AIWC for the masses: Supporting Architecture-Independent Workload Characterization on OpenMP, OpenACC, CUDA and OpenCL

Beau Johnston
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
Australian National University
Canberra, Australia
beau.johnston@anu.edu.au

Jacob Lambert
University of Oregon
Eugene, Oregon, USA
jlambert@cs.uoregon.edu

Josh Milthorpe
Australian National University
Canberra, Australia
josh.milthorpe@cs.anu.edu.au

Jeffrey Vetter
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
vetter@ornl.gov

***Abstract*—The next-generation of supercomputers will feature a diverse mix of accelerator devices. These accelerators span an equally wide range of hardware properties. Unfortunately, achieving good performance on these devices has historically used multiple programming languages, but in the present day results in fragmentation of implementation – where an increasing amount of a programmer's effort is expended to migrate codes between languages in order to examine performance of a new device when it hits the market. We have previously shown that presenting the characteristics of a code in a architecture-independent fashion is useful to predict exection times. From examining these highly accurate predictions we propose that Architecture-Independent Workload Characterization (AIWC) metrics are also useful in determining the suitability of a code and potential accelerators. To this end, we extend the usability of AIWC by supporting additional programming languages common to accelerator-based High-Performance Computing (HPC). We use two compilers to perform source-to-source level translation between CUDA-to-OpenCL, OpenMP-to-OpenCL and OpenACC-to-OpenCL and extend the usefulness of the AIWC tool by evaluating the base execution behaviour on these outputs. Essentially, We examine how AIWC metrics change between OpenMP, OpenACC, CUDA and OpenCL implmentations of identical applications commonly used in scientific benchmarking.**

***Index Terms*—architecture independent workload characterization, portability, OpenCL, CUDA**

I. Introduction

High-Performance Computing (HPC) today is dominated by closed, proprietary software models. Dispite OpenCL having existed for over a decade, it has generated little traction/adoption by the scientific programming community at large. Fragmentation between implementations of scientific codes already exists. Many lines of legacy code have already been reimplemented in accelerator directive-based languages such as OpenACC and OpenMP, some computationally intensive kernels have been implemented in CUDA. Unfortunately, many of these codes must target a new accelerator within a few years, as the supercomputer is rapidly updated, and the accelerators used on a node are just as quickly replaced – often by different vendors and likely another class of accelerator (MIC, GPU or FPGA). We can easily imagine a world where developers at research laboratories will have full-time jobs rewriting the same codes on loop, taking years optimising and rewriting kernels for the upcoming system. This will only grow as the HPC center's dependence on heterogenous accelerator architectures increases, pushed by energy-efficiency requirements in the era of exascale computing, and is untenible.

Workload characterization is an important tool to examine the essential behaviour of a code, its underlying structure, and identifying the performance limiting factors – an understanding of the latter is critical when considering the portability between accelerators. These by examining the SPIR-V execution traces – on an abstract OpenCL device. For instance, a code with regular memory accesses, predictible branching and is highly parallel (utilizing a large number of threads) is a suitable candidate for selection for a GPU type of accelerator, conversely, inherently serial tasks are more suited for CPU devices which commonly offer a higher clock-speed. In the past we have used AIWC and Workload Characterization to perform accurate run-time predictions of OpenCL codes over multiple accelerators – motivated to the automatic scheduling of kernels to the most appropriate accelerator on an HPC system based on these essential characteristics. We have also shown that these characteristics are useful in guiding a developers efforts to achieving good performance on accelerators by outlining the potential bottlenecks of the implementation of an algorithm (in the amount of parallelism available, memory access patterns and scaling over problem sizes, etc). Both motivations will be undone if we don't get industry adoption and communtity support for SPIR-V (and the OpenCL runtime).

