Towards Language-Agnostic Architecture-Independent Workload Characterization for HPC Accelerators

Author(s) omitted for blind review

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 required multiple programming languages with a separate implementation for each device. In the present day this results in the fragmentation of implementation – where an increasing amount of a programmer's effort is expended to migrate codes between languages in order to use a new device. We have previously shown that presenting the characteristics of a code in a architecture-independent fashion is useful to predict execution 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 for 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 from 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 implementations of identical applications. This is useful to guide a developer through optimizations, extend predictions for scheduling over languages and evaluate differences between compilers, frameworks and toolchains.

CCS CONCEPTS

• **Software and its engineering** → *Translator writing systems and compiler generators*;

KEYWORDS

compilers, workload characterization, portability

ACM Reference Format:
Author(s) omitted for blind review. 2020. Towards Language-Agnostic Architecture-Independent Workload Characterization for HPC Accelerators. In *Proceedings of 49th International Workshop on Parallel Processing (ICPP '20).* ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.475/123_4

ICPP '20, 17-20 August, 2020, Edmonton, Canada.

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

1 INTRODUCTION

High-Performance Computing (HPC) today is dominated by closed, proprietary software models. Despite 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, while some computationally intensive kernels have been implemented in CUDA. Unfortunately, many of these codes must frequently target new accelerators, as supercomputers are 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 universities, in industry and research laboratories will have full-time jobs rewriting the same codes on loop, taking years optimising and rewriting kernels for the upcoming system. This demand will only grow as the HPC center's dependence on heterogeneous accelerator architectures increases, pushed by energy-efficiency requirements in the era of exascale computing, and is untenable.

Workload characterization is an important tool to examine the fundamental behaviour of a code and its underlying structure. It facilitates identifying performance-limiting factors – an understanding of which is critical when considering the portability between accelerators. Characterization, in this setting, occurs by examining the LLVM execution traces on an abstract OpenCL device and tracking and recording statistics about the fundamental nature of these codes. For instance, a code with regular memory accesses and predictable branching that 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 Architecture-Independent Workload Characterization (AIWC) to perform accurate runtime predictions of OpenCL codes over multiple accelerators – motivated by the goal of automatically scheduling 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 developer's efforts in optimizing 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). Unfortunately, both

motivations will be undone without industry adoption and community support for the OpenCL runtime.

Standard Portable Intermediate Representation (SPIR) is an intermediate language for parallel compute and graphics by Khronos Group, primarily treated as an abstract LLVM based subset of codes to allow portable transitions between OpenCL supported devices. It is low-level enough for general compiler optimizations to occur, but is more abstract than the allocation registers and micro-architecture-specific optimizations. Recent advances with SYCL, HIP/ROCM, and oneAPI may increase adoption of open-source models and present an alternative to the OpenCL-only characterization approach. These frameworks are also based on the SPIR representation, support the OpenCL runtime and memory model, and can be leveraged by the AIWC and Oclgrind debugging tools. Meanwhile, compilers are maturing and are increasingly able to provide source-to-source translations and code transformations, to generate low-level device-optimized code, and to allow implementations in one language to be mapped to another. The goal of this project is to extend AIWC support for all languages (1) by assessing the AIWC feature-space outputs of contemporary source-to-source compiler/translator tools, and (2) by exploring any differences compared to a native OpenCL implementation to identify any potential inefficiencies of the translation by these tools.

In summary, in this project we extend the use of AIWC to evaluate the current state-of-the-art in source-to-source translation. Primarily we examine the feasibility of leveraging existing language implementations of codes that automatically map back to OpenCL. In Section~**??** we summarize the common languages used on accelerators and quantitatively survey the community's interest in each. This is done to motivate our interest in tooling and offering support for frameworks that rely on back-end generation of SPIR codes. In Section~**??** we discuss related work. In Section~**??** we present our methodology, highlighting our selection of source-to-source translation tools used in our experiments. We present our expiremental results in Section~5, and conclude in Section~**??** with a summary of our findings and the directions for future-work.

