The Journal of Open Source Software

ReProspect - A framework for reproducible prospecting of CUDA applications

Romin Tomasetti [1*¶] **and Maarten Arnst** [1*]

1 University of Liège, Belgium ¶ Corresponding author * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Neea Rusch

Reviewers:

- @dinesh-rt
- @cedricchevalier19

Submitted: 31 December 2025
Published: unpublished

Summary

`ReProspect` is a Python framework designed to support reproducible prospecting of CUDA code—that is, the systematic analysis of CUDA-based libraries and software components through API tracing, kernel profiling, and binary analysis.

`ReProspect` builds on NVIDIA tools: Nsight Systems, Nsight Compute, and the CUDA binary utilities. It streamlines data collection and extraction using these tools, and it complements them with new functionalities for a fully programmatic analysis of these data, thus making it possible to encapsulate the entire prospecting analysis in a single Python script.

`ReProspect` provides a practical foundation for developing novel use cases of CUDA code prospecting. It supports collaborative code development by enabling developers to share concise, reproducible analyses that motivate design decisions and help reviewers grasp the impact of proposed changes. It also enables new types of tests that go beyond traditional output-correctness validation in CI/CD pipelines, such as validating the presence of instruction patterns in binaries or confirming expected API call sequences for key library functionalities. Additionally, `ReProspect` can act as a framework for structuring research artifacts and documenting analyses, enabling others to reproduce the work and build upon it more effectively.

Statement of need

HPC software development strives to achieve performance and sustain it over time, while hardware and software evolve. However, the modern programming landscape relies on complex software stacks and compiler toolchains. Therefore, tools are needed to analyse the interaction of the code with other layers across the stack and ultimately with the hardware.

The ability to carry out the analysis fully programmatically ensures reproducibility of the results by others while opening a range of new use cases that can be integrated in the development cycle. For instance, whereas test suites traditionally check output correctness of public functionalities, they could also verify application runtime events, kernel performance, or generated machine code.

For the CUDA stack, NVIDIA provides a set of proprietary tools guaranteed to be up-to-date with their software and hardware. The runtime analysis tools Nsight Systems (NVIDIA Corporation, 2025d) and Nsight Compute (NVIDIA Corporation, 2025c) are designed for API tracing and kernel profiling, respectively. They both provide a GUI for exploring the results, as well as a low-level Python API for accessing the raw data. The CUDA binary utilities (NVIDIA Corporation, 2025a) provide command-line access to machine code (SASS or PTX (NVIDIA Corporation, 2025f)) and other information embedded in the binaries. However, while these tools allow raw data to be extracted, they do not themselves provide the infrastructure for effective programmatic analysis.

State of the field

Several well-established open-source tools are already available. Caliper (Boehme et al., 2016) can intercept CUDA API calls through the NVIDIA CUPTI library (NVIDIA Corporation, 2025b). It can interface with the Python package Hatchet (Bhatele et al., 2019) to organize results into a hierarchical data structure. Thicket (Brink et al., 2023) adds kernel profiling support through Nsight Compute, with a primary focus on exploratory data analysis of multi-run performance experiments. HPCToolkit (Zhou et al., 2021) is another comprehensive suite designed for large-scale parallel systems. It includes CUDA API tracing through CUPTI and kernel profiling through PAPI (Terpstra et al., 2010), and it has binary analysis capabilities to attribute performance data to calling contexts. It has a visual interface, and it can output raw performance data for programmatic analysis, *e.g.* using Hatchet. Score-P (Knüpfer et al., 2012) integrates multiple performance analysis tools in a common infrastructure. It can record CUDA API calls and GPU activities through CUPTI and provides standardized data formats.

Although script-driven runtime analysis is also possible with these well-established tools, developing `ReProspect` as an independent package enables a design optimised for our use cases: concise, reproducible, low-overhead, easy-to-adopt, script-driven analysis of individual units of functionality. Beyond runtime analysis, `ReProspect` introduces new binary analysis functionalities to inspect machine code for expected instruction sequence patterns. To the best of our knowledge, these functionalities are not covered by existing tools.