Recent advances with SYCL, HIP/ROCM, and oneAPI may increase adoption of open-source models, however, why wait? We can leverage the backend of these frameworks, SPIR-V, which are based on, and support the OpenCL runtime and memory-model. Meanwhile, compilers are maturing and are increasingly able to provide source-to-source translations and code transformations, and are capable of generating low-level device-optimized code, allowing implemenions in one language to be mapped to another. The goal of this paper is to extend AIWC support for all languages, and is achieved firstly by assessing the AIWC feature-space outputs of contemporary source-to-source compiler/translator tools, and secondly, highlights the differences against a native OpenCL implementation – to identify any potential inefficiencies of the translation by these tools.

In summary, in this paper we extend the use of AIWC to evaluate the current state-of-the-art in source-to-source translation to primarilily examine the feasibility of leveraging existing language implmentations of codes (as an example of those currently popular) to automatically map back to OpenCL. We summarize the common languages used on accelerators and survey the communities interest in each, this is done to motivate our interest in tooling and offering support for evaluating the back-end generation of SPIR-V codes. Related work is discussed, then we present our methodology highlighting our selection of source-to-source translation tools used in our experiments. We present results then close with a summary of our findings and the direction for future-work.

II. Accelerator Programming Frameworks and their Adoption

CUDA – NVIDIA's proprietary software model – has been around since 2007. It has been widly adopted to repurpose GPUs from rendering in gaming workloads to highly-parallel compute intensive tasks common to scientific codes. Unfortunately, it is a single-vendor language, thus CUDA codes are executed solely on NVIDIA devices. There has been some recent activity by AMD with ROCm, which aims to automatically translate CUDA codes to their HIP (Heterogeneous-Compute Interface for Portability) framework – an AMD defined standard and modified clang compiler.

OpenCL (Open Compute Language) was introduced shortly after CUDA (in 2009) as a standard agreed upon by accelerator vendors. This ideally allowing a code to be written once and executed on any (OpenCL complient) device. Vendors typically each develop their own-backend OpenCL runtime. This can result in greater variation in performance since vendors can choose there extent of support and optimization. To remedy this POCL (Portable Computing Language) is an Open Source initive providing an implementation of the OpenCL standard. It is analogous to HIP but for OpenCL instead of CUDA – both leverage Clang and LLVM, Clang for the front-end compilation of codes and LLVM for the kernel compiler implementation. POCL offers backends to NVIDIA devices (via a mapping of LLVM to CUDA/PTX), HSA (Heterogeneous System Architecture) GPUs such as those offered by ARM and AMD, along with CPUs and ASIPs (Application-Specific Instruction-set Processor).

OpenMP [1] exists as one of the most widely-used tools in high-performance computing, in-part because of it's high-level directive-based approach and broad availability. While OpenMP has traditionally been used to generate multi-threaded code on CPU devices, the recent addition of offloading directives in the OpenMP 4.X+ standards has extended OpenMP to also support accelerator devices, primarily GPUs. Similarly to OpenACC, the OpenMP offloading directives provide a high-level alternative to low-level offloading models like OpenCL and CUDA, and provide existing OpenMP programmers a familiar entrance to accelerator computing.

OpenACC [2] has been developed as a high-level directive-based alternative to low-level accelerator programming models. Lower-level approaches like CUDA and OpenCL typically require the programmer to explicitly manage mappings of parallelism and memory to device threads and memory, and require detailed knowledge about the target device. In contrast, OpenACC allows programmers to augment existing programs with directives to generically expose available parallelism, shifting the burden of thread and memory mapping to the underlying compiler. Because of its straightforward API and implications for performance portability, OpenACC has become an attractive option for domain scientists interested in exploring accelerator computing.

