Memory-safe single-source heterogeneous programming support for Rust

David Wood (2198230W)

February 17, 2020

Abstract

In the last two decades, single-core processor performance has plateaued and both industry and academia have turned to heterogeneous systems, those composed of varied processors and architectures, to push the performance envelope. Graphics processing units (GPUs), traditionally designed for image rendering, are regularly being used for parallel computation; and a next generation of processors, with thousands of cores and terabytes of memory bandwidth, are being designed to tackle machine learning, computer vision and artificial intelligence problems. Heterogeneous systems are being deployed widely and in safety-critical contexts, such as self-driving cars. However, heterogeneous programming models are still derived from C and C++, languages with fundamentally unsafe memory models - the leading source of security vulnerabilities. We propose a new heterogeneous programming model, leveraging the advancements in programming language design and unique region-based memory management found in Rust. This system will improve programmer productivity, correctness and performance in heterogeneous computing while lowering the barrier-to-entry to innovate by decoupling compiler and runtime concerns.

1 Background

As the end of Moore's Law approaches, heterogeneous systems - those composed of multiple kinds of processor or core are becoming increasingly necessary to achieve higher performance.

Moore's Law predicted that transistor count in processors would double about every two years. This prediction has proven accurate due to repeated improvements in transistor manufacturing - processor companies have been able to pack more transistors onto the same size of chip each generation. However, recent seven and ten nanometer manufacturing processes result in transistors that are a mere 40 atoms across, indicating a trend which cannot continue.

Decreased transistor size results in better performing cores as clock speed can be increased without impacting power consumption. The relationship between frequency (clock speed), voltage (power consumption) and capacitance (transistor size) is known as Dennard's Scaling.

Unfortunately, a fourth factor - leakage - resulted in the breakdown of Dennard's Scaling in the last decade. As single-core performance plateaued, processor companies turned to parallelism to continue to push the performance envelope and thus the increase in multi-core processors in the last decade.

By combining different kinds of processors and cores, heterogeneous systems are now able to achieve far higher performance than traditional CPUs. For example, GPUs are constructed for massive parallelism, and are thus far more suited to parallel processing tasks than CPUs. As a result, both academia and industry have turned to GPUs to accelerate their workloads, which are far removed from the traditional graphics tasks that the chips were designed for - a trend known as general-purpose graphics processing (GPGPU).

In addition, a new generation of startup companies are building processors specifically for artificial intelligence and computer vision. These processors, characterised by massive core counts and exceptional memory bandwidth, are also being integrated into heterogeneous systems.

Due to the varied hardware in heterogeneous systems, they traditionally require target-specific approaches to development. Different programming languages or libraries need to be used for each kind of hardware. This is in contrast to single-source heterogeneous programming, where code is written once and compiled to different targets for different hardware.

While some programming models allow programs to be written which execute across heterogeneous platforms, these platforms are typically low-level and require lots of boilerplate code to get started - which impacts programmer productivity, correctness and performance.

Furthermore, heterogeneous systems programming is dominated by languages deriving from C and C++. Therefore, heterogeneous systems programming is faced by the same memory safety, security, concurrency and correctness challenges as systems software written in C and C++. By its very nature, heterogeneous systems programming often deals with greater concurrency than traditional systems software.

As heterogeneous systems are being deployed in an ever-greater variety of systems, ensuring that these systems are correct and free of vulnerabilities is paramount. For example, specialised processors for artificial intelligence and machine learning are being evaluated and integrated by automotive companies for use in safety critical contexts, such as self-driving cars.

2 Solution and Key Ideas

2.1 Memory Safety

Rust is a modern systems programming language with a focus on safety and efficiency. Rust leverages advances in programming language design and a novel approach to memory management - region-based memory management - to create a systems programming language which guarantees memory safety at compile-time while producing efficient code.

Existing heterogeneous programming models require code is written in languages derived from C and C++, which have fundamentally unsafe memory models. A key idea of this research is to leverage the unique features of the Rust programming language to provide a safe foundation for heterogeneous programming.

Any heterogeneous programming model implemented in Rust would benefit from the language's unique ownership and borrowing systems which provide compile-time guarantees of memory safety - eliminating a significant source of security vulnerabilities.