2 ACCELERATOR PROGRAMMING FRAMEWORKS AND THEIR ADOPTION

CUDA [10] – NVIDIA's proprietary software model – has been around since 2007. CUDA has been widely adopted by the scientific community as a means to repurpose GPUs from traditional 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 their ROCm software stack to provide an alternative. For example, their Hipify tool automatically generates a HIP (Heterogeneous-Compute Interface for Portability) representation – an AMD defined standard – from input CUDA code.

OpenCL [4] (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 complaint) device. Vendors typically each develop their own-backend OpenCL runtime. This can result in greater variation in performance since vendors can choose their extent of support and optimization. To remedy this POCL (Portable Computing Language) is an Open Source initiative providing an implementation of the OpenCL standard. POCL's relationship to OpenCL is analogous to HIPs relationship to 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).

OpenACC [11] 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.

OpenMP [3] exists as one of the most widely-used tools in high-performance computing, in-part because of its 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.

The confidentiality of many supercomputer-scale scientific codes means it is unknown what percentage of kernels are developed in CUDA, OpenCL, OpenMP or OpenACC by the scientific community. If we assume the open-source community adoption of accelerator programming frameworks reflects their popularity in the scientific HPC community, a survey of GitHub can help indicate their relative adoption rates. As of January 2020, the number of github repositories including CUDA in the repository title was 18k; 8k for OpenCL, 5k for OpenMP 5K, and fewer than 400 for OpenACC. If we compare the lines of code (Github presented metric) as a measure of language popularity, github classified 8M lines of CUDA code, 2M lines of OpenCL, 2M lines of OpenMP, and 124K lines of OpenACC. Of the newer frameworks, SYCL is either in the title or description of 144 repositories, and github classified 402k lines of code as SYCL. At the time of

writing, OneAPI is less than a year old, and there are already 56 repositories and 4k lines of code related to it.

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 individual languages by GitHub. However CUDA is listed as a language with ~5.9k repositories listed. Based on these results, we suspect that OpenCL may not be positioned to win the race to adoption.

Appropriate support of translator tools will allow us to ensure OpenCL is utilized – as a back-end runtime supported on most devices. Mapping back to a common OpenCL runtime is an obvious choice, as it supports the greatest range of accelerator devices and multiple front-end languages. The idea of a common back-end representation like OpenCL potentially avoids fragmentations and repeated implementation as systems are updated and accelerators replaced, and having efficient tools to enable this mapping OpenCL is paramount. In this work, we use AIWC to evaluate the similarities and differences between outputs of translation tools on functionally equivalent kernel codes. We show that AIWC can be used to guide the understanding of the tools mapping between high-level source languages and OpenCL, and potentially guide improvements to these tools.

3 RELATED WORK

3.1 AIWC

The Architecture-Independent Workload Characterization (AIWC) tool describes a workload in terms of statistics around the programs fundamental behaviour. For instance, summary statistics on the arithmetic intensity, branching behaviour and types of parallelism and the granularity of parallelism expressed within the code. It operates as a plugin within the Oclgrind [12] LLVM OpenCL device simulator – namely it simulates an abstract OpenCL device and processes each work-item, it is lossless, deterministic and slower than executing on real accelerator hardware. During execution the AIWC plugin stores traces around behaviour that is representative of the workload while being removed from any phenomena that would impact the performance on actual hardware. These traces are evaluated at the end of each kernel run to compute the characterizing statistics. The listings of these statistics and how they are computed is available in associated literature [7]. The collected metrics have shown to accurately represent the codes characteristics and have shown their suitability in a predictive methodology where the summary statistics were (the only arguments) provided to perform precise execution time predictions over a range of accelerator devices [8].

The change in these metrics offer 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, utilizing translator tools to migrate codes written in other languages into OpenCL will increase the impact of AIWC.

3.2 OpenARC

The Open Accelerator Research Compiler (OpenARC) [9] 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.

3.3 Coriander

The Coriander tool has the functionality of translating CUDA to OpenCL codes. It achieves compiling the device-side (kernel) code into OpenCL C and by providing a compiler for host-side code, including memory allocation, copy, streams and kernel launches. Unlike OpenARC it skips source-to-source level translation and instead produces LLVM-IR/SPIR directly. We use Coriander to translate CUDA codes into OpenCL so we can evaluate the similarities in functionality of benchmarks with AIWC.

