, `load`, `save`, `print`, `import`. The model checking algorithm of `VoxLogicA-GPU` is shared with `VoxLogicA`. After parsing, parametric macros are expanded, while at the same time (to avoid explosion of the syntax tree) the aforementioned task graph is computed. A major issue is that each task allocates a memory area proportional to the size of the input image to store its results, thus *garbage collection* is required. The current strategy is a simple reference counting, as the number of reverse dependencies of each task (i.e. the tasks taking the given one as argument) is known before execution, and no task is created at run time. This problem is more relevant to the GPU implementation: as a GPU memory is usually smaller than a CPU one, and GPU buffers are explicitly allocated by the programmer, large formulas can easily lead to *Out of Memory* errors at run time. If a reference counter turns to 0, no more tasks take the given one as an input, and the pointer referencing the buffer can be disposed. As no pointer longer refers that GPU memory area, this can be reused. A task is an operator of the language or an output instruction. The semantics of the former is delegated to the GPU implementation of the VoxLogicA API, defining the core type `Value`, which is instantiated as a shorthand for a type called `GPUImage`. Such type represents a computation, asynchronously running on a GPU, whose purpose is to fill an image buffer. `GPUImage` contains a pointer to a buffer stored in the GPU, its imaging features, and an *OpenCL event* (an handle to the asynchronous computation). The latter is used to wait for termination before transferring the results to the CPU and to make task dependencies explicit to the GPU for proper sequencing. Since commands, parameters, and results must be transferred from the CPU to the GPU and back, keeping pointers to GPU buffers minimises this overhead, allowing for the reuse of partial results. Thus, data is transferred only at the beginning of the com-

putation and when retrieving results to be saved to disk. The model checker is responsible for decreasing reference counts after each task terminates, and for scheduling garbage collection when a reference counter reaches 0. Each operator is implemented in a small module running on CPU, whose only purpose is to prepare memory buffers and launch one or more *kernels* (i.e. functions running on GPU). As in `VoxLogicA`, the reachability operator $\rho\ \phi_1[\phi_2]$ is implemented using connected components labelling.

2.1 Connected components labelling in `VoxLogicA-GPU`

We designed a simple algorithm for connected component labelling, biased towards implementation simplicity, although efficient enough for our prototype. Similarly to the classic result in [19], the algorithm exploits the *pointer jumping* technique[6]: see Algorithm 1 for the pseudo-code of the kernels (termination checking is omitted) and Figure 2 for an example. After initialisation, *mainIteration* is iterated. By pointer jumping, it converges in logarithmic time with respect to the number of pixels N, but it may fail to correctly label connected components with corners in specific directions (see Figure 2, Iteration 13). Then *reconnect* is called, checking if there are two adjacent pixels with different labels, and changing one of them (deterministically chosen) so that the two labels now coincide The way *reconnect* changes the image ensures that *mainIteration* will restart and will be enabled to converge again. The termination condition is reached when *reconnect* does not change the image, which requires a global check on its input and output. For checking termination we adopted a *reduce*-type operation[7]: it takes $\log(N)$ iterations, since it divides the image size at each iteration until a single-pixel image containing a boolean flag is obtained. If the termination condition is false, the algorithm restarts from *mainIteration*[8]. In most cases, *reconnect* is called a very small number of times before convergence, and the total number of iterations is in the order of $log(N)$ (see [10] for details).

3 Preliminary evaluation

This section illustrates the scalability results obtained in our preliminary tests[9]. Experiments have been executed on a machine equipped with an `Intel Core i9-9900K` and a `NVIDIA RTX 3080` GPU. This is indicative of the attainable speed-up as both CPU and GPU are current, high-end (workstation-oriented) devices. It is important to remark that CPU and GPU execution times are subject to high variability. Indeed, a highly parallel test may run about 8 times

[6] *Pointer jumping* or *path doubling* is a design technique for parallel algorithms that operate on pointer structures, such as linked lists and directed graphs. It allows an algorithm to follow paths with a time complexity that is logarithmic with respect to the length of the longest path. It does this by "jumping" to the end of the path computed by neighbors. See https://en.wikipedia.org/wiki/Pointer_jumping.