2.2 Decoupled Compiler and Runtime

Existing heterogeneous programming models often require substantial modifications to the language compiler. Regularly, these modifications are heavily tied to the semantics of the heterogeneous programming model. As a result of the inter-dependencies between the compiler and runtime in existing heterogeneous programming models, experimentation to advance the state-of-the-art has a high barrier to entry.

By leveraging Rust's macro system, these inter-dependencies can be substantially reduced. Rust implements a hygienic macro system which allows for safe code generation, unlike C macros or C++ templates. Procedural macros enable library authors to define attributes that can be applied to functions and types to generate code. These are functions which take the token stream representing the code written by the user and return a token stream which replaces or is compiled in addition to the user's code.

Through use of macros, a heterogeneous programming model would be able to implement validation that the user code being compiled for devices does not contain any illegal constructs without implementing these checks in the language's compiler. In addition, structures used to contain state for the runtime components of the heterogeneous platform could be generated for each function being compiled for devices.

In addition, Rust's unique ownership and borrowing system introduces the concept of lifetimes to track the duration of a program's execution where objects are allocated. It is common for processors used in heterogeneous systems to have multiple address spaces. Pointers in different address spaces refer to different regions of memory. Effective utilisation of address spaces is critical to achieving high performance.

While some heterogeneous programming models require that address spaces are stated explicitly, many perform address space inference in the compiler (or runtime, which isn't supported on all hardware). Address space inference can be considered isomorphic to Rust's lifetime inference and thus existing language and compiler echanisms can be leveraged.

2.3 Performance

Rust's ownership and borrowing system provide guarantees about aliasing which can be leveraged to improve the performance of the heterogeneous programming model.

Aliasing is when multiple pointers refer to the same memory. In Rust, pointers (known as references) can either be shared or mutable. Programmers are limited to having one mutable reference to data *or* one or more shared references, but not both. This limitation is integral to providing Rust's memory safety guarantees but also presents opportunities for optimisation.

Intraprocedural optimisations can assume that if data is mutated through a pointer that other pointers do not refer to the same data. In traditional systems software, this can allow some memory accesses to be cached and redundant computations to be avoided. In heterogeneous systems, this information can be leveraged to avoid unnecessary data transfers between the host and device.

3 Objectives

- Improve the productivity of programmers working with heterogeneous systems by implementing a heterogeneous programming model in a modern systems programming language that utilises advances in programming language design.

- Reduce the barrier-to-entry in experimentation and research in heterogeneous programming models by decoupling compiler modifications from runtime implementations.

- Fortify the foundations of heterogeneous programming by addressing memory unsafety that results from use of programming languages, such as C and C++, with fundamentally unsafe memory models.

- Improve performance of heterogeneous systems by enabling new optimisations that leverage the aliasing guarantees provided by the Rust language.

4 State of the Art

4.1 OpenMP

OpenMP is a library for shared-memory multiprocessing, consisting of compiler directives, library routines and environment variables. OpenMP supports C, C++ and Fortran. In 2018, OpenMP 5.0 built on limited GPGPU support specified in earlier OpenMP 4.0 and 4.5 releases from 2013 and 2015 respectively.

Before OpenMP 4.0 implemented initial support, Lee et al. [7] implemented a compiler framework for transforming OpenMP applications to CUDA. Similarly, Lee et al. [6] implemented OpenMPC, which builds on OpenMP with an abstraction over CUDA's interface.

Additional research has been conducted in this area to compare the performance and productivity implications of using OpenMP instead of the other heterogeneous frameworks presented here [8] and on further performance optimisations that could be implemented [12].

4.2 OpenACC

OpenACC is a programming standard for parallel computing. Like OpenMP, compiler directives are used to identify code is to be parallelised. As a result, OpenACC requires compiler support.

For a long time, there weren't many compilers which implemented the OpenACC standard, and thus there have been efforts to allow translation of OpenACC to OpenMP 4.0 [14] and OpenCL C [15]. Nowadays, GCC 9.1 offers almost complete OpenACC 2.5 support. Li et al. evaluated the performance of OpenACC in the PGI compiler with CUDA and found that while generated code was slightly slower, OpenACC had fewer `memcpy` calls and less data transfer time.

4.3 OpenCL

OpenCL is a framework for writing parallel programs which execute across a diverse array of processors. OpenCL allows the programmer to write "compute kernels" in OpenCL C, a subset of C99 modified to fit OpenCL's device model. As such, OpenCL is not a single-source programming model and requires modifications to the compiler (to support OpenCL C's semantics and syntax).