Software design

`ReProspect` is organized into three main components: API tracing, kernel profiling, and binary analysis (Figure 1).

Each component is designed to allow the entire analysis to be encapsulated into a concise Python script. This includes launching the underlying analysis tool, collecting the output into Python data structures, and performing the subsequent analysis.

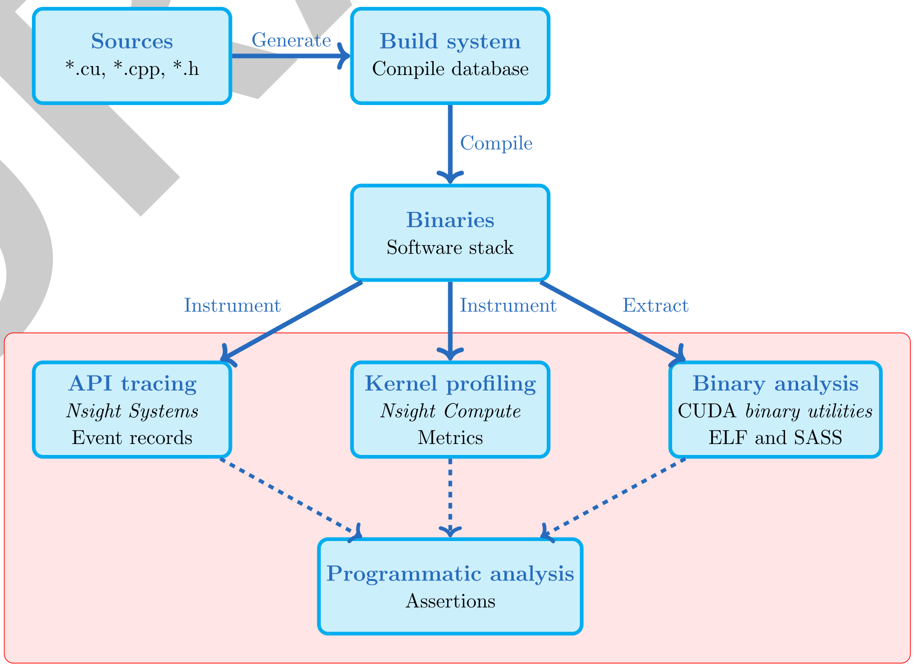

Figure 1: Overview of `ReProspect`.

API tracing and kernel profiling

The `ReProspect` `Command` and `Session` classes streamline launching Nsight Systems and Nsight Compute to collect a focused set of metrics most relevant for the analysis. The collected data are gathered in a `Report`, queryable by NVTX range annotations (NVIDIA Corporation, 2025e), readily amenable to test assertions.

To avoid unnecessary re-runs, `ReProspect` provides a `Cacher` that can serve the `Report` from a database.

Binary analysis

`ReProspect` provides a set of tools for extracting and analysing the content of CUDA binaries.

The `CuObjDump` and `NVDisasm` classes drive and parse the output of the underlying CUDA binary utilities to retrieve the SASS code and resource usage of kernels (*e.g.* registers, constant memory).

The `ELF` class decodes ELF-formatted sections to extract complementary information, including the symbol table, toolchain metadata (from the *.note.nv.tkinfo* section), and kernel attributes such as launch bounds (from the *.nv.info.<kernel>* section) (Hayes et al., 2019).

Beyond data extraction, `ReProspect` provides an extensible framework for inspecting machine code for expected instruction patterns. This framework is structured as a hierarchy of matchers. At the lowest levels, matchers analyse instructions and their components (opcodes, modifiers, and operands). These matchers can be composed into instruction sequence matchers, enabling the identification of more intricate patterns.

One of the key challenges with SASS matching is the evolution of the CUDA instruction set and compiler toolchains. `ReProspect` addresses this challenge by abstracting away architecture- and toolchain-specific details at each level of the matcher hierarchy. For example, `AddressMatcher` matches a memory address operand, adjusting the expected address format for the target architecture. Then, at the instruction level, `LoadMatcher` matches loads from memory; it uses `AddressMatcher` for the memory address and thus needs only to adjust the opcode and modifiers for the target architecture. By extending this hierarchical design to instruction sequence and basic block matchers, `ReProspect` enables robust, composable matching across CUDA architectures and compiler toolchains.