The confidentiality of most super-computer scale scientific codes means it is unknown what percentage of kernels are developed in CUDA, OpenCL, OpenMP or OpenACC. If we assume the open-source community adoption of accelerator programming frameworks reflects their popularity used by the scientific HPC community, a quick survey of GitHub can help indicate their adoption. As of January 2020, the number of repositories including CUDA in the title was 18k, OpenCL was in 8K repositories, OpenMP 5K repositories and OpenACC was featured < 400 repositories. If we compare the lines of code as a measure of language popularity, CUDA was listed more than 8M times, OpenCL occured 2M times, OpenMP 2M times while OpenACC was featured just ~124K times. From the newer frameworks, SYCL is either in the title or description of 144 repositories, 402k lines of code in the search contained SYCL. At the time of writing, OneAPI is less than 6 months old which biases this comparison, nonetheless there are 56 repositories and 4k lines. This survey was performed by searching for "CUDA", "OPENCL", "OPENMP", "OPENACC", "SYCL" and "ONEAPI" in GitHub[1]. We are unable to be more precise around actual repositories using accelerator programming frameworks because OpenCL, OpenMP and OpenACC are not tracked as languages by GitHub – CUDA is listed as a language and has ~5.9k repositories listed as such. Based on these results, we suspect that OpenCL isn't in the pole position to winning the race to adoption, however, translator tools may allow us to ensure OpenCL is used regardless – as a backend runtime to SPIR-V intermediate representation.

Mapping back to a common OpenCL runtime is an obvious compromise, since it supports the greatest range of accelerator devices, and by supporting multiple front-end languagues and allowing developers the freedom to use desired language, still increases the impact of OpenCL. This common backend holds the potential to avoid the fragmentation and repeated reimplementation as systems are updated and accelerators replaced. Having efficent tools to enable this mapping is paramount. We use AIWC to evaluate the similarities and differences between outputs of translation tools on functionally equivilent kernel codes and show that it can be used to guide understanding of the mapping between versions and potential improvments to these tools.

III. Related Work

A. SPIR-V

- LLVM common IR of all kernel codes

B. AIWC

- Brief about how AIWC currently operates on the Oclgrind LLVM OpenCL device simulator.
- The collected metrics have shown to accurately represent the codes characteristics and the suitability of accelerator devices with precise execution time predictions
- Offers insights around optimization of a code for accelerators by presenting interesting summaries for the developer to consider.
- Unfortunately, Oclgrind – and thus also AIWC – only support the OpenCL programming language by simulating an ideal (and abstract) OpenCL device.
- Abstract OpenCL device simulation enables that AIWC already architecture independent

C. OpenARC

The Open Accelerator Research Compiler (OpenARC) [3] has been developed as a research-oriented OpenACC and OpenMP compiler. OpenARC performs source-to-source translations and code transformations to generate low-level device-optimized code, like CUDA or OpenCL, specific to a targeted device. OpenARC's primary strength is it's ability to enable rapid prototyping of novel ideas, features, and API extensions for emerging technologies.

In this work, we leverage OpenARC's OpenACC to OpenCL and OpenMP to OpenCL translations and use the output with AIWC to characterize the workloads resultant translation. This integration allows us to extend AIWC to characterize high-level codes written with OpenACC and OpenMP.

D. Coriander

At first glance we may be hopeful that AMD's HIP could be used for all CUDA to OpenCL backend translations, however, sadly, it does not use OpenCL as a back-end, instead choosing to use LLVM to generate kernels for the HSA Runtime and Direct-To-ISA skipping intermediate layers such as PTX, HSAIL, or SPIR-V. It is unknown why this decision was made and by skipping SPIR we are unable to perform the analysis with AIWC over the output since it skips any potential abstract "ideal"/"uniform" device, useful to checking errors, validating memory accesses and performing workload characterization that AIWC and Oclgrind provides. Instead, we chose Coriander for the functionality of CUDA to OpenCL source-level translation.

TODO, summarize Coriander

IV. Methodology

- For each benchmark a comparison of the AIWC feature-spaces of the OpenCL vs
  - OpenACC
  - OpenMP
  - CUDA is presented. The Rodinia Benchmark Suite was selected since they offer a number of scientific benchmarks each implemented in all the targeted programming frameworks we compare.

