Satoorn: A Source-to-Source Conversion Tool for Safety Critical Model Based Generated GPU Code

Marcos Rodríguez[†*], Alejandro J. Calderón[*], Leonidas Kosmidis[‡†], Irune Yarza[*]
[*]*Ikerlan Technology Research Centre*, Arrasate, Spain
[†]*Universitat Politècnica de Catalunya (UPC)*, Barcelona, Spain
[‡]*Barcelona Super Computing Center (BSC)*, Barcelona, Spain

***Abstract*— Model-Based Design (MBD) and safety-critical standards intersect in many ways that can enhance the product development process. Safety-critical systems require rigorous methodologies to ensure reliability, while MBD leverages mathematical models to optimise system development and reduce human errors. Traditionally, these fields have been applied to CPU-based architectures, with extensive literature documenting their success. However, the introduction of Graphics Processing Units (GPUs) in heterogeneous computing presents new challenges for safety compliance. These challenges arise, among others, from the parallel nature of GPUs, their complex memory management, which complicates the prediction of program execution times, and the use of commonly used programming languages that introduce memory vulnerabilities through pointers and dynamic memory allocation. In this work, we present Satoorn, a tool that bridges the gap between MBD-generated code and safety-compliant languages for GPUs. Satoorn transforms Simulink-generated CUDA code into Brook Auto, a safety-critical compliant GPU language according to ISO 26262. This tool enhances the integration of MBD artefacts into safety-critical systems, ensuring adherence to stringent safety standards. We evaluate this approach using CUDA code generated from a Simulink model for lane tracking example representing part of an image-processing pipeline for automotive sensors.**

I. Introduction

Safety-critical systems are those where failure or malfunction could result in serious harm to people, the environment, or significant financial loss. To ensure such systems meet rigorous requirements, a range of standards and norms have been established across industries. MBD has emerged as a robust framework for developing these systems, using mathematical models to simulate, analyse, and optimise complex behaviour prior to physical implementation. It has seen widespread adoption in sectors such as aerospace [1], automotive [2], medical devices [3], industrial automation [4], railway [5], and nuclear energy [6].

MBD and safety-critical development are naturally complementary: both rely on structured, analysable artefacts that support rigorous verification, both place traceability at the core of the development lifecycle, and MBD's high-level modelling languages reduce the likelihood of human error during system specification.

However, despite these common points, MBD and safety certification requirements have not always worked together; a notable success case is Chami's work [7], where Model-Based Systems Engineering (MBSE) techniques model the operational aspects of railway systems and integrate them with requirements and safety analysis.

The parallel and specialised nature of General Purpouse Graphics Processing Units (GPGPUs) challenges compliance with safety standards, requiring innovative solutions beyond traditional safety-critical system design. Among other factors, the memory hierarchy, non-deterministic execution, and lack of formal verification support inherent to GPGPU architectures make it difficult to provide the required guarantees. Furthermore, the most widely adopted programming languages for GPGPU development—most notably CUDA—were designed with performance as the primary objective, with little consideration for safety-critical requirements. As a result, they lack the necessary properties for certification, such as deterministic behaviour, restricted memory access, and compliance with coding guidelines. This gap between the capabilities of current GPGPU programming models and the demands of safety-critical domains motivates the need for alternative approaches.

In this work, we address one of the key limitations identified above: the lack of safety-compliant GPU programming languages suitable for MBD-generated code. To this end, we present **Satoorn**, a software tool that automatically transforms CUDA code generated from Simulink models into Brook Auto [8] / BRASIL [9], a GPU programming language designed for safety-critical applications. Satoorn is built on the Clang-C++ compiler infrastructure and leverages Abstract Syntax Tree (AST) analysis to perform structured, semantics-preserving code transformations, enabling a conversion process that bridges MBD workflows with certifiable GPU programming environments. Satoorn's functionality is demonstrated through a lane detection and tracking ADAS use case modeled in Simulink.

The rest of the paper is organised as follows: Section II presents the background, Section III the related work, Section IV describes Satoorn's transformation pipeline, Sections V and VI present the use case and its evaluation, and Section VII concludes.

