THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

xDSL: A common compiler ecosystem for domain specific languages

Citation for published version:
Brown, N, Grosser, T, Fehr, M, Steuwer, M & Kelly, P 2022, 'xDSL: A common compiler ecosystem for domain specific languages', Supercomputing 2023, Denver, United States, 12/11/23 - 17/11/23.

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 15. Mar. 2025

xDSL: A common compiler ecosystem for domain specific languages

Nick Brown
EPCC
University of Edinburgh
Edinburgh, UK
n.brown@epcc.ed.ac.uk

Tobias Grosser
School of Informatics
University of Edinburgh
Edinburgh, UK

Mathieu Fehr
School of Informatics
University of Edinburgh
Edinburgh, UK

Michel Steuwer
School of Informatics
University of Edinburgh
Edinburgh, UK

Paul Kelly
Department of Computing
Imperial College London
London, UK

I. The challenge

Developing high performance simulation codes that can exploit current generation supercomputers is extremely difficult, and this problem is only going to get worse as we move further into the exascale era. Architectures are going to become more diverse and the scale much larger, resulting in the direct programming of such machines needing to incorporate decisions around parallelism at multiple levels which will require significant expertise in many different HPC technologies. Domain scientists and engineers neither have the knowledge or interest in manually dictating such low level, tricky details in their code and, put simply, they would much rather be doing their science rather than having to concern themselves with intricate details of parallelism.

Many in the HPC community agree that Domain Specific Languages (DSLs) are one way in which this challenge can be solved and have an important potential role in future HPC software development. While the term language can be off-putting, in the past few years DSLs have been commonly delivered as domain-specific abstractions, making them far more attractive. Typically embedded within existing languages, such as Python, C++, or Fortran, the programmer exploits domain-specific abstractions to write problem specifications driven by their science rather than detailing exactly how this is to be implemented at the lowest levels.

Abstracting the scientific programmer from these tricky, low-level details of parallelism and locality not only significantly improves programmer productivity but has also been repeatedly demonstrated to increase performance. This is because, by leveraging the rich domain knowledge encoded by the programmer, the compilation toolchain can make effective parallelism choices, such as where to schedule components and how these should communicate. Furthermore, if the DSL is designed correctly, then the programmer is able to express their workload in an architecture independent fashion, thus making portability across architectures accessible, with the compilation toolchain doing much of the heavy lifting.

A. Siloed toolchains: A major drawback of DSLs

Given the benefits of DSLs, a question is why they are not ubiquitous and why do programmers still use technologies such as MPI, CUDA, and OpenMP directly? One significant blocker is the siloing of underlying compilation toolchains. This is where DSLs are frequently developed in isolation, requiring significant investments of effort on behalf of DSL developers because there is little or no intermediate abstractions or infrastructure shared between projects. This is bad for DSL developers as it can require significant time and resources to bring new abstractions to production, resulting in a relatively large software stack to be maintained and extended which can be especially challenging when considering portability to new architectures. Moreover, the provision of additional tools such as debuggers and profilers can often require considerable integration work if even possible.

For scientific software developers considering whether to use a DSL or not, this siloing is very disadvantageous as it represents risk. Whilst the DSL's abstractions might be appropriate, there can be serious concerns over long term maintenance and whether the DSL will target future generations of supercomputer architectures. This can be especially concerning if a DSL has been developed by a small group and grown organically, without the guarantee of long term funding.

II. Our solution: A common ecosystem built on LLVM and MLIR

Recent advances, e.g. Multi-Level Intermediate Representation (MLIR) [1] and LLVM [2], provide potential technological solutions to this problem of DSL toolchain siloing. LLVM is a collection of modular compiler and toolchain technologies, and LLVM backends are provided by the community, and vendors, for a wide variety of hardware. Each of these accepts LLVM's Intermediate Representation (LLVM-IR) of a user's code and transforms it to a form that will execute on target hardware. Consequently, a compiler developer can write a front-end for a language which generates LLVM-IR, and this will then be able to execute across a wide variety of hardware by selecting appropriate LLVM backend(s).

A challenge is that LLVM-IR is low level, requiring a significant number of transformations between the high level representation of user's code and LLVM-IR. To address this MLIR, a technology for intermediate representations, was developed by Google. In addition to providing a series of IR dialects and transformations, MLIR is also a framework where developers can add their own. By targeting the transformation

of user code into higher level dialects, these can be mixed and manipulated at different levels of abstractions, enabling progressively lowering the abstraction level to LLVM-IR. Much of this lowering it undertaken by existing dialects and transformations, thus significantly reducing the overall software effort needed by the compiler developer.

There are two further benefits of LLVM and MLIR, firstly because these are supported by large communities and backed by some of the worlds largest technology companies, the future of such technologies is secure. Secondly, a wealth of third party tooling, including debuggers and profilers, exists.

However, in their current form these technologies are complex to use and require a significant learning curve. To address this and lower the barrier to entry we have developed a Python based toolbox for MLIR, known as xDSL, where dialects and utilities can be conveniently expressing in Python and seamlessly converted to and from the existing MLIR C++ framework. Figure 1 provides an illustration of our xDSL ecosystem, where the Python toolbox is illustrated in blue and building on-top of LLVM and MLIR in grey. The core component of our toolbox is IRDL [4], which a Python-based domain specific language for expressing and manipulating MLIR dialects, and on-top of this we provide utilities such as enabling transformations between dialects and parsing. There are several simple, foundational dialects provided by MLIR and their Python counterparts are also provided by xDSL using IRDL to express them. There is a considerable amount of mechanism provided too so that developers can easily specify their own dialects and transformations in Python with xDSL handling associated complexities.