Coriander was used to convert a subset of the Rodinia Benchmark suite from CUDA to OpenCL translation while OpenARC was used for both OpenMP to OpenCL and OpenACC to OpenCL translation.

Source code changes include C++ programs are unsupported as input programs, as such we need to change the `bfs` application extension. Most of the translation was autonomous, only one explicit typecast for a user defined struct needed to be clarifed use 'struct' keyword to refer to the type. Use of the preprocessor also doesn't work

[1] www.github.com

within OpenARC thus `TRANSFER_GRAPH_NODE` which was `typedef`ed to 1 was explicitly replaced.

- Steps taken to perform code translation from openacc to opencl kernel and wrapper

Use of Coriander was largely autonomous except it lacks support for `cudaMemcpyToSymbol` function calls.

Deprecated calls in the code-base to `cudaThreadSynchronize` also required replacing with `cudaDeviceSynchronize`. As such, the translation effort was almost entirely focused on replacing these operations with . Manually replacing certain includes – for instance `<math.h>` to `<cmath>` – were needed for the translation effort since Coriander is tied to LLVM 3.9 and is hard-coded to only support C++11. **TODO:** examine bfs (maybe fix invalid memory accesses) **TODO:** list which demos weren't able to build using Coriander

- Coriander and AIWC were required to both be statically linked against the same version of LLVM due to an error which occurs when two seperate instances of LLVM are registered into the same vtable. **do I mean vtable?** error " 'phi-node-folding-threshold' registered more than once" These changes have been provided in as an artefact with Docker incase you wish to try for yourself [2] Also many of the CUDA codes would crash during the translation to OpenCL – thus, only a small subset of the Rodinia Benchmark suite are presented, nonetheless this work forms a proof-of-concept and these issues within Coriander can be fixed.

V. Results

We now present the results of this comparison over Gaussian Elimination (GE). The GE benchmark was provided by The Rodinia Benchmark Suite [4] with an OpenACC, CUDA and OpenCL implementation. We found the existing implementations to lack a perfect mapping between versions, in particular our work modifies the partitioning of work to ensure an equivilent division of work is allocated between versions. We also built the OpenMP version – with 4.0 accelerator directives – based on the OpenACC version. All implmentations have been divided into two discrete computationally intensive regions/kernels – known as `Fan1` and `Fan2`. `Fan1` calculates the multiplier matrix while `Fan2` modifies the matrix A into the Lower-Upper Decomposition. For our experiments the application was evaluated over a fixed dataset of a 4x4 matrix, where the same data was applied to all the implementations.

A. *Fan1*

4 work-items were used for all implementations, however the way parallelism is expressed differ between languages and for an apples to apples comparison in generated AIWC features required code changes to the kernel. These changes are listed in full in the associated Jupyter artefact

Add url once this is pushed to github

, for convenience a summary

In OpenCL

The initial CUDA implementation had a variation in parallelism due to the way it is expressed in CUDA compared to OpenCL. To this end, it was adjusted to more closely mirror the behaviour of the OpenCL version offered in the benchmark. The `Block` size was explicitly set to the `MAXBLOCKSIZE` (512 threads), our change: `block_size = (Size % MAXBLOCKSIZE == 0) ? MAXBLOCKSIZE : Size;` states that if we have smaller work to do than the max block size, just run 1 block of that size, which mirrors the way OpenCL expresses parallelism of this benchmark – i.e. the `global workgroup size` is the total number of threads to execute run in teams of `local workgroup size`. Thus, the CUDA implementation went from `512` workitems being invoked (where only 4 of them did any meaningful work) to `4` workitems being run.