Unlike CUDA, OpenCL is an open standard maintained by the Khronos Group, an industry consortium. As a result, OpenCL supports a wider variety of target platforms than other heterogeneous platforms and exposes a lower-level interface.

Due to OpenCL's low-level nature, Memeti et al. [8] found that programming with OpenCL required significantly more effort than programming OpenACC for the SPEC ACCEL benchmark suite [4] (which is designed to test accelerator performance); and required approximately double the effort of CUDA for the Rodinia benchmark suite. In addition, Su et al. [13] compared the performance of OpenCL with CUDA and found that CUDA was 3.8% to 5.4% faster, which is unsurprising as OpenCL as-a-whole cannot be tuned to perform well for specific devices.

4.4 CUDA

CUDA is a parallel computing platform developed by NVIDIA exclusively for their own GPU hardware. Like OpenCL, CUDA is not a single-source programming model and requires compiler support - programs are written in CUDA C/C++ and are compiled using NVIDIA's own compiler or using Clang's CUDA support.

CUDA is also fairly low-level, and as such, efforts have been made [2] to implement higher-level programming models which use OpenCL and CUDA. Moreover, efforts to translate between OpenCL and CUDA while maintaining performance have been made [5][11][9].

4.5 SYCL

SYCL is a high-level abstraction layer built atop OpenCL. SYCL enables a single-source style where device functions are written in a subset of standard C++ within regular code. At runtime, SYCL constructs an asynchronous task graph from the user's code and schedules kernels using the SYCL backend (typically OpenCL) to perform heterogeneous computation.

While SYCL does requires compiler modifications, some implementations only use a modified compiler for the device compilation and work with a host compiler. Compiler modifications are required in order to support offloading to different intermediate languages consumed by device drivers; and for address space inference.

Initial versions of SYCL targeted OpenCL with SPIR (an intermediate language based on LLVM IR consumed by device drivers). Newer versions of the SYCL specification support OpenCL with SPIR-V or NVPTX (CUDA's intermediate language) and even CUDA.

Like OpenCL, SYCL is an open standard maintained by the Khronos Group. SYCL is being used on the upcoming Aurora supercomputer at Argonne National Labs. SYCL is among the closest comparisons to the programming model proposed in this document. Some research has found that SYCL can outperform OpenCL [3] but other research has found that, like OpenCL, portability does not mean performance portability [10] (when compared to performance of other heterogeneous programming models).

4.6 DCompute

DCompute is a set of libraries to enable native execution of the D programming language on GPUs and other heterogeneous programming languages. DCompute compiles D to SPIR-V [16] for execution with OpenCL.

DCompute is also similar to the project proposed in this document, aiming to address the shortcomings that result from C and C++ in existing heterogeneous programming models, such as OpenCL and SYCL. DCompute attempts to leverage D's meta-programming facilities and type-system to improve the state of heterogeneous computing. However, D does not provide the same memory safety guarantees as those provided by Rust and that would be leveraged by this research.

5 Methodology

5.1 Review existing approaches to offloading (WP1)

Offloading is the feature that would enable certain functions to be compiled to an intermediate representation for execution on accelerators.

Research would be conducted into the approaches taken by other languages to implement offloading and the long-term impact of their decisions. This research would prove valuable in offering alternatives and potential drawbacks.

In addition, any prior RFCs to the Rust project which relate to offloading or similar functionality would need to be reviewed. Lessons gleaned from this would improve the design of the feature and improve the chances that it is accepted when proposed to the Rust project.

Deliverable: Report describing the approaches taken by other heterogeneous programming models to implement offloading, with a focus on how these approaches are exposed to end-users.

5.2 Write RFC for offloading in Rust (WP2)

In order to propose a new language feature for Rust, an RFC would be drafted. RFCs require a extensive motivation section which justifies the need for a feature, a detailed design, consideration of how the feature would be taught to new users, any drawbacks or alternatives and any unresolved questions.

In addition to the research performed in WP1, experimentation could proceed in libraries outside of the core language (using the macro system). While it would not be feasible to make these approaches fully functional, gaining hands-on experience with different approaches would be valuable and could make the RFC more compelling (this has helped with other RFCs in the past).

Any RFC for this feature would need to discuss the interface exposed to users to request offloading of a given function and the mechanism with which the embedded device binary code could be retrieved.