II. Background

Ensuring safety in autonomous systems requires adherence to rigorous regulatory standards. This section begins by introducing MBD as a structured path to compliance, then outlines key regulations for safety-critical software, explores the evolution of safety-oriented GPU languages, and concludes

with the role of Clang C++ and Abstract Syntax Trees (AST) in code analysis.

A. Model-based design

In safety-critical sectors, MBD is typically applied within structured development processes such as the V-Model, where high-level models are simulated, verified, and validated throughout the lifecycle before being automatically transformed into deployable code [10], [11]. This automatic code generation step is a key enabler in safety-critical workflows: it eliminates a significant source of human error and ensures that the implementation faithfully reflects the validated design. Industrial tools such as Simulink by MathWorks, LabVIEW by National Instruments, and ANSYS are prominent examples of MBD environments supporting this workflow.

Of particular relevance to this work is the capability of certain MBD tools to target heterogeneous computing platforms. Simulink, through its GPU Coder toolbox, supports automatic code generation for GPU architectures, enabling the deployment of model-based designs on massively parallel hardware. However, the GPU code produced by this toolchain is exclusively generated in CUDA. While this language is the dominant one for GPU development, it was conceived with performance as the primary objective and lacks the properties required for functional safety certification, such as deterministic memory access, restricted language constructs, and compliance with coding guidelines mandated by standards.

B. Safety-critical regulation for software

ISO 26262 is an international functional safety standard for road vehicles, covering the entire development lifecycle of safety-critical automotive software and hardware. Its Part 6 specifically addresses software-level development and imposes a set of concrete recommendations that directly shape how safety-critical code must be structured.

Among others, Recommendation 1b of Table 1 calls for the exclusion of language constructs that may result in unhandled run-time errors, while Recommendation 1e supports the use of consistent naming and structural conventions. Recommendation 1h addresses the risks of dynamic memory allocation, which introduces nondeterministic behaviour due to runtime memory management and potential fragmentation, and Recommendation 1i of Table 3 establishes requirements for the appropriate management of shared resources, reinforcing the need for explicit memory ownership and access control. Finally, a general requirement running throughout ISO 26262-6 calls for controlled concurrency and explicit synchronisation mechanisms, so as to avoid unintended timing and data hazards.

C. Safety compliant GPU programming languages

As highlighted in section B, safety-critical GPU programming requires languages and frameworks that comply with stringent standards such as ISO 26262 [12], which impose constraints on control flow, memory management, and pointer usage. In this context, Khronos Group [13] introduced certifiable/qualifiable subsets of these languages, named OpenGL SC 2.0 [14], Vulkan SC [15] and SYCL SC [16]. OpenGL SC and Vulkna SC have been utilised in automotive safety-critical systems like Advanced Driver-Assistance Systems (ADAS), avionics displays, industrial automation, and medical imaging equipment. Despite their suitability for safety-critical industries, the low-level nature of OpenGL SC 2.0 and Vulkan SC poses challenges in development due to their excessive amount of code they require, demanding meticulous error handling and debugging, while SYCL SC, a higher level language, is still under development.

Following ISO 26262 standards, Brook Auto [8] / BRASIL [9] emerged as an iteration of Brook [17], integrating OpenGL SC 2.0 as its backend and refining the Brook frontend to match safety-critical programming guidelines. Drawing from Brook's methodology, Brook Auto retains features such as opaque structures (*streams*) utilisation over pointers and static memory allocation, while introducing a requirement for static data size definition. Also, it forbids recursion and imposes kernels to have only one output argument. Notable modifications include: ensuring full code coverage by eliminating dead code, restricting pointer usage, refining runtime design for deterministic execution, and optimising graphics API compatibility for ISO 26262. Table I provides a summary of GPU programming languages mentioned before.