[7] See e.g. https://en.wikipedia.org/wiki/MapReduce.

[8] Since checking termination takes $log(N)$ iterations, instead of waiting for *mainIteration* to converge, *reconnect* is called each k iterations ($k = 8$ in the current implementation, which experimentally proved to be a reasonable compromise).

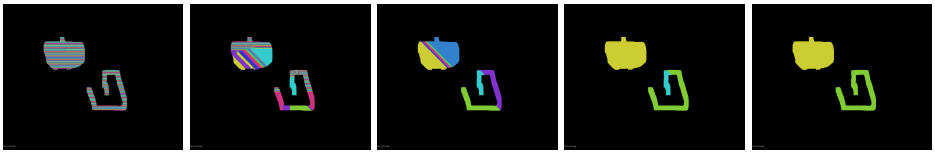
[9] All the tests we present, and the script to run them, are available in the source code repository https://github.com/vincenzoml/VoxLogicA/tree/experimental-gpu.

Algorithm 1: Pseudocode for connected components labelling

```
1  initialization(start: image of bool, output: image of int × int)
2      // parallel for on GPU
3      for (i, j) ∈ Coords do
4          if start(i,j) then
5              output(i, j) = (i, j) // null otherwise
6  mainIteration(start: image of bool, input, output: image of int × int)
7      // parallel for on GPU
8      for (i, j) ∈ Coords do
9          if start(i,j) then
10             (i′, j′) = input(i, j) // pointer jumping
11             output(i, j) = maxNeighbour(input, i′, j′)
12 reconnect(start: image of bool, input, output: image of int × int)
13     // parallel for on GPU
14     for (i, j) ∈ Coords do
15         if start(i,j) then
16             (i′, j′) = input(i, j)
17             (a, b) = maxNeighbour(input, i, j)
18             (c, d) = input(i′, j′)
19             if (a, b) > (c, d) then
20                 output(i′, j′) = (a, b) // Requires atomic write
```

faster on CPU with 16 cores (a current high-end desktop workstation) than a machine with 2 cores (a current travelling laptop), as witnessed by the law on theoretical speed-up given by parallel machines [14]. Since the range of current CPUs is highly variable, so are the execution times in our tests. This fact also explains the different speedup in our tests comparing CPU and GPU on sequential and parallel tasks (see Figure 3 and Figure 4). In the parallel test, all the 16 cores of the chosen CPU are exploited, thus the CPU is more efficient.

We built two kinds of large formulas for stressing the tool: sequential (i.e. of shape $f(g(\dots(x))))$ and "parallel" ones, where the operators are composed in order to maximise parallelism. More precisely, formulas are written in order to have many independent sub-formulas (i.e., having shape $f(g(\dots,\dots),h(\dots,\dots)))$. In the CPU implementation, such sub-formulas can be computed in parallel, up to the number of available cores. Note again that maximising CPU usage entails a smaller speedup for the GPU. Figure 3 and 4 report execution times for each type of test. Each row reports the number of tasks to execute (i.e, the number of nodes in the directed acyclic graph described in Section 2), and the obtained speed-up for the two GPU algorithms. In all cases, `VoxLogicA-GPU` achieves a relevant speed-up. The CPU version performs better on very small formulas, due to the overhead needed to set up GPU computation. The version with garbage collection is much slower than the version without. This is due to garbage collection being run in the current implementation as soon as reference counts reach 0, and recall that memory deallocation and reallocation is particularly expensive on GPUs. Obvious improvements are expected by scheduling garbage collection



Iteration 1 Iteration 9 Iteration 13 Iteration 17 Iteration 24

Fig. 2. CC-labelling of a 2048 × 2048 pixels image in 24 iterations. Different colours represent different labels. Reconnect is called every 8 main iterations. Iteration 13: the main iterations converged; the image does not change until iteration 16 (reconnect). Iteration 17: label propagation after reconnect. Iteration 24: termination.