In OpenMP 4 threads are explictly requested by setting the `OMP_NUM_THREADS=4` environment variable at runtime while work-items in OpenACC was manually modified to support the same number of parallelism as the other versions. **mention other omp changes**

OpenARC uses `workers` and `gangs` variables to express parallelism in the OpenACC to OpenCL setting. To this end, we added these variables and the `MAXBLOCKSIZE` to be 512 to be equivilent to the CUDA version of the Gaussian Elimination benchmark. `gangs = (Size % MAXBLOCKSIZE == 0) ? MAXBLOCKSIZE : Size;` is set to be analagous to `block_size` (`block_size = (Size % MAXBLOCKSIZE == 0) ? MAXBLOCKSIZE : Size;`) which we added to the CUDA version, similarly, `workers = (Size/gangs) + (!(Size%gangs)? 0:1);` is identical to the CUDA version of `grid_size` (`grid_size = (Size/block_size) + (!(Size%block_size)? 0:1);`). Finally, the OpenACC pragmas where modified to explicitly use the `workers` and `gangs` variables: from `#pragma acc parallel loop present(m,a)` to `#pragma acc kernels loop independent gang(gangs) worker(workers)`.

In summary, the OpenACC implementation went from `64` workitems being invoked (where only 4 of them did any meaningful work) to `4` workitems being run.

The AIWC metrics of this kernel are presented in figure~1.

Metrics (along the x-axis) have been grouped by category and is indicated by colour. These categories outline the overall type of characteristic being measured by each metric. The blue metrics (Opcode and Total Instruction Count) show the "Compute" category (which denote the

[2] https://github.com/ANU-HPC/coriander-and-oclgrind

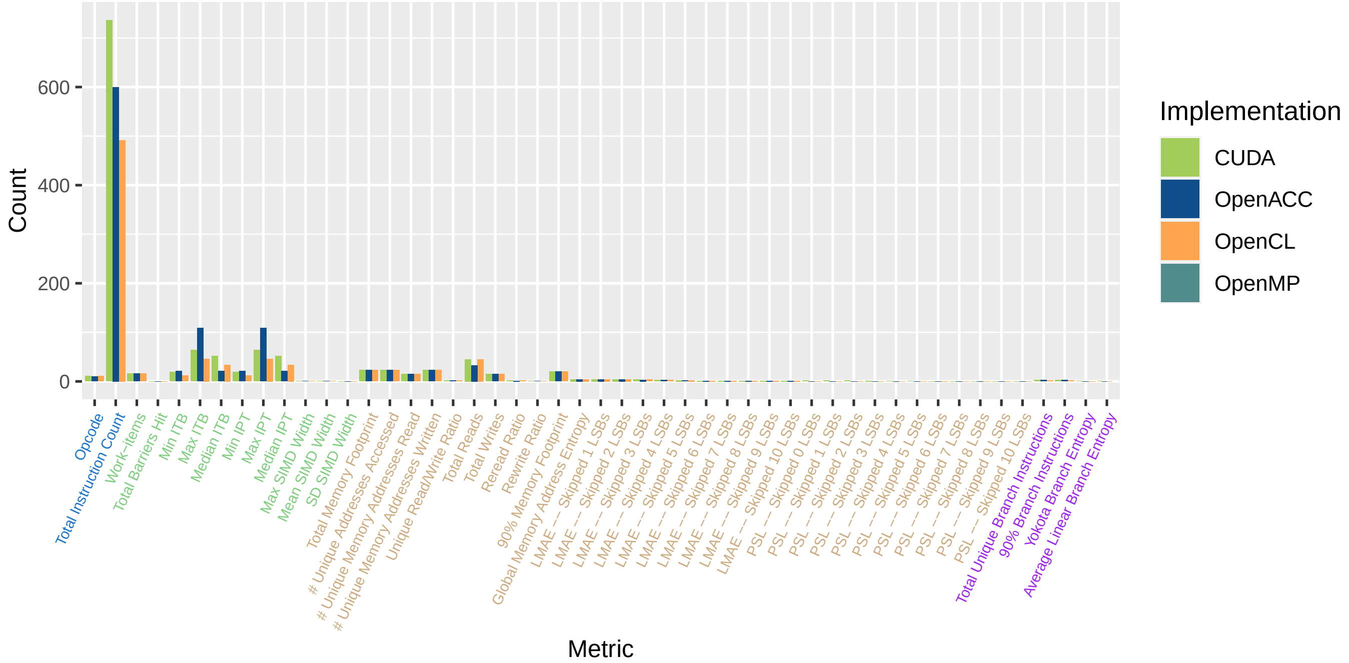

Figure 1: Absolute difference in AIWC metrics between each translated Fan1 kernel implementation.