4 METHODOLOGY

Since AIWC metrics were selected to be architecture independent, it is a reasonable assumption that these metrics should also be language independent. The methodology to verify this assumption is outline in the following. Given a functionally identical application but implemented in OpenCL, CUDA, OpenMP and OpenACC, we utilise contemporary translation tools to generate AIWC compatible OpenCL codes, then generate AIWC metrics, and finally compare the AIWC feature-spaces of different languages against a baseline OpenCL version. Because AIWC analysis operates directly at the SPIR-V/LLVM-IR level, we utilize translation tools to lower the input languages to the desired abstraction level. Coriander was used to generate LLVM IR from CUDA input, while OpenARC was used for both OpenMP to OpenCL and OpenACC to OpenCL translation. An identical version of clang and LLVM (3.9.0) is used to lower the output OpenCL to SPIR from all tools.

We selected the Gaussian Elimination application from the Rodinia Benchmark Suite [2] for our evaluation. By design Rodinia applications are targeted toward accelerator-based systems, and many of the applications in the benchmark suite are represented in multiple programming models, including CUDA, OpenCL, and OpenMP. Furthermore, additional OpenMP and OpenACC implementations of various Rodinia

[1] www.github.com

applications have been developed by open source extension projects [5], which we utilize in our evaluation. The Gaussian elimination application is composed of two kernels, we present the comparison of the first `Fan1` kernel in Results (Section 5), `Fan2` was also analysed and available in the associated Jupyter artefact

reference artefact

but for brevity is omitted from the paper. We also discuss the required changes made to each implementation to get the closest approximation of work between versions.

Figure 1 presents a summary of the workflow and the various representations generated to interoperate between translators, compilers and AIWC. This shows how different source implemenations of the same algorithm can generate equivilent workload characterization metrics. The entire workflow is composed of several stages and representations – broadly progressing from source code written in various languages, computational intensive regions are translated into OpenCL kernels, compiler with Clang into SPIR, executed on the Oclgrind OpenCL device simulator with the AIWC plugin, AIWC then presents and stores these metrics as a set of features describing the characteristics of the code. OpenCL can skip the translation stage and kernel representation since the kernel is already presented in OpenCL-C. OpenARC is used to perform source-to-source translation from OpenACC and OpenMP to OpenCL-C kernels. Note, coriander does not perform source-to-source translation effectively skipping the kernel representation and compilation stages, it still uses the same version of Clang (from LLVM 3.9.0) to produce the SPIR however this is used from within the tool effectively using clang behind-the-scenes for the compilation stage however the details are hidden from the user and are thus omitted in the diagram. Our hypothesis is that these characteristics should be largely independent of language used to implement it, although it is expected that different compiler optimizations and translation strategies will subtly change the metrics. Part of the goal of this paper is to examine the magnitudes of change imposed by language, compiler and translator. With the similarities between metrics despite the original implementation can be used to evaluate the similarities of application and potential deficiencies in specific compilers.

The translation was largely automatic using both tools to convert between languages. OpenARC doesn't support C++ programs, thus, some benchmarks we examined had to be converted to ANSI C89, and our biggest pain was with user defined `struct`'s needing to be explicitly mentioned when used. Coriander lacks support of `cudaMemcpyToSymbol` function calls and required manually replacing to `cudaMemcpy`, similarly, deprecated calls in the code-base to `cudaThreadSynchronize` also required replacing with `cudaDeviceSynchronize`. The greatest effort in using these tools was in setting them up in the first place, Coriander is selective in versions of LLVM supported, while using both Oclgrind with Coriander raises an interesting LLVM issue. Essentially multiple libraries that are using LLVM, which has some static initialization functions to register passes, end up breaking if they're called multiple times from the same process. To solve this issue we build and link all tools (AIWC and the translators) against a shared version of LLVM. We provide a Docker artefact to alleviate this pain for future users.[2]

5 RESULTS

We now present the results of this comparison over Gaussian Elimination (GE). The GE benchmark was provided by The Rodinia Benchmark Suite [1] 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.

