PhD studentship: Automated Testing and Verification of Machine Learning Compilers and Runtimes

A fully-funded (**home fees**) studentship is available for a student interested in studying techniques to improve the reliability of machine learning compilers and runtime systems, supervised by Professor Alastair Donaldson, leader of the FastPL group in the Department of Computing at Imperial College London.

Interested candidates should contact Alastair Donaldson (alastair.donaldson@imperial.ac.uk) before 30 April 2025.

Note: the studentship only funds tuition fees at the "home" rate. Please see Imperial's Fee Status page to check whether you qualify for this.

Background

Machine learning compilers and runtime systems are central to the training of ML models: training large models such as Google's Gemini and OpenAI's GPT family requires code that must run reliably for months across millions of cores, distributed across tens of thousands of nodes. An ML compiler (e.g. XLA) generates this code, and is fundamentally different from a traditional compiler. A traditional compiler lowers a source code module to machine code or bytecode. In contrast, an ML compiler takes a high-level model description (e.g. in TensorFlow or PyTorch) and outputs a highly complex concurrent and distributed application that runs on a heterogenous range of devices and interacts with specialised runtime libraries. To achieve performance, ML compilers perform critical optimisations that take advantage of modern CPU hardware and massively-parallel accelerators (TPUs and GPUs); they hide latency through double-buffered DMA transfers that overlap computation and communication; and they orchestrate a delicate dance of managing synchronisation between concurrent components, aiming to incur minimal synchronisation overhead whilst ensuring freedom from data races and memory consistency issues.

Defects in ML compilers can fundamentally undermine the training of large models. Concurrency-related defects, such as data races, are particularly challenging to detect, reproduce and debug. Program analysis and testing techniques to avoids such problems before ML models are put into production are thus critically needed.

Research Opportunity

This PhD project offers a chance to investigate the combination of static and dynamic program analysis techniques to automatically analyse:

- The code generation algorithms used by ML compilers;
- The implementation of the runtime libraries on which ML compilers depend.

The aim of these analysis will be to identify bugs, with a special focus on *concurrency errors* such as data races and deadlocks, and on the synthesis of examples that allow such errors to be reliably reproduced so that they can be fixed.

Potential Research Directions

This is an open-ended project that can be adapted to suit the needs of an excellent student, but some initial research ideas include:

Extracting coordination and data movement skeletons. ML compiler output comprises code in a range of languages and can be broadly partitioned into (a) *coordination and data movement code*, which controls concurrency and orchestrates the flow of data through the training pipeline, and (b) *computation code*, comprising the intense computational "number crunching" kernels at the heart of the training process. A promising idea is therefore to extracting a *coordination and data movement skeleton*: a faithful abstraction of the coordination and data movement code. A skeleton will abstractly represent the compiler-generated code that tracks the way concurrent components will be managed, and the data that they access, and will capture sufficient detail to enable subsequent analysis of e.g.: whether sufficient synchronisation is place to avoid corruption between concurrent DMA transfers; and whether computational kernels feature sufficient barrier synchronization operations to eliminate data races. An important challenge will be how to deal with data-dependent control flow, where computation code influences data movement.

Constraint-based detection of concurrency issues. Having extracted a coordination and data movement skeleton (or a similar representation), there will be scope for developing various analysis techniques for assessing whether it is possible for concurrency bugs to arise. Inspired by previous successful work on the analysis of GPU programs, a promising idea is to follow a constraint-based approach. A skeleton will be analysed to determine the conditions under which concurrent interactions are unsafe (i.e., lead to races), and expressed as a set of satisfiability modulo theories (SMT) constraints. These constraints will then be discharged by an SMT solver, so that solutions to the constraints map to concurrency errors. From such solutions, it will be possible to mine information that can be mapped back to the compiler-generated code to help with debugging.

Reproducing concurrency bugs via controlled scheduling. The concurrency issues reported by the above analysis techniques will provide *evidence* of a concurrency error in compiler-generated code but will not allow direct reproduction of the error. Another strand of work could focus on automatic generation of reproducers. Based on the evidence arising from program analysis, compiler-generated code will be instrumented in a manner that will attempt to *force* the identified concurrency issue. This could involve deterministic forcing by controlling CPU thread schedules (building on prior work on record and replay with controlled scheduling). When such control cannot be exerted (e.g. for problems that depend on the interleaving of threads on a custom accelerator), there is room to investigate dynamic techniques for teasing out concurrency bugs via "hostile environments", which create the adverse conditions required to trigger unusual thread schedules, and which have been successful in the PI's prior work on triggering weak memory errors in GPU kernels.

Fuzz testing of ML compiler infrastructure. As a complement or alternative to the above analysis-based ideas, there is broad scope for applying generative fuzzing to test the reliability of ML compilers and the runtime components on which they depend. This could include intensive testing of low-level infrastructure, such as the NCCL collective communications library used for multi-GPU training across NVIDIA GPUs, or fuzzing of other key libraries of GPU kernel primitives.