Creating teaching materials for C++ and CUDA with xeus-cpp

Hristiyan Shterev

High School of Mathematics “Akademik Kiril Popov” Plovdiv

Mentor: Vassil Vassilev

About Me

School Email: hristiyanshterev_20a@schoolmath.eu

Personal Email: hristiyansterev@gmail.com

GitHub: Hris Shterev

Location: Bulgaria

I am Hristiyan Shterev and I`m a high-school student with a strong interest in C++ and systems programming. I have around four years of experience working with C++, during which I have built a solid foundation in the language. Through my internship at Compiler Research, I aim to expand my understanding of interactive execution environments and Jupyter kernels by working with the xeus-cpp project.

Project Overview

The xeus-cpp project provides a Jupyter kernel that enables interactive C++ programming in Jupyter notebooks. It allows C++ code to be compiled and executed cell by cell while preserving state across executions, which makes it useful for experimentation and learning, but also introduces behavior that is not always obvious.

This project starts with contained, simpler tasks that assume prior C++ experience, rather than beginner-level programming. The initial phase focuses on writing structured C++ programs in xeus-cpp notebooks and observing how they behave when split across multiple cells. These examples are used to explore and explain key aspects of the kernel, such as execution order, state persistence, recompilation, and error handling.

As the project progresses, the focus shifts toward more advanced usage patterns, including modifying existing code, reusing definitions across cells, and understanding how changes affect previously executed code. The goal is to build a clear mental model of how xeus-cpp works from a user's perspective.

With this foundation, the project then moves to a high-level exploration of the xeus-cpp codebase to understand its overall structure and execution flow. Based on this understanding, the final stage focuses on small, practical contributions such as improving documentation, adding example notebooks, or clarifying existing behavior.

Project goals

The main goal of this project is to understand how xeus-cpp and CUDA work and how to use them together to contribute by creating teaching materials with the Jupiter notebook. More specific goals include:

- Understanding xeus-cpp and its state management and differences from normal C++ compilations
- Learn how the notebook remembers variables and functions from one cell to the next
- Porting different courses to xeus-cpp
    - C/C++: Tutorial: Learning resources C/C++
    - OpenMP Tutorial: An Introduction to Parallel Programming with OpenMP
    - CUDA Tutorial: CUDA by example
- Trigger and document most possible linking and runtime errors
- Explore the source code of xeus-cpp and learn about how it works. Contribute small and safe improvements
- Finish with a documented report of everything done in this project

- Create visual guides to show how data moves between the CPU and the GPU

Weeks	Activities	Deliverables	Milestones
Week 1	Confirm the xeus-cpp setup and run small and side-by-side examples. Compare with nvcc for performance gaps	Report performance gaps with working CPU & GPU example like vector addition and scalar multiplication	Confirm functional xeus-cpp + CUDA environment; execute test for performance
Week 2	Study how cells work and transfer variables. Track the memory addresses of a vector and check if GPU pointers stay valid if you run cells out of order or multiple times.	Documentation of state management across cells, observations on execution order and recompilation, vector addition and scalar multiplication	Understand xeus-cpp state management and differences from traditional compilation. A clear map of how long data stays alive during a live session
Week 3	Test repeated redefinition of functions, classes, and CUDA kernels. Examples include class definition within inline/out of line methods	Report on redefinition behaviors and edge cases, example notebooks showing findings	List of specific code patterns that users must avoid to prevent kernel panics
Week 4	Porting C++: Adapt topics on C++11 through C++20. Implement smart pointers, STL vectors, and classes to show state persistence.	Modern C++ Lab notebooks showing the shift from C-style arrays to safe STL containers.	Successful use of Modern C++ features in xeus-cpp
Week 5	Porting C course: Adapt tutorials on memory management and pointers. Focus on manual allocation vs stack allocation	Interactive pointers and memory management lab notebooks	Demonstrate low level memory control in a JIT environment

Week 6	Trigger and categorize compilation, linking, and runtime errors like out of bounds device memory access	A detailed report on how the system handles CUDA crashes and the quality of the error logs.	Determine if xeus-cpp is strong enough for messy, real-world debugging. Understand error propagation and diagnostics quality
Week 7	Buffer week: fix bugs, refine tasks	Updated and cleaned notebooks, fixed issues	Ensure all prior work is stable and well-documented
Week 8	Design and test interactive workflows with multiple units & shared state sorting library split across cells and test if they work	A template project showing the best way to organize a large CUDA project within a notebook	Verified workflow for moving beyond simple one-cell scripts.
Week 9	Porting “CUDA by example”: Port Chapter 3 (Julia Set) for visualization and Chapter 5 (Thread Cooperation) for dot product calculations.	CUDA Entry Notebooks featuring interactive Julia Set generation and vector reduction.	Verify GPU kernel execution within the interactive environment.
Week 10	Use NVIDIA profiling tools like nsys and nvprof to see exactly where time is being spent. Measure if the JIT process or the kernel launch itself adds any significant lag.	Data showing the "performance tax" of using an interactive notebook instead of a compiled program.	A clear understanding of the speed trade-offs for interactive development.
Week 11 - 12	Explore xeus-cpp source code and follow the path a piece of code takes from the moment you hit “Shift + Enter” in the browser until it reaches the LLVM JIT engine.	An architectural diagram showing the internal data flow from the cell to the GPU	Full understanding of the internal mechanics of the interpreter.

Week 13	Buffer week: fix bugs, refine tasks, additional testing	Cleaned and validated notebooks, bug fixes	Ensure readiness for codebase exploration and contributions
Week 14 - 15	Contribute to the project. This involves fixing bugs,, updating the documentation or adding CUDA examples to the main repository	Official pull requests submitted to the xeus-cpp GitHub page	Successful contribution of tangible code to the open-source community
Week 16	Porting OpenMP: Adapt Numerical Integration and Loop Worksharing from CERN materials. Implement thread-count scaling and timing tests.	OpenMP Lab notebooks demonstrating parallel regions and race condition diagnostics.	Multi-threaded execution without external build flags.
Week 18	WASM Porting Attempt: Try to run the C/C++ and OpenMP tutorials in a browser using WASM.	A document explaining which parts of your C, C++, and OpenMP tutorials actually worked in the browser and which parts "broke."	"See if the parallel tutorials we built can run in a web browser so that users don't have to install CUDA or compilers on their own machines."
Week 17	Wrap up the project by re-running all benchmarks. Combine all the testing data and source analysis into a final report.	A comprehensive final report summarizing the limits and strengths of interactive CUDA.	Complete project with documented results and contributions; summarize findings and future directions