amount of work to be done per thread and the diversity of the instruction sets required), metrics in green present "Parallelism" type metrics (these metrics are broadly around number of threads available in the workload, the amount of independence between threads and whether vectorization/SIMD is appropriate), "Memory" are presented in beige (and are included to collect the spread, proximity and raw number of memory addresses accessed), while purple metrics indicate "Control" (the predictability of branching during control flow of the workload). A full list and description of these metrics is available [5] but for brevity is not further discussed in this paper.

The y-axis presents the absolute count of each AIWC metric. The bars have been coloured according to Implementation (as shown in the legend) with CUDA in green, OpenACC in blue, OpenCL in tan and OpenMP in grey. Each metric has the four implementations grouped together, thus Figure~1 gives a visual inspection of the feature-space comparison of each metric between all implementations. It's expected that OpenCL should be the lowest count – or the lowest overhead – of all the implemenations regardless of metric, since it serves as the baseline; a compiler doing source-to-source translations would have to be doing additional optimizations to result in lower counts than the OpenCL baseline.

A flat-line at 100% is ideal, since it shows no discernable difference between metrics captured by AIWC. In other words, if the applications workload characteristics are identical between languages the translator is doing an excellent job in preserving the structure (in terms of memory accesses, parallelism and compute work) of the code regardless of language. Thus, in a perfect translation all AIWC metric counts would be equal between all implementations.

Figure~3 shows Fan1's comparison between implementations but have been normalize to show the relative difference between each implementation against the baseline OpenCL counts; and presents the same results but allows us to zoom into the finer details of these raw counts.

We see no causes where the compiler improves beyond the initial OpenCL baseline.

what about low PSL?

B. *Fan2*

Thus we can use example to illustrate how AIWC metrics highlight discrepencies between languages and implementations and how it can be used to guide sourcecode changes. Two separate loops in the `Fan2` function were consolidated into one, to mirror how the task is performed in the OpenCL and CUDA implementations of the algorithm.

VI. Conclusions and Future Work

This work extends the applicabilty of AIWC by supporting multiple languages and have demonstrated it's usefulness to evaluate the overhead and complexities of the OpenCL output from two source code translation tools. We believe this methodology can be boardly more useful in the future development of translators.