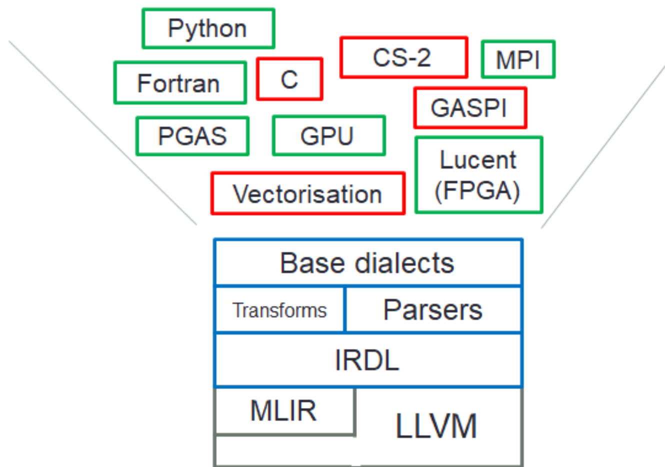

Fig. 1. Illustration of xDSL technology stack. All aspects, apart from MLIR and LLVM are provided by xDSL but with the dialects mirroring their MLIR counterparts where available.

At the top of figure 1 it can be seen that there are various dialects (and associated transformations) represented by green and red boxes. Those in green are already provided, in red are those which are planned for development. Crucially we do not dictate how the stack should be used, instead DSL developers can pick and choose which dialects and transformations are appropriate for them and mix these as required. By doing so they are able to take advantage of the existing transformations and optimisations that have been provided and ultimately result in executable code targeting a wide variety of platforms. Furthermore, if they need to, they can define their own dialects and transformations in Python using IRDL and our framework.

Consequently, our xDSL ecosystem is a combination of the Python based toolbox enabling convenient expression of dialects, transformations, and easy integration with MLIR, as well as numerous HPC focused dialects. DSLs then become a thin abstraction layer on-top of this existing infrastructure.

A. *Case study: Integrating xDSL with a Fortran DSL*

PSyclone [3] is a Fortran-based DSL developed by STFC and in use by the Met Office for several of their weather and climate models. Providing a separation of concerns between the scientific application and mechanisms of parallelism, PSyclone is currently provided by an entirely bespoke compiler infrastructure resulting in the disadvantages highlighted earlier.

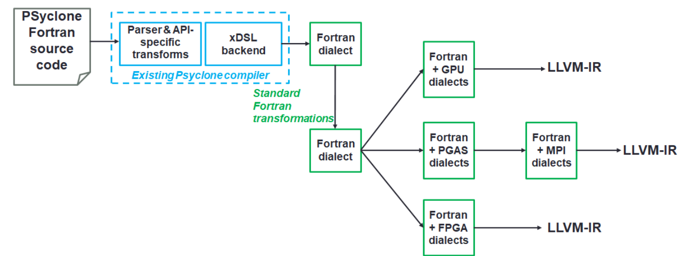

Fig. 2. Illustration of PSyclone xDSL integration. The boxes and text in green are the contribution of xDSL, with a backend developed to generate xDSL Fortran dialect.

Figure 2 illustrates the integration of xDSL with PSyclone, where a user's Fortran source code is first parsed by the existing PSyclone tooling before being passed to our xDSL PSyclone backend. This generates a representation of the user's code in our Fortran xDSL/MLIR dialect, before a series of existing Fortran and optional PSyclone specific transformations can be applied. Figure 2 provides an example of targeting GPUs via transforms which incorporate the GPU dialect, targeting distributed memory by first mixing the PGAS dialect and then converting this into the MPI dialect, or FPGAs via the Lucent (FPGA) dialect. Regardless, the dialects are then transformed into the appropriate LLVM-IR form which will be used as input to the appropriate hardware backend.

Crucially, from the perspective of PSyclone, we are reusing the existing Fortran, GPU, PGAS, MPI, and Lucent dialects as well as transformations, optimisations, and conversion into LLVM-IR. Consequently PSyclone integration has required a new backend in the existing PSyclone toolchain to generate the initial xDSL Fortran IR, and optional PSyclone xDSL transformations which can be plugged in.

III. Community Involvement

xDSL is open source and we welcome contributions and users, see https://www.xdsl.dev and https://github.com/xdslproject for details.

References

[1] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N. and Zinenko, O., 2020. MLIR: A compiler infrastructure for the end of Moore's law. arXiv preprint arXiv:2002.11054.

[2] Lattner, C. and Adve, V., 2004, March. LLVM: A compilation framework for lifelong program analysis & transformation. In International Symposium on Code Generation and Optimization, 2004. CGO 2004. (pp. 75-86). IEEE.

[3] Adams, S.V., Ford, R.W., Hambley, M., Hobson, J.M., Kavčič, I., Maynard, C.M., Melvin, T., Müller, E.H., Mullerworth, S., Porter, A.R. and Rezny, M., 2019. LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models. Journal of Parallel and Distributed Computing, 132, pp.383-396.

[4] Fehr, M., Niu, J., Riddle, R., Amini, M., Su, Z. and Grosser, T., 2022, June. IRDL: an IR definition language for SSA compilers. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (pp. 199-212).