TABLE I: Comparative evaluation of safety–compliant GPU languages

Criteria	OpenGL SC 2.0	Vulkan SC	SYCL SC	Brook Auto BRASIL
Safety certification	++	++	0	++
Abstraction level	– –	– –	0	–
Ease of development	– –	– –	0	–
Learning curve	– –	– –	0	–
Control exec./memory	+	++	0	+
Runtime predictability	+	++	0	+
Memory management	++	+	0	++
Stream–based computation	++	+	0	++
Flexibility (general–purpose)	–	+	0	– –
Tooling / maturity	+	+	– –	–
Debugging complexity	– –	– –	0	–

D. Clang-Based Code Analysis and Transformation

Clang is a fast, modular compiler frontend for the C language family, compatible with GCC and MSVC, and using Low Level Virtual Machine (LLVM) to optimise and generate code for multiple architectures. Its modularity supports integration with analysis, refactoring, and code generation tools.

A key feature is its ability to build and manipulate Abstract Syntax Trees (ASTs), structured representations of code used for analysis, validation, refactoring, and automated transformations. Developers can programmatically access ASTs to create custom tools.

III. Related Work

Paul Sathre et al. [18] present CU2CL, a source-to-source translator from CUDA to OpenCL aimed at extending GPU

code beyond NVIDIA platforms. CU2CL preserves code formatting and comments to maintain readability.

Another alternative was built by Robert Mueller-Albrecht [19] who developed a tool called SYCLomatic, enabling code migrations from CUDA code to ready-to-build C++ with SYCL code, leveraging a YAML rule file. SYCL is an open Khronos standard for heterogeneous computing based on standard C++.

ReSYCLator [20] is an Eclipse-CDT plug-in that automatically transforms CUDA C++ into SYCL C++ [21]. Due to weak Nsight™ integration with CDT's analysis tools, the authors developed a custom CUDA-C++ analyser (CRITTER) to enable AST-based transformations.

However, unlike our work, none of these works focus on safety critical GPU language subsets nor have considered tool qualification and target model based design development.

IV. Problem formulation and proposing approach

GPUs are powerful computing accelerators, but integrating them into safety-critical frameworks is still challenging, even with MBD and code generators. The Simulink code generator only produces CUDA, which is considered unsafe due to its use of dynamic allocations and pointers as described in Section II-C.

To address these limitations, the *Satoorn* tool was developed as a bridge between advanced parallel computing platforms and the requirements of safety-critical systems. **It transforms the CUDA code generated from Simulink into an equivalent higher-level representation in Brook Auto**, which serves as an abstraction layer over OpenGL SC 2.0. This translation process, while enabling code reuse and reducing development overhead, presents notable challenges due to fundamental differences in language paradigms: CUDA follows a thread-centric model, whereas Brook Auto adopts a stream-centric approach. These differences, along with variations in memory management, syntax, and supported operations, require meticulous code parsing and restructuring. To streamline Satoorn use, we have developed a user interface in Qt Creator that orchestrates the transformation pipeline step by step, as detailed in the following sections.

Simulink generated files are organised according to a specific structure, as shown in Figure 1, which provides a visual representation of their arrangement. The following enumeration provides a detailed breakdown of the generation:

- **Project_name.cu:** contains variables, functions and calls that involve the use of the GPU, such as *cudaMalloc*, *cudaMemcpy* or kernels.
- **Project_name.h:** contains the class that defines the model, structures of data and functions used by the model.
- **Project_data.cu:** In this file, the values of the data are initialised.
- **Project_name_private.h:** contains local define constants and local data for the model and subsystems.
- **Project_name_types.h:** provides forward declarations for the real-time model data structure and the parameters data structure.
- **rtwtypes.h:** As Simulink employs its own variable types and structures, this file redefines C standard types.
- **ert_main.cu:** this is the main file of the project; its name depends on the toolbox used. It contains the project class and functions used to launch the program.

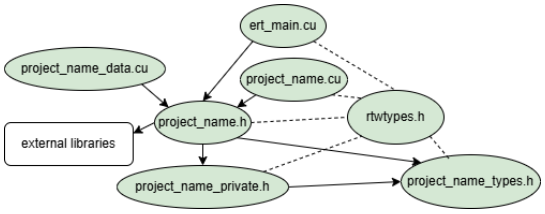


Fig. 1: Generated files dependencies

Leveraging Clang's parsing capabilities for C++ and CUDA, we were able to detect and transform parts of the Simulink's generated code as needed. We have outlined five distinct steps that are executed sequentially to ultimately convert files into Brook Auto syntax. The order of execution is crucial as it ensures that the code compiles successfully after each transformation, and that the output remains compatible with the input of the subsequent phase. In the following content, we will delve deeper into each phase with dedicated subsections, namely:

- **Phase 1 — Global changes:** Global changes: type, cast and comment normalisation across the whole source.
- **Phase 2 — GPU functions splitting:** GPU functions splitting: removal of structures/classes from kernel interfaces.
- **Phase 3 — File distribution:** File distribution: reorganisation of the code into the target file structure.
- **Phase 4 — Syntax change:** Syntax change: conversion into Brook Auto syntax.
- **Phase 5 — File creation and format changer:** File creation and format change: Makefile/main generation and final file formats.

A. Phase 1: Global changes

The objective of this phase is to convert Simulink types into C-style types. Despite the presence of *rtwtypes.h*, Brook Auto struggles to comprehend redefinitions. This transformation is achieved through three distinct stages: a resizer, a type fixer and a comment erasing.

The "resizer" scrutinises *project_name.cu* and *project_name.h*, identifying Simulink types and adjusting the allocation type based on the standard C++ as summarised in Table II. Additionally, the script transforms the way of static casts to C++ standard and adds a directive to ensure that arrays are 16-bit aligned. Finally, a python script detects C++ style comments in the code that begin with "//" erasing the line, while C-style comments between "/*" and "*/" are allowed.

At the end of phase 1 the following files are parsed and copied into a different file to keep intact the original files generated by Simulink:

TABLE II: Phase 1 type transformations

Simulink generated type	Target type
(const) (u)int64_T / (u)int64_t	(const) (unsigned) int
(const) (u)int32_T	(const) (unsigned) int
(const) (u)int16_T	(const) (unsigned) int
(const) (u)int8_T	(const) (unsigned) char
(const) (u)real_T / (u)real32_T	(const) (unsigned) float
double	float
pointer_T	void
(const) (u)char_T	(const) (unsigned) char
boolean_T	unsigned char

TABLE III: Phase 1 file transformation

Input File	**Output File**
(Original) project_name.cu	(P1) project_name_fixed.cu
(Original) project_name.h	(P1) project_name_fixed.h

B. Phase 2: GPU functions splitting

Brook Auto kernels have two main restrictions: they cannot take structures or classes as arguments, and they use only one output variable. This limitation is inherited from Brook Auto's OpenGL SC 2.0 backend, which allows a single framebuffer output per GPU fragment shader. Writing to multiple arguments via & is also prohibited due to the lack of pointers. This phase detects such cases, extracts the needed internal data, and duplicates functions as required.

For *project_name_fixed.cu*, the tool analyses kernels and stores their definitions in a vector for easy access. Subsequent steps duplicate kernel declarations, functions, calls, and related CPU–GPU copies, memory allocations, and variable declarations.

At the end of phase 2, the following files are going to be parsed; at this moment, changes are overwritten in the same file:

TABLE IV: File transformation from Phase 1 to Phase 2

Input File	**Output File**
(P1) project_name_fixed.cu	(P2) project_name_fixed.cu
(P1) project_name_fixed.h	(P2) project_name_fixed.h

C. Phase 3: File distribution

This phase arranges the output of *project_name_fixed.cu* to approximate the target architecture, organising kernel declarations, definitions, allocations, variables, and calls into their designated source files. At this phase, we still retain the CUDA notation because each time a Clang program is executed, the project is compiled.

TABLE V: File transformation from Phase 2 to Phase 3

Input File	**Output File**
(P2) project_name_fixed.cu	(P3) project_name_fixed.cu (P3) SatoornKernel_BRHI.h (P3) SatoornKernel_BRI.cu

1) **project_name_fixed.cu**: global and device kernel declarations and functions are removed, along with the include and define directives; the include directive to "SatoornKernel_BRHI.h" is added.
2) **SatoornKernel_BRHI.h**: contains global and device kernel declarations.
3) **SatoornKernel_BRI.cu**: contains global and device kernel functions, including directive to *"SatoornKernel_BRHI.h"* is added.