No. of Tasks	CPU	GPU		GPU-GC	
	Time	Time	Speed-up	Time	Speed-up
11	410ms	190ms	2.15	200ms	2.05
35	1470ms	190ms	7.73	230ms	6.39
67	1800ms	190ms	9.47	230ms	7.82
195	8200ms	200ms	41.00	320ms	25.62
259	10900ms	210ms	51.90	360ms	30.27
1027	43600ms	350ms	124.57	980ms	44.48
4099	174600ms	Out of memory	-	4100ms	42.58
8195	479000ms	Out of memory	-	12000ms	39.91

Fig. 3. Execution times for the sequential test.

to be run only when a memory usage threshold is reached. However, we plan to design a garbage collector which is more specific to the execution patterns of a model checker. We also carried out a preliminary assessment of the brain tumour segmentation case study of [7]. Given the current restrictions of `VoxLogicA-GPU` to 2D images and the core logical primitives (see Section 2), it is only possible to use a simplified dataset and specification, obtaining too small tasks for interesting measurements. We omit the full results (see [10]), but we note that a mild speed-up was obtained: this is interesting, as the CPU version uses a state-of-the-art imaging library designed for high efficiency.

4 Conclusions and Future Work

Our preliminary evaluation of spatial model checking on GPU is encouraging: large formulas benefit most, with significant speedups. Connected components labelling will be a focus for future work: indeed, the topic is very active, and our simple, proof-of-concept algorithm might well be replaced by state-of-the-art procedures (see e.g. the recent [1]). The currently attained speed-up can be used, for instance, for interactive calibration of parameters or for automated parameter optimisation, e.g. using gradient descent algorithms. However, given the peak performance of recent GPUs, our results are just the tip of the iceberg of what can be achieved. Future work will concentrate on fully exploiting more powerful

No. of Tasks	CPU	GPU		GPU-GC	
	Time	Time	Speed-up	Time	Speed-up
10	70ms	180ms	0.38	200ms	0.35
26	260ms	180ms	1.44	200ms	1.30
43	310ms	180ms	1.72	210ms	1.47
61	500ms	190ms	2.63	210ms	2.38
73	510ms	190ms	2.68	220ms	2.31
174	860ms	200ms	4.30	270ms	3.18
323	1600ms	220ms	7.27	340ms	4.70
472	2400ms	290ms	8.27	430ms	5.58
621	3000ms	360ms	8.33	510ms	5.88
1813	8800ms	Out of memory	-	1200ms	7.33
3005	14600ms	Out of memory	-	2000ms	7.30

Fig. 4. Execution times for the parallel test.

GPUs, using out-of-order execution to permit the execution of more independent tasks at the same time, and taking into account GPU-specific architectural features (memory banking, number of channels, etc.). Making `VoxLogicA-GPU` feature-complete with respect to `VoxLogicA` is also a goal. In this respect, we remark that although in this work we decided to go through the "GPU-only" route, future developments will also consider a *hybrid* execution mode with some operations executed on the CPU, so that existing primitives in `VoxLogicA` can be run in parallel with those that have a GPU implementation. Usability of `VoxLogicA-GPU` would be greatly enhanced by a user interface. However, understanding modal logical formulas is generally considered a difficult task, and cognitive/human aspects may become predominant with respect to technological concerns. Formal methods could be used to mitigate such concerns (see e.g. [9]).