Deliverable: An RFC describing the motivation, detailed design, drawbacks, alternatives and unresolved questions regarding a language feature for "offloading".

5.3 Modify build system to support offload targets (WP3)

Before proceeding with compiler modification, the build system would need to be modified so that a build of LLVM which can generate code for SPIR-V or NVPTX is linked with the compiler.

If support for SPIR-V is prioritised over NVPTX, then integration of the Khronos Group's SPIR-V to LLVM translator into the build system would also be required, as LLVM does not have a SPIR-V backend by default. Alternatively, a recently announced SPIRV backend for LLVM by ARM could be used - although this backend has less use in production systems, so could result in instability.

Deliverable: Nightly builds of the Rust compiler are being produced which link against the new LLVM toolchain that has support for the required offloading targets.

5.4 Add support for offloading to compiler driver (WP4)

Modifications to the "compiler driver" would then be required. Rust's compiler driver orchestrates the phases of the compiler and would be modified so that code generation phases can be repeated for different targets (both host and device).

In addition, Rust's code generation infrastructure would need to be modified to support only generating code for a subset of the current project being compiled (only the call graph of functions annotated as being offloaded).

Offload bundling infrastructure would need to be added to the compiler driver to inject a compiled device code binary into the host binary. This mechanism would need to conform to the accepted RFC.

Extensions to the compiler test infrastructure would need to be implemented to validate that device code is being correctly generated and injected.

Deliverable: Tests have been added and pass which demonstrate that valid device code is being injected into the binary for simple functions.

5.5 Implement feature for offloading (WP5)

Building on WP1, WP2, WP3 and WP4, implementation of the accepted offloading feature could then begin in the compiler and would be tested behind a feature flag in nightly builds before being stabilised.

Depending on the outcome of WP1, addition of a compiler flag or language attribute would then occur, enabling end-users to request that functions are offloaded and available in the device binary.

Deliverable: Functioning implementation of the accepted RFC from WP1 with associated tests demonstrating correct behaviour.

5.6 Investigate existing approaches to address spaces (WP6)

It would first be necessary to determine the scope of the address space support that will be implemented by the project.

For example, in Intel's SYCL implementation, only the "generic" address space is supported. Generic address spaces are a feature found in OpenCL 2.0 and allow their compiler to avoid implementing address space inference, instead every pointer lives in the generic address space and inference happens in the OpenCL runtime. However, this approach requires hardware support that isn't always available, leaving some processors unsupported.

Assuming that non-generic address spaces are supported, implementations of address spaces in other heterogeneous programming models would be reviewed. In most languages, some form of address space inference is implemented, as requiring the user to annotate all pointers with the appropriate address space would hamper code re-use and reduce ergonomics.

Deliverable: A report detailing the approach taken by other heterogeneous programming models to support multiple address spaces and the advantages/disadvantages of each.

5.7 Implementation and design of address spaces in Rust (WP7)

Address space support, like offloading, will require language design to add to the Rust language.

As address space inference could be considered isomorphic to Rust's existing lifetime inference implemented in the borrow checker, a detailed comparison of the two systems should be conducted to identify any places where there are incompatibilities or where extensions to existing infrastructure are required.

Furthermore, in this work item, address space support will be described in an RFC and implemented into the compiler.

Deliverable: An RFC describing the modifications required to Rust's borrowing and ownership systems in order to support multiple address spaces. In addition, a functioning implementation of the accepted RFC for address spaces with associated tests demonstrating correct behaviour would be presented.

5.8 Implementation of device code validation and structure generation using procedural macros (WP8)

Heterogeneous systems often impose limitations on the programming constructs that can be employed in device code, due to limitations in hardware. For example, recursion is often unsupported.

In this work item, a procedural macro will be implemented which will be used to annotate functions as device functions. This macro will desugar into the compiler attribute used for offloading.

Using the token stream of the function being processed, the procedural macro will produce an abstract syntax tree of the call graph to validate that no unsupported language features or programming patterns are being employed.

In addition, structures will be generated which are specialised for each device function for the purpose of storing extra information required for kernel execution.

Deliverable: A functioning implementation of the accepted RFC for address spaces in Rust with associated tests demonstrating correct behaviour.

5.9 Experimentation with programming model design (WP9)

Existing heterogeneous programming models vary widely in the API that they expose and this can have substantial impact on the performance achievable by the heterogeneous platform and on the productivity that can be achieved.