5.1 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**

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

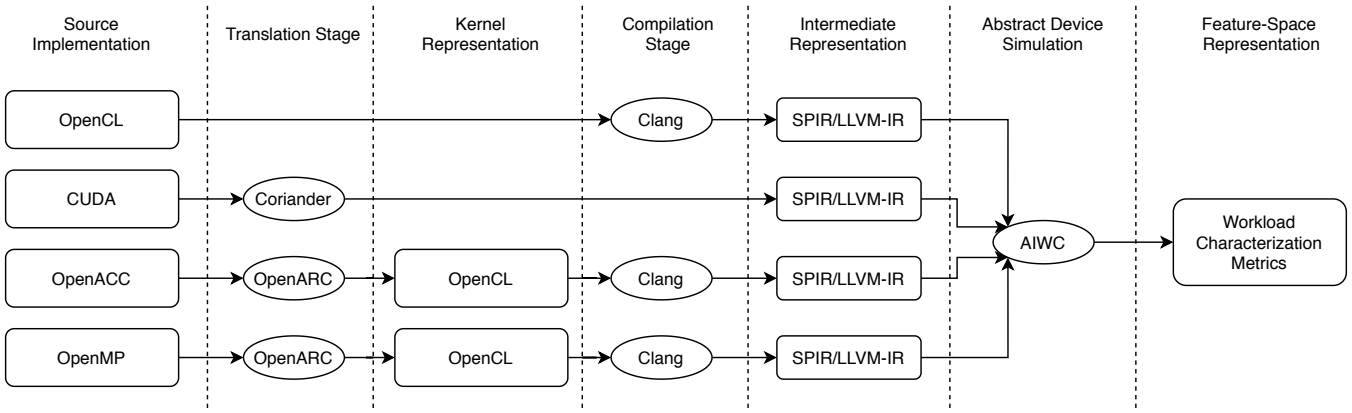

Figure 1: Workflow of using translators and compilers to interoperate with AIWC.

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)`.

The resulting source code of each of the four implementations is listed in Table 1.

The baseline OpenCL kernel code for Fan1 is presented in Listing 1-i, CUDA in , OpenACC in , and OpenMP in.

write summary

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~2.

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 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 [6] 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~2 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.

Figure~3 shows the same comparison of Fan1 implementations with normalization against the baseline OpenCL counts, and is done to show the relative difference between each implementation – allowing a closer inspection of the differences. A flat-line at 0% is ideal since it shows no difference between metrics captured by AIWC, a perfect translation between implementations results in the same instructions being executed, operating on the same sequence of locations in memory, under the same degree of parallelism and identical AIWC metrics will ensue. 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. The implementations have been separated by colour and grouped into metrics for contrast. Firstly, the Opcode diversity metric is the same between all implementations however the number of instructions executed differ – the CUDA translation has 24% more instructions than OpenCL, while OpenACC and OpenMP increase this count by 37%. To understand the reason, we must examine the translated kernels, generated SPIR and the associated traces of each implementation, these are discussed in Sections 5.2, 5.3 and 5.4 respectively. We see all "Memory" (beige) metrics (on the x-axis) do not indicate any difference of any implementations against the OpenCL case – this is good as it ensures that all the same frequency of memory accesses, the type (whether a read or write), the locations and

i: OpenCL

```
__kernel void Fan1(__global float *m, __global
    ↪ float *a, const int size, const int t){
    int gid = get_local_id(0) + get_group_id(0)
        ↪ * get_local_size(0);
    if (gid < size-1-t) {
        m[size*(gid+t+1)+t] = a[size*(gid+t+1)+t]
            ↪  / a[size*t+t];
    }
}
```

ii: CUDA

```
__global__ void Fan1(float *m, float *a, int
    ↪ size, int t)
{
    int gid = threadIdx.x + blockIdx.x *
        ↪ blockDim.x;

    if(gid < size-1-t){
        m[size*(gid+t+1)+t] = a[size*(gid+t+1)+t]
            ↪  / a[size*t+t];
    }
}
```

iii: OpenACC

```
void Fan1(float *m, float *a, int size, int t)
{
    int i;
    #pragma acc kernels loop independent gang(
        ↪ fan1_gangs) worker(fan1_workers)

    for (i=0; i < size-1-t; i++)
        m[size*(i+t+1)+t] = a[size*(i+t+1)+t] / a
            ↪ [size*t+t];
}
```

iv: OpenMP

```
void Fan1(float *m, float *a, int size, int t)
{
    int i;
    #pragma omp target teams distribute parallel
        ↪  for num_teams(fan1_teams)
        ↪ num_threads(fan1_threads)
    for (i=0; i < size-1-t; i++)
        m[size*(i+t+1)+t] = a[size*(i+t+1)+t] / a
            ↪ [size*t+t];
}
```