References

1. Allegretti, S., Bolelli, F., Grana, C.: Optimized block-based algorithms to label connected components on GPUs. IEEE Transactions on Parallel and Distributed Systems **31**(2), 423–438 (2020)
2. Bakas, S., Akbari, H., Sotiras, A., Bilello, M., Rozycki, M., Kirby, J.S., Freymann, J.B., Farahani, K., Davatzikos, C.: Advancing the cancer genome atlas glioma MRI collections with expert segmentation labels and radiomic features. Scientific Data **4** (2017)
3. Banci Buonamici, F., Belmonte, G., Ciancia, V., Latella, D., Massink, M.: Spatial logics and model checking for medical imaging. Software Tools for Technology Transfer **22**(2), 195–217 (2020)
4. Bartocci, E., Bortolussi, L., Loreti, M., Nenzi, L.: Monitoring mobile and spatially distributed cyber-physical systems. In: Talpin, J., Derler, P., Schneider, K. (eds.) MEMOCODE 2017. pp. 146–155. ACM (2017)
5. Bartocci, E., Gol, E., Haghighi, I., Belta, C.: A formal methods approach to pattern recognition and synthesis in reaction diffusion networks. IEEE Transactions on Control of Network Systems **5**(1), 308–320 (2016)

6. Belmonte, G., Broccia, G., Ciancia, V., Latella, D., Massink, M.: Feasibility of spatial model checking for nevus segmentation. In: Bliudze, S., Semini, L. (eds.) FORMALISE@ICSE 2021. p. To appear (2021)
7. Belmonte, G., Ciancia, V., Latella, D., Massink, M.: Voxlogica: A spatial model checker for declarative image analysis. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 281–298. Springer (2019)
8. Berkovich, S., Bonakdarpour, B., Fischmeister, S.: GPU-based runtime verification. In: IPDPS 2013. pp. 1025–1036. IEEE Computer Society (2013)
9. Broccia, G., Milazzo, P., Ölveczky, P.C.: Formal modeling and analysis of safety-critical human multitasking. Innovations in Systems and Software Engineering **15**(3-4), 169–190 (2019)
10. Bussi, L., Ciancia, V., Gadducci, F.: A spatial model checker in GPU (extended version). CoRR **abs/2010.07284** (2020)
11. Ciancia, V., Latella, D., Massink, M., Paškauskas, R., Vandin, A.: A tool-chain for statistical spatio-temporal model checking of bike sharing systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 657–673 (2016)
12. Ciancia, V., Gilmore, S., Grilletti, G., Latella, D., Loreti, M., Massink, M.: Spatio-temporal model checking of vehicular movement in public transport systems. Software Tools for Technology Transfer **20**(3), 289–311 (2018)
13. Grosu, R., Smolka, S., Corradini, F., Wasilewska, A., Entcheva, E., Bartocci, E.: Learning and detecting emergent behavior in networks of cardiac myocytes. Communications of ACM **52**(3), 97–105 (2009)
14. Gustafson, J.L.: Reevaluating Amdahl's law. Communications of the ACM **31**(5), 532–533 (1988)
15. Ma, M., Bartocci, E., Lifland, E., Stankovic, J., Feng, L.: SaSTl: Spatial aggregation signal temporal logic for runtime monitoring in smart cities. In: ICCPS 2020. pp. 51–62. IEEE (2020)
16. Neele, T., Wijs, A., Bošnački, D., van de Pol, J.: Partial-order reduction for GPU model checking. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 357–374. Springer (2016)
17. Nenzi, L., Bortolussi, L., Ciancia, V., Loreti, M., Massink, M.: Qualitative and quantitative monitoring of spatio-temporal properties with SSTL. Logical Methods in Computer Science **14**(4) (2018)
18. Osama, M., Wijs, A.: Parallel SAT simplification on GPU architectures. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 21–40. Springer (2019)
19. Shiloach, Y., Vishkin, U.: An O(logn) parallel connectivity algorithm. Journal of Algorithms **3**(1), 57–67 (1982)
20. Tsigkanos, C., Kehrer, T., Ghezzi, C.: Modeling and verification of evolving cyber-physical spaces. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) ESEC/FSE 2017. pp. 38–48. ACM (2017)
21. Wijs, A., Bošnački, D.: Many-core on-the-fly model checking of safety properties using GPUs. Software Tools for Technology Transfer **18**(2), 169–185 (2016)
22. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: Unleashing GPU explicit-state model checking. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 694–701. Springer (2016)