D. Phase 4: Syntax change

This phase focuses on syntax modifications to transition CUDA-generated code into Brook Auto's programming style. This transformation occurs in six stages:

1) Modify kernel declarations and functions to remove pointers, adapt output variables, add index variables, and adjust thread configuration.
2) Adapt GPU memory allocation to Brook Auto's *stream* notation.
3) Simplify GPU→CPU and CPU→GPU copies using **streamWrite** and **streamRead**.
4) Transform GPU→GPU copies by selecting the appropriate function based on source and destination.
5) Adjust kernel calls by removing pointers, adapting indices, and removing CUDA thread configuration.
6) Remove explicit GPU memory deallocation.

At the end of phase 4, the following files are going to be parsed:

TABLE VI: File transformation from Phase 3 and Phase 2 to Phase 4

Input File	**Output File**
(P3) SatoornKernel_BRHI.h	(P4) SatoornKernel_BRHI_improved.h
(P3) SatoornKernel_BRI.cu	(P4) SatoornKernel_BRI_improved.cu
(P3) project_name_fixed.cu	(P4) project_name_fixed_improved.cu
(P2) project_name_fixed.h	(P4) project_name_fixed.h

E. Phase 5: File creation and format changer

A templated Makefile is generated from the project's header, ensuring consistent variable and function naming. Using Clang's matcher, the header is parsed to locate the class that stores core project data, forming the basis for later modifications and enabling seamless integration of new functionality. The main file only needs a format change.

To ensure proper compilation, GPU functions in *SatoornFunctions.br* are isolated from CPU code, since *.br, .bri*, and *.brhi* files are compiled with *brcc*, while *.cpp, .hpp*, and *.h* files use *gcc*. For this, a new file, *SatoornFunctionsCPU.cpp*, is created to host CPU-only functions and call the GPU routines defined in *SatoornFunctions.br*. Figure 2 illustrates the final architecture. A Python script then generates renamed and reformatted copies of the "core" files, updating includes and preserving consistency, as summarised in Table VII.

TABLE VII: File transformation from Phase 4 and Original to Phase 5

Input File	Output File
(P4) SatoornKernel_BRHI_improved.h	(P5) SatoornKernel.brhi
(P4) SatoornKernel_BRI_improved.cu	(P5) SatoornKernel.bri
(P4) project_name_fixed.h	(P5) Satoorn_header_declassified.h
(P4) project_name_fixed_improved.cu	↗ (P5) SatoornFuntions.br ↘ (P5) SatoornFunctionsCPU.cpp
(Original) project_name_data.cu	(P5) project_name_data.cpp
(Original) *_main.cu	(P5) Satoorn_main.cpp

The final architecture from Satoorn's output is illustrated in Figure 2.

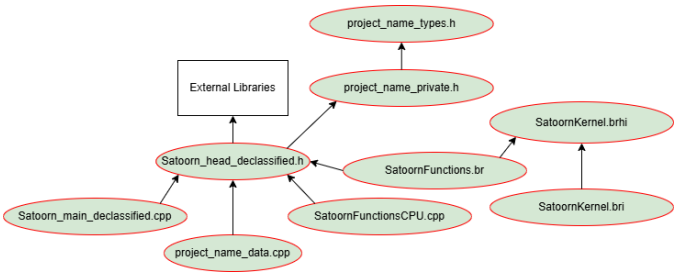


Fig. 2: Final file architecture

V. Use Case Definition

Satoorn is evaluated with a Simulink use cases with CUDA GPU code generation, in order to demonstrate its automation and the functional equivalence of the Brook Auto generated code.

The experimental evaluation presented in this work is conducted on an NVIDIA Jetson Orin AGX, which reflects a more capable embedded target used in automotive and edge computing scenarios.

The use case considered in this work is inspired by a lane detection scenario employed in ADAS, as described by Anumakonda [22]. Lane detection is a representative example of a safety-relevant workload, as it contributes to passenger protection by enabling functions such as lane keeping assistance. This task requires the real-time processing of high amounts of sensor's data, making it well suited for acceleration on GPUs.

Based on this reference scenario, a corresponding lane detection pipeline was modelled in Simulink, as illustrated in Figure 3. The Simulink model implements a complete lane detection pipeline composed of four main computational stages, each mapped to a MATLAB Function block and configured for GPU code generation. The pipeline operates on full-resolution RGB frames of size 1280 x 720 pixels and produces an annotated output image in which the detected lane area and lane boundaries are overlaid onto the original input.