Listing 1: All source code implementations of the Fan1 kernel separated by language.

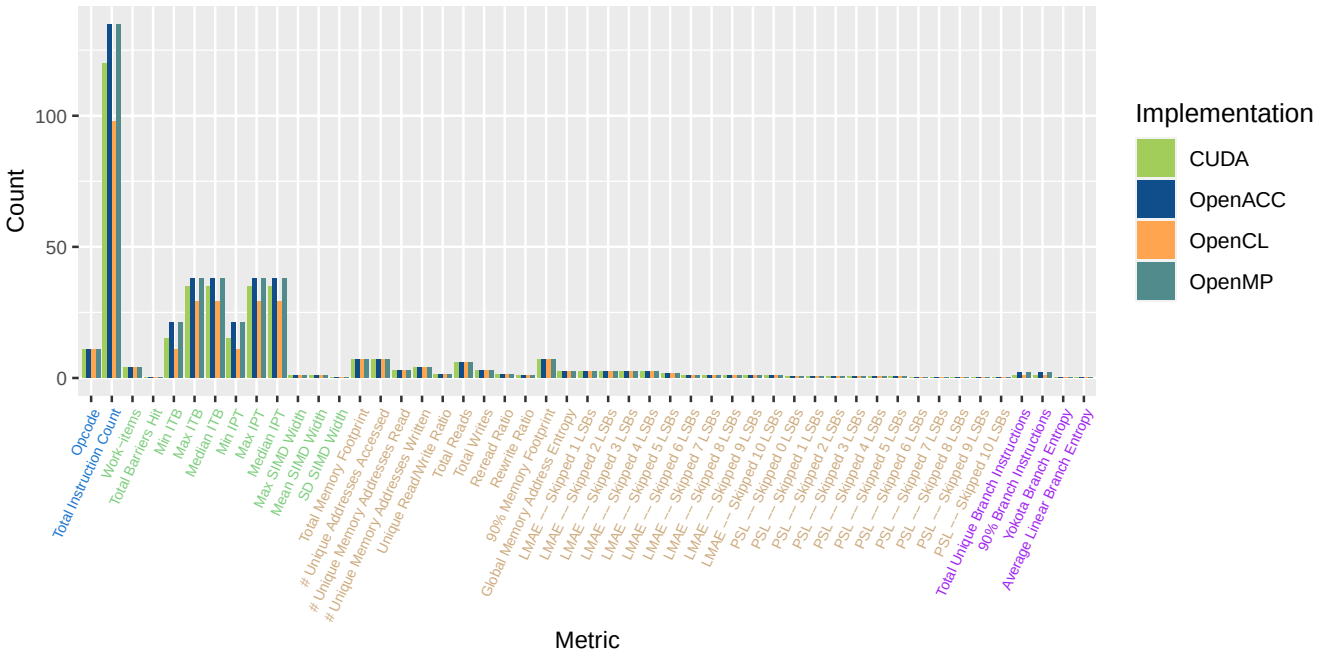

Figure 2: Absolute difference in AIWC metrics between each translated Fan1 kernel implementation.

order of memory accesses are preserved and are equivalent in all implementations, and shows an indistinguishable amount of work has occurred.

The "Total Unique Branch Instructions" and "90% Branch Instructions" are doubled in both the OpenACC and OpenMP versions compared to OpenCL and CUDA. This is because of the absolute doubling in the number of branch instructions the reason becomes apparent when considering the OpenARC generated OpenCL kernels and is discussed during the remainder of this Section.