The following snippet illustrates how matchers can be composed to assert the presence or absence of a 16-bit floating-point code path, *e.g.* in the SASS codes in Table 1:

```python
arch = NVIDIAArch(...)
instructions = Decoder(...)
cfg = ControlFlow.analyze(instructions)

matcher_ldg = instructions_contain(instructions_are(
    LoadGlobalMatcher(arch, size=16, extend='U', readonly=False),
    LoadGlobalMatcher(arch, size=16, extend='U', readonly=False),
))
blk, matched_ldg = BasicBlockMatcher(matcher_ldg).match(cfg=cfg)

matcher_hmnmx2 = instructions_contain(instruction_is(
    Fp16MinMaxMatcher(pmax=True))
    .with_operand(index=1, operand=f'{matched_ldg[0].operands[0]}.H0_H0')
    .with_operand(index=2, operand=f'{matched_ldg[1].operands[0]}.H0_H0')
    .with_operand(index=0, operand=RegisterMatcher(special=False))
)
matched_hmnmx2 = matcher_hmnmx2.match(blk.instructions[matcher_ldg.next_index:])
```

Table 1: Comparison of the SASS code generated for the `sm_100` architecture for the 16-bit `__half` (left) *vs* 32-bit `float` (right) maximum function.

`__hmax(const __half, const __half)`	`fmax(const float, const float)`
`LDG.E.U16 R2, desc[UR6][R2.64]`	`LDG.E.U16 R2, desc[UR6][R2.64]`
`LDG.E.U16 R5, desc[UR6][R4.64]`	`LDG.E.U16 R4, desc[UR6][R4.64]`
`...`	`...`
`HMNMX2 R5, R2.H0_H0, R5.H0_H0, !PT`	`HADD2.F32 R6, -RZ, R2.H0_H0`
`...`	`HADD2.F32 R7, -RZ, R4.H0_H0`
	`FMNMX R6, R6, R7, !PT`
	`F2FP.F16.F32.PACK_AB R3, RZ, R6`
	`...`
`STG.E.U16 desc[UR6][R6.64], R5`	`STG.E.U16 desc[UR6][R6.64], R3`

Research impact statement

`ReProspect` has been successfully used as a support for contributions to the open-source Kokkos library (Trott et al., 2022) and the development of an in-house finite element code built on top of the open-source Trilinos library (Mayr et al., 2025) (Arnst & Tomasetti, 2024) (Tomasetti & Arnst, 2024).

The `examples directory` contains several case studies inspired by these research efforts.

`Kokkos::View` allocation

CUDA API tracing provides insight into microbenchmarking results assessing the behavior of `Kokkos::View` allocation.

See online example and Kokkos issue.

Impact of `Kokkos::complex` alignment

Our in-house finite element code uses complex arithmetic for frequency-domain electromagnetism simulations. This example combines kernel profiling and SASS analysis to assess how aligning `Kokkos::complex<double>` to 8 or 16 bytes impacts memory instructions and traffic.

See online example.

Dynamic dispatch for virtual functions on device

This example uses `ReProspect` to create a research artifact that reproduces the dynamic dispatch instruction pattern identified by (Zhang et al., 2021).

See online example.

Atomics with `desul`

Kokkos provides extended atomic support through the `desul` library (Trott et al., 2022), which maps atomic operations to one of several methods with varying performance. The choice of the method depends on intricate logic, and must be tested. A micro-benchmarking approach is feasible, but requires the physical device and suffers from runtime variability. Yet, the machine code already contains information about the selected code paths. This case study demonstrates how to verify which method is chosen by matching an instruction sequence pattern.

See online example.

Code availability

`ReProspect` is available under the `LGPL-3.0` license on GitHub.

Acknowledgements

This work was supported by the Fonds de la Recherche Scientifique (F.R.S.-FNRS, Belgium) through a Research Fellowship.