The pipeline extracts lane candidates via grayscale conversion, Sobel gradients, and HSV thresholding, then applies a perspective transformation to obtain a bird's-eye view. Lane boundaries are detected using a sliding-window approach and finally projected back onto the original frame as a coloured

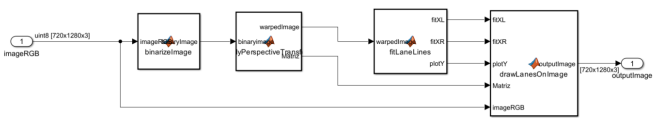


Fig. 3: Simulink model of the lane detection pipeline developed in this work

drivable region. The resulting visualization is combined with the input frame to produce the final output image, as illustrated in Figure 4.



Fig. 4: Frame of the Simulink model output

VI. Results

In the following paragraphs, we present examples of code transformations, including those applied to the kernel "sobel_prueba_7_Outputs_kernel1" and other representative parts of the code.

Listing 1 shows a typical kernel declaration generated by Simulink. In Listing 2, the tool detects the output as an object with two modified variables in the original code. So, it generates separate kernels for each variable. This Brook Auto segment appears in the *SatoornKernel.brhi* file in Figure 2.

Listing 1: Example of kernel declaration generated by Simulink

```
static __global__ __launch_bounds__(512, 1) void
    sobel_prueba_7_Outputs_kernel1
  (uint8_T rtb_imag_0[2764800]);
```

Listing 2: Example of kernel declaration in Brook Auto

```
/*KERNEL STATEMENT Detected*/ kernel void
    sobel_prueba_7_Outputs_kernel1(out unsigned char
    rtb_imag_0<>, unsigned char rtb_imag_0_prime<>);
```

Listing 3 shows the corresponding CUDA kernel generated by Simulink. The kernel uses a scattering strategy to write the output, which Brook Auto prohibits, requiring a gathering strategy as in Listing 4. Although this can be detected at compilation, the tool cannot yet handle it automatically due to variations in scattering syntax. For now, it flags such cases during syntax conversion and suggests manual fixes. This

Brook Auto segment is part of the *SatoornKernel.bri* file in Figure 2.

Listing 3: Example of kernel function generated by Simulink

```
static __global__ __launch_bounds__(512, 1) void
    sobel_prueba_7_Outputs_kernel1 (uint8_T rtb_imag_0
    [2764800])
{
  int32_T i;
  i = static_cast<int32_T>(mwGetGlobalThreadIndex());
  if (i < 2764800) {
    rtb_imag_0[i] = static_cast<uint8_T>(0U);
  }
}
```

Listing 4: Example of kernel function in Brook Auto

```
kernel void sobel_prueba_7_Outputs_kernel1(out unsigned
    char rtb_imag_0<>, unsigned char rtb_imag_0_prime<>)
{
    rtb_imag_0 = 0;
}
```

Listings 5 and 6 show GPU instructions for data transfers and kernel calls, with duplicates to match earlier changes and copy types consistent with CUDA. For *DeviceToDevice* copies, Brook Auto decides between *streamRead* or *streamWrite*. This code is in *SatoornFunctions.br* (Figure 2).

Brook Auto groups these functions under *brookFnc_part8* to avoid compiling with CPU code containing structures. Corresponding calls, like those are made from *SatoornFunctionsCPU.cpp* and declared in *Satoorn_head_declassified.h*.