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

5.2 Kernel Representation

Listing 2 presents the OpenCL kernels generated in the Kernel Representation stage. The workflow from Figure 1 shows how the mix of translators interoperate with AIWC, and justifys why CUDA and OpenCL implementations are excluded from this comparison. Namely, no translation is needed for the OpenCL implementation, while Coriander operating on the CUDA implementation does not generate any kernel representation form and only offers an intermediate representation – which are discussed in Section 5.3. Of the two translated kernels presented in Listing 2-i and 2-ii can be compared directly to the hand-coded OpenCL kernel presented in Listing 1-i – they are the translated OpenARC output of OpenACC and OpenMP (from 1-iii and 1-iv) respectively. We see the pragmas are preserved in the translated output regardless of whether OpenACC or OpenMP are used, however the OpenMP pragma is expressed in terms of OpenACC – the number of threads and number of teams are rewritten as workers and gangs – this is due to an intermediate step in OpenARC which converts OpenMP into OpenACC so it can directly use the OpenACC to OpenCL functionality. Regarding generated OpenCL kernels, both versions are equivilent, sharing the same logic (identical instructions at the same line numbers). Both have the same number of lines in the generated kernels – although the lines in the OpenMP based version are longer because of longer variable names.

When compared to the OpenCL hand-coded version shown in Listing 1-i, both generated kernels have a fundamental difference in structure. There is the same check (in the form of an `if`-statement) to ensure work isn't occuring beyond the defined global boundary – expressed as global work size in OpenCL. However, there is an added `for`-loop than exists in the OpenCL baseline (Listing 1-i).OpenARC expresses all pragma based acceleration as using both local and global workgroups – this makes sense as many kernels use local workgroups to utilise shared memory and ensure good memory access patterns (in the form of cache reuse on many hardware architectures) – but the Fan1 base-line kernel doesn't. This artifact of translation explains many of the differences in the AIWC metrics when comparing the OpenACC and OpenMP to OpenCL and is discussed further in both the Intermediate-Representation analysis, in Section 5.3, and trace analysis, in Section 5.4.

5.3 Intermediate-Representation

A comparison between generated LLVM/SPIR is presented in Listings 3 and 4. Both identify differences in SPIR between Coriander (for CUDA implementations) and OpenARC (OpenACC and OpenMP) compiler outputs against the OpenCL based native version. The similarities between OpenMP and OpenACC implementations of the Fan1 kernel – along with using the same compiler/translator toolchain – means that the generated SPIR are identical and thus consolidated into a single Listing (4-ii).

5.4 Trace Analysis

To examine these differences in actual execution based on the LLVM-IR codes we added the printing of the name of each executed instruction thereby giving a trace of each implementation. This was achieved by adding:

```
if(workItem->getGlobalID()[0]==0){
   printf("%s\n",opcode_name.c_str());
}
```

to the function `instructionExecuted` to AIWC (in `src/plugins/Worklo`
which is triggered as a callback when the Oclgrind simulator executes each instruction. Since oclgrind is a multithreaded program – to the extent that each OpenCL workitem is run on a separate pthread – we only print the log if it occurs on the first thread. The default Gaussian Elimination test data is run on 4 threads and calls the `Fan1` and `Fan2` kernels three (3) times. For this analysis we only store the traces of first execution of the `Fan1` kernel. These traces were then piped from each of the implementations.

The differences between traces are shown in Listing 5. The OpenCL trace is shown in Listing 5-i and presents the baseline progression of instructions expected, ii is the CUDA trace, OpenACC in iii and OpenMP in iv. Each trace should be read as the LLVM instruction executed over time as we proceed down the Listing. Blank lines have been inserted to align common instructions in the trace between implementations, this is to present the clearest difference between traces. Instructions of interest have also been coloured – red indicates added instructions no apparent in the baseline OpenCL trace, blue instructions show a reordering of instructions between traces and olive shows substitution (or deviation) of instructions. The CUDA trace shows that each

tie in instructions added with each memory lookup from the SPIR

. The OpenACC trace has no instruction reordering but has instructions added to componsate for the different control flow (looping) to support the workitems in a workgroup logic – as was described in Section **??**. There is no difference in traces between OpenMP and OpenACC traces because it uses the same OpenARC compiler toolchain.

6 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