A review of existing heterogeneous programming model APIs should be conducted to determine the most suitable approach for this project. In addition, other APIs that could be relevant should be included, such as Vulkan - a Khronos standard for low-overhead graphics and computing. Furthermore, popular libraries and frameworks in the Rust ecosystem should be surveyed to ensure that the final API is idiomatic.

After the review of existing APIs has been completed, experimentation and basic implementations would be created to test the design's suitability for real-world applications.

Deliverable: Report detailing the API that is best suited to the heterogeneous programming model with justifications for each of the key decisions.

5.10 Remaining runtime implementation (WP10)

Remaining runtime features such as the scheduling of device code execution and data transfer between host and device will be implemented. By this point, it will have been determined which existing heterogeneous programming model will be being used as a backend - OpenCL or CUDA.

This work item will depend on the structures generated in WP8 which will contain information such as the offset of different parameters in the function, which can be required for some APIs in OpenCL or CUDA.

In addition, this functionality requires information about the data that requires transferred to and from the device and the data dependencies between device functions, which will be accumulated from the API calls designed WP9 into structures generated for each device function.

Deliverable: A functioning heterogeneous programming model which is capable of running a suite of benchmarks described in WP11 and a set of tests which demonstrate intended behaviour on all supported platforms.

5.11 Benchmarking (WP11)

An implementation of standard benchmarks for heterogeneous programming models, such as Rodinia or SPEC ACCEL, would be implemented and optimised. Equivalent implementations in other heterogeneous programming models, such as OpenCL, CUDA or OpenMP (as discussed in Section 4) would be implemented or acquired.

Execution of the benchmarks could then proceed on standardised hardware. To ensure a fair comparison, benchmarks would run their computation a pre-determined number of times before benchmarking begins to "warm-up" (this avoids incorrect data as a result of caching in hardware). In addition, benchmarks would be required to be able to run all computations without restarting the process (this avoids process startup contributing to noise in the results).

Deliverable: Report detailing the performance of the implemented heterogeneous programming framework in comparison with other heterogeneous programming models, including in-depth analysis of the results.

5.12 Correctness Validation (WP12)

In addition, in order to verify the improvements to correctness and memory safety, fuzzing would be performed on all of the benchmark implementations on all heterogeneous platforms. Some analysis would be performed on any crashes and segmentation faults discovered during this process to determine the root cause of any failures encountered.

Deliverable: Report containing data obtained from fuzzing including analysis of any failures found.

6 Risks

One risk faced by the project is that implementation of address space support, which utilises the existing support in Rust's borrow checker for lifetimes in the language, uncovers a case where there is an irreconcilable clash between the rules required for address space inference and those for lifetimes. If this were the case, an alternate approach could be implemented.

Under this circumstance, a new smart pointer type could be introduced into the Rust standard library which represent pointers into different address spaces. Smart pointers are data structures which act like pointers and are commonly used in Rust. Some special-casing of any new smart pointer types would be required in the Rust compiler so that it is correctly lowered to the appropriate address space when the Rust compiler is generating LLVM IR. Infrastructure which enables the special-casing of standard library types already exists in the Rust compiler in the form of *language items*, so this mechanism would not need to be added.

Another risk faced by the project is that RFCs to upstream features required by this project into the Rust language would not be accepted by the language's community and project's language team. Should this occur, the potential impact of the project would be greatly reduced. While the project could continue in a fork of the compiler for research purposes - results of which could later influence the acceptance of RFCs to achieve similar goals - it would make adoption of the heterogeneous programming model implemented in this research by academia or industry less likely.

7 Measurable Outcomes

- Improved productivity of programmers working with heterogeneous systems. due to utilisation of language features such as algebraic data types and increased compile-time checking.
- Increased experimentation and implementations of heterogeneous runtimes. These runtimes would leverage the compiler foundations implemented by this research and utilise the runtime APIs of existing lower-level heterogeneous programming models such as OpenCL, CUDA or OpenMP. For example, this could lead to advances in scheduling algorithms for functions which run on heterogeneous devices.
- Improved correctness of heterogeneous systems and decreased occurrence of vulnerabilities due to Rust's compile-time checking of memory safety.
- Improved performance as new optimisations are applied to generated host and device code as a result of new guarantees about pointer aliasing.

8 Academic Impact and National Importance

This research contributes to EPSRC's *programming languages and compilers* research area by contributing to the development of a new heterogeneous programming model in the Rust programming language. In addition, EPSRC's *software engineering* research area is also relevant, as the programming model developed by this research will enable advances in correct, principled and secure software engineering for heterogeneous systems. *Software engineering* is a growth area for EPSRC.

In a society becoming increasingly reliant on software, vulnerabilities are an important problem that can have a real impact on the safety and security of everyday users. Microsoft, whose Windows operating system owns 72% of the UK market share, has found that over 70% of the CVEs assigned for vulnerabilities in Microsoft products are as a result of memory unsafety bugs [1]. Existing heterogeneous programming models, such as OpenCL, OpenMP, CUDA or SYCL are all written in languages which have a fundamentally unsafe memory model.

More recently, specialised processors for artificial intelligence and computer vision are being used in heterogeneous systems and deployed in safety critical systems (such as self-driving cars), where correctness is vitally important.

The proposed project addresses the vulnerabilities and bugs that result from the unsafe memory models through implementation in the Rust programming language. Rust's unique ownership and borrowing system provides compilation-time guarantee of memory safety unparalleled by existing heterogeneous programming models.

Within academia, the decreased barrier-to-entry for experimentation with heterogeneous programming models that would be enabled by this project, through the explicit decoupling of compiler and runtime concerns, would enable new avenues of research. For example, currently, the trade-offs inherent in the application programming interfaces exposed by existing heterogeneous systems is challenging to research due to the compiler support required for any change to a C/C++-derived heterogeneous programming model's interface.

References

[1] M. S. R. Center. A proactive approach to more secure code, 2019 (accessed February 9, 2020). https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/.

[2] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). *SIGPLAN Not.*, 47(6):1–12, June 2012.

[3] Z. Jin and H. Finkel. Evaluation of medical imaging applications using sycl. In *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 2259–2264, Nov 2019.

[4] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W. Hwu, H. Li, M. Müller, W. Nagel, M. Perminov, P. Shelepugin, K. Skadron, J. Stratton, A. Titov, K. Wang, M. Van Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran. Spec accel: A standard application suite for measuring hardware accelerator performance. In S. Hammond, S. Jarvis, and S. Wright, editors, *High Performance Computing Systems*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 46–67. Springer-Verlag, 1 2015.

[5] J. Kim, T. T. Dao, J. Jung, J. Joo, and J. Lee. Bridging opencl and cuda: A comparative analysis and translation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[6] S. Lee and R. Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010.

[7] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, page 101–110, New York, NY, USA, 2009. Association for Computing Machinery.

[8] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler. Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ARMS-CC '17, page 1–6, New York, NY, USA, 2017. Association for Computing Machinery.

[9] H. Perkins. Cuda-on-cl: A compiler and runtime for running nvidia® cudatm c++11 applications on opencltm 1.2 devices. In *Proceedings of the 5th International Workshop on OpenCL*, IWOCL 2017, New York, NY, USA, 2017. Association for Computing Machinery.

[10] I. Z. Reguly. Performance portability of multi-material kernels. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 26–35, Nov 2019.

[11] P. Sathre, M. Gardner, and W.-c. Feng. On the portability of cpu-accelerated applications via automated source-to-source translation. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2019, page 1–8, New York, NY, USA, 2019. Association for Computing Machinery.

[12] C. Shen, X. Tian, D. Khaldi, and B. Chapman. Assessing one-to-one parallelism levels mapping for openmp offloading to gpus. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'17, page 68–73, New York, NY, USA, 2017. Association for Computing Machinery.

[13] C. Su, P. Chen, C. Lan, L. Huang, and K. Wu. Overview and comparison of opencl and cuda technology for gpgpu. In *2012 IEEE Asia Pacific Conference on Circuits and Systems*, pages 448–451, Dec 2012.

[14] N. Sultana, A. Calvert, J. L. Overbey, and G. Arnold. From openacc to openmp 4: Toward automatic translation. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, XSEDE16, New York, NY, USA, 2016. Association for Computing Machinery.

[15] T. Vanderbruggen and J. Cavazos. Generating opencl c kernels from openacc. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, New York, NY, USA, 2014. Association for Computing Machinery.

[16] N. Wilson. Dcompute: Compiling d to spir-v for seamless integration with opencl. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18, New York, NY, USA, 2018. Association for Computing Machinery.