Listing 5: Example of kernel call generated by Simulink

```
 sobel_prueba_7_Outputs_kernel26<<<dim3(1U, 1U, 1U), dim3
     (32U, 1U, 1U)>>>
   (*gpu_pts, *gpu_Tvec);
 sobel_prueba_7_Outputs_kernel27<<<dim3(1U, 1U, 1U), dim3
     (32U, 1U, 1U)>>>
   (*gpu_accumulatedData, c_sumOfPointDistancesFromOrigin,
       *gpu_tmp, *gpu_Tvec);
 sobel_prueba_7_Outputs_kernel28<<<dim3(1U, 1U, 1U), dim3
     (32U, 1U, 1U)>>>
   (*gpu_tmp, *gpu_Tvec, *gpu_B);
```

Listing 6: Example of kernel call in Brook Auto

```
void Satoorn_Fnc12(float c_sumOfPointDistancesFromOrigin){
  int i, j;

  /*Kernel CALL Detected*/ sobel_prueba_7_Outputs_kernel26(
      gpu_pts, gpu_Tvec, gpu_Tvec);
  /*Kernel CALL Detected*/
      sobel_prueba_7_Outputs_kernel27_Tvec(gpu_Tvec,
      gpu_Tvec, gpu_Tvec);
  /*Kernel CALL Detected*/
      sobel_prueba_7_Outputs_kernel27_tmp(
      gpu_accumulatedData, c_sumOfPointDistancesFromOrigin,
       gpu_tmp, gpu_tmp);
  /*Kernel CALL Detected*/ sobel_prueba_7_Outputs_kernel28(
      gpu_tmp, gpu_Tvec, gpu_B, gpu_B);
}
```

Beyond structural transformation, the functional equivalence of the generated code was assessed through an automated golden-reference comparison: both implementations processed the same input video, and the final RGB output frame (1280×720) was dumped at each iteration and compared channel-wise over 30 consecutive frames, where a pixel is bit-identical only if all three RGB channel values match exactly.

TABLE VIII: Functional equivalence of CUDA and Brook Auto outputs over 30 consecutive frames

Metric	Mean	Worst frame
Bit-identical pixels (%)	99.996	99.993
SSIM, per channel (4 d.p.)	1.0000	0.9999
PSNR, combined (dB)	87.3	61.8
PSNR, worst channel (B) (dB)	82.9	58.6
MAE (per channel, 0–255)	0.0001	0.0004
Differing pixels (of 921 600)	41.7	64

The CUDA reference executes on the GPU, whereas the Brook Auto version runs on the BRT CPU backend of the PIL setup: Brook Auto maps kernels onto OpenGL SC 2.0 fragment shaders, and the current memory allocation strategy of this mapping reserves substantially more memory than the application requires, exceeding the capacity of the target GPU and preventing deployment on it — an implementation aspect of the backend whose resolution falls outside the scope of this work. Independent of the execution backend, bit-exact equality cannot be expected: floating-point operation ordering and fused multiply-add (FMA) usage differ across architectures and compilers, so the same expression can legitimately differ in the last place between the two implementations. Equivalence is therefore evaluated through image similarity metrics. Table VIII summarises the results: on average, 99.996% of pixels are bit-identical per frame, with a per-channel Structural Similarity Index Measure (SSIM) of at least 0.9999, a Peak Signal-to-Noise Ratio (PSNR) above 58 dB per channel, and a Mean Absolute Error (MAE) below 0.0005 in every frame. The remaining differences (∼42 pixels per frame, out of 921 600) are confined to the boundary of the rendered lane overlay and are consistent with sub-pixel floating-point rounding rather than algorithmic divergence, confirming that the transformation preserves the functional behaviour of the original pipeline.

VII. Conclusion and Future Work

We managed to transform a representative use case into an executable solution that complies with functional safety standards. This confirms the viability of our approach and shows that it can effectively connect model-based design with safety-aware execution environments.

At this stage, the parser is still in its early phase. Evaluations have been carried out in a Processor-in-the-Loop (PIL) setup, with a focus on validating functionality rather than performance. Moving towards a Hardware-in-the-Loop (HIL) configuration will be an important milestone, since it will allow us to assess real-time behaviour and robustness under conditions much closer to the target hardware.

Looking ahead, two directions would further strengthen the tool: automating the CPU/GPU code partitioning (*SatoornFunctionsCPU.cpp* vs. *SatoornFunctions.br*), which would remove the remaining manual step of the pipeline, and translating the few cases where scatter-based operations (e.g., histograms) appear onto a gather-based model.

ACKNOWLEDGEMENTS

The research presented throughout this paper has received funding from the CERVERA programme within the framework of the MEDUSA project (grant no. CER-20231011) and Horizon Europe programme under the UP2DATE4SDV project (grant no. 101203060). Claude Opus 4.8 was used for language editing improve grammar and clarity; all edits were subsequently reviewed and revised by the authors.