AI usage disclosure

No AI was used for the design of the code. Alongside traditional tools such as `pylint` and `mypy`, `Claude` and `CoPilot` were used as assistants to improve implementation details of individual functions. AI helped improve the clarity of the manuscript.

References

Arnst, M., & Tomasetti, R. (2024). *Design patterns and performance analysis of polymorphism in multiphysics FE assembly on GPU*. https://hdl.handle.net/2268/314521.

Bhatele, A., Brink, S., & Gamblin, T. (2019). Hatchet: Pruning the overgrowth in parallel profiles. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/3295500.3356219

Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.-T., Gimenez, A., LeGendre, M., Pearce, O., & Schulz, M. (2016). Caliper: Performance introspection for HPC software stacks. *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 550–560. https://doi.org/10.1109/SC.2016.46

Brink, S., McKinsey, M., Boehme, D., Scully-Allison, C., Lumsden, I., Hawkins, D., Burgess, T., Lama, V., Lüttgau, J., Isaacs, K. E., Taufer, M., & Pearce, O. (2023). Thicket: Seeing the performance experiment forest for the individual run trees. *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 281–293. https://doi.org/10.1145/3588195.3592989

Hayes, A. B., Hua, F., Huang, J., Chen, Y., & Zhang, E. Z. (2019). *Decoding CUDA binary - file format*. Zenodo. https://doi.org/10.5281/zenodo.2339027

Knüpfer, A., Rössel, C., Mey, D. an, Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., & Wolf, F. (2012). Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, TAU, and vampir. In H. Brunst, M. S. Müller, W. E. Nagel, & M. M. Resch (Eds.), *Tools for high performance computing 2011* (pp. 79–91). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-31476-6_7

Mayr, M., Heinlein, A., Glusa, C., Rajamanickam, S., Arnst, M., Bartlett, R., Berger-Vergiat, L., Boman, E., Devine, K., Harper, G., Heroux, M., Hoemmen, M., Hu, J., Kelley, B., Kim, K., Kouri, D. P., Kuberry, P., Liegeois, K., Ober, C. C., … Yamazaki, I. (2025). *Trilinos: Enabling scientific computing across diverse hardware architectures at scale*. https://doi.org/10.48550/arXiv.2503.08126

NVIDIA Corporation. (2025a). *CUDA binary utilities*. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

NVIDIA Corporation. (2025b). *NVIDIA CUDA profiling tools interface (CUPTI) - CUDA toolkit*. https://developer.nvidia.com/cupti.

NVIDIA Corporation. (2025c). *NVIDIA nsight compute*. https://developer.nvidia.com/nsight-compute.

NVIDIA Corporation. (2025d). *NVIDIA nsight systems*. https://developer.nvidia.com/nsight-systems.

NVIDIA Corporation. (2025e). *NVTX: NVIDIA Tools Extension Library*. https://github.com/NVIDIA/NVTX.

NVIDIA Corporation. (2025f). *Parallel thread execution ISA version 9.1*. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

Terpstra, D., Jagode, H., You, H., & Dongarra, J. (2010). Collecting performance data with PAPI-c. In M. S. Müller, M. M. Resch, A. Schulz, & W. E. Nagel (Eds.), *Tools for high performance computing 2009* (pp. 157–173). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-11261-4_11

Tomasetti, R., & Arnst, M. (2024). *Efficiently implementing FE boundary conditions using stream-orchestrated execution on GPU*. https://hdl.handle.net/2268/314520.

Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., & Wilke, J. (2022). Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, *33*(4), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

Zhang, M., Alawneh, A., & Rogers, T. G. (2021). Judging a type by its pointer: Optimizing GPU virtual functions. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3445814.3446734

Zhou, K., Adhianto, L., Anderson, J., Cherian, A., Grubisic, D., Krentel, M., Liu, Y., Meng, X., & Mellor-Crummey, J. (2021). Measurement and analysis of GPU-accelerated applications with HPCToolkit. *Parallel Computing*, *108*, 102837. https://doi.org/10.1016/j.parco.2021.102837