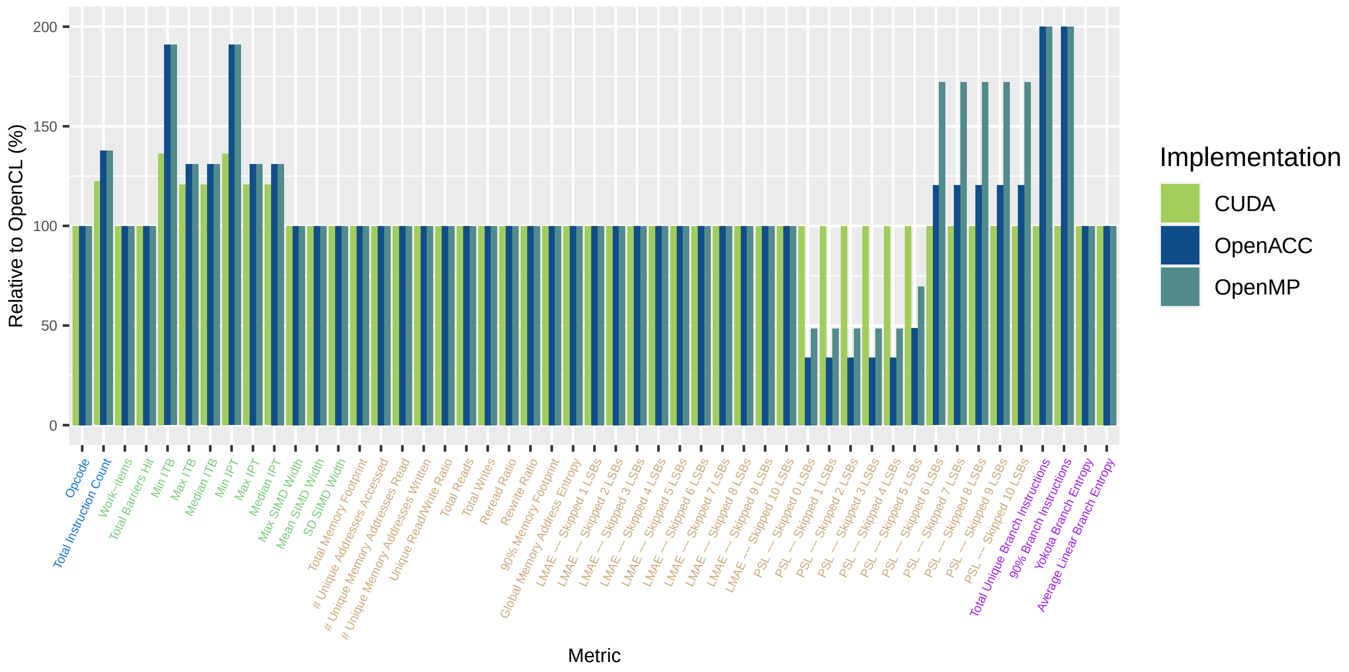

Figure 2: Relative AIWC metrics between each translated implementation of the Fan1 kernel against the baseline OpenCL.