REFERENCES

[1] X. Yang, Z. Wu, Y. Hou, and J. Liu, "Mbd attributes template method of aeronautical products," *MATEC Web of Conferences*, vol. 139, p. 00017, 01 2017.

[2] B. R. Mudhivarthi, V. Saini, A. Dodia, P. Shah, and R. Sekhar, "Model based design in automotive open system architecture," in *2023 7th International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2023, pp. 1211–1216.

[3] Digital Engineering 247, "Model-based design for medical devices," 2015, accessed: 2025-02-25. [Online]. Available: https://www.digitalengineering247.com/article/model-based-design-for-medical-devices

[4] K. Kshirsagar, P. Shah, and R. Sekhar, "Model based design in industrial automation," in *2022 6th International Conference On Computing, Communication, Control And Automation (ICCUBEA*, 2022, pp. 1–6.

[5] J. Barrett, N. Wang, J.-M. Dalbavie, and C. Carricajo, "Agile model-based systems engineering of passenger train operational design," *Transportation Research Procedia*, vol. 72, pp. 295–302, 2023, tRA Lisbon 2022 Conference Proceedings Transport Research Arena (TRA Lisbon 2022),14th-17th November 2022, Lisboa, Portugal. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352146523007044

[6] J. K. Corrado, "Model-based systems engineering adoption in the u.s. nuclear industry," *Nuclear Engineering and Design*, vol. 432, p. 113752, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0029549324008525

[7] M. Chami, P. Oggier, O. Naas, and M. Heinz, "Real world application of mbse at bombardier transportation," 09 2015.

[8] M. M. Trompouki and L. Kosmidis, "Brook auto: High-level certification-friendly programming for gpu-powered automotive systems," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[9] ——, "Brasil: A high-integrity gpgpu toolchain for automotive systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 2019, pp. 660–663.

[10] MathWorks, "Why adopt model-based design white paper," Tech. Rep., 2021. [Online]. Available: https://www.mathworks.com/content/dam/mathworks/white-paper/why-adopt-model-based-design-white-paper.pdf

[11] A. Bergmann, "Benefits and drawbacks of model-based design," *KMUTNB International Journal of Applied Science and Technology*, vol. 7, pp. 15–19, 09 2014.

[12] I. O. for Standardization, "Iso 26262-6:2018(en) road vehicles — functional safety — part 6: Product development at the software level," 2018. [Online]. Available: {https://www.iso.org/obp/ui/en/#iso:std:iso:26262:-6:ed-2:v1:en}

[13] Khronos Group, "The khronos group inc," 2010. [Online]. Available: {https://www.khronos.org/}

[14] K. Group, "Opengl sc overview — the khronos group inc," 2020. [Online]. Available: {https://www.khronos.org/openglsc/}

[15] ——, "Vulkan sc overview — the khronos group inc," 2020. [Online]. Available: {https://www.khronos.org/vulkansc/}

[16] ——, "Sycl sc overview — the khronos group inc," 2023. [Online]. Available: {https://www.khronos.org/syclsc/}

[17] I. Buck, "Brook for gpus: Stream computing on graphics hardware," 2004.

[18] P. Sathre and G. Martinez, "Cu2cl (cuda-to-opencl)," Tech. Rep., 2017. [Online]. Available: https://chrec.cs.vt.edu/cu2cl/

[19] R. Mueller-Albrecht, "Syclomatic: Sycl adoption for everyone - moving from cuda to sycl gets progressively easier: Advanced migration considerations," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3648115.3648124

[20] T. Stauber and P. Sommerlad, "Resyclator: Transforming cuda c++ source code into sycl," in *Proceedings of the International Workshop on OpenCL (IWOCL)*, 2019.

[21] Khronos Group, "Sycl: Single-source c++ programming for opencl," 2020, accessed: 2024-11-14. [Online]. Available: {https://www.khronos.org/sycl/}

[22] S. Anumakonda, "Pairing lane detection with object detection," 2021, accessed: 2025-12-30.