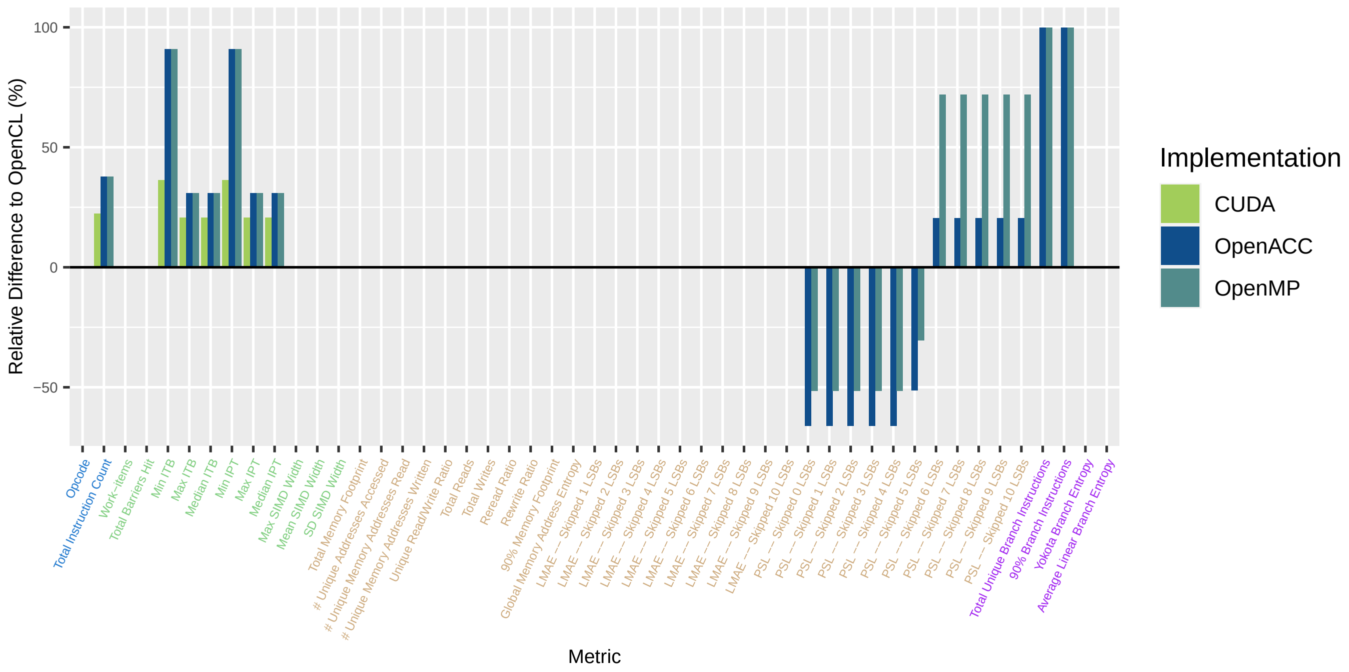

Figure 3: Relative difference in AIWC metrics between each translated Fan1 kernel implementation against the baseline OpenCL.

Since AIWC metrics are based on an abstract/ideal OpenCL device simulator there are architecture-specific optimizations that may occur when you target the LLVM-IR/SPIR to a specific accelerator however this is expected to be consistant reguardless of the compiler front-end, and the former issue is not a subject for this work.

We will apply the methodology of generating AIWC feature-spaces to other languages, specifically OpenMP and OpenACC – since this is the typical means of accelerating conventional HPC workloads using many-core CPUs. The ability to examine the characteristics of these kernels in large code-bases allow optimization work to occur on these kernels, effectively guiding a developers hand to optimize a code by providing strategies to minimise certain AIWC metrics shown to be advantageous to specific accelerators. Additionally, examining the AIWC features of existing code-bases may facilitate identifying a better accelerator match as they become available.

Thus, this work identifies the potential overheads when translating between functionally identical implementations of kernels written in different languages by examining differences in ther respective AIWC metrics. We also offer a methodology which uses AIWC over a number of tests kernels, and against an OpenCL implementation as a base-line, is an option to assess the suitability of any changes made to the translator. This work will assist the improvement of the translation tools on offer, increasing the adoption of SPIR-V and the use of the OpenCL runtime behind-the-scenes, resulting in less fragmentation between software models and languages on contemporary HPC systems.

This paper shows that the visual inspection of AIWC metrics facilitates a high-level (and quick) overview of computational characteristics of kernels, and we've found that how they change has enabled us to compare the execution behaviour of codes in response to two different compilers performing translation. We've seen how source-code modifications in our selected benchmark kernels change these features – in our instance to more closely resemble an OpenCL baseline.We believe the same methodology will be useful for compiler engineers to evaluate their own translators – especially with the increasing use of LLVM as a backend, on which AIWC and this approach is based. This is useful since it is abstracted to an ideal OpenCL device and, as such, is free from micro-architecture and architectural details. We proprose this methodology will also encourage application developers of scientific codes to take a deeper-dive into their codes, and more generally, our future work will examine how AIWC metrics provide a developer insights around the suitability of the kernel code when initially selecting for, then optimizing on, a specific accelerators.

References

[1] L. Dagum and R. Menon, "OpenMP: An industry-standard api for shared-memory programming," *Computing in Science & Engineering*, no. 1, pp. 46–55, 1998.

[2] OpenACC, "OpenACC: Directives for Accelerators." [Online]. Available: http://www.openacc.org, 2011.

[3] S. Lee and J. S. Vetter, "OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *Proceedings of the 23rd international symposium on high-performance parallel and distributed computing*, 2014, pp. 115–120.

[4] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 ieee international symposium on workload characterization (iiswc)*, 2009, pp. 44–54.

[5] B. Johnston, "Characterizing and predicting scientific workloads for heterogeneous computing systems," PhD thesis, [Online]. Available: https://openresearch-repository.anu.edu.au/handle/1885/162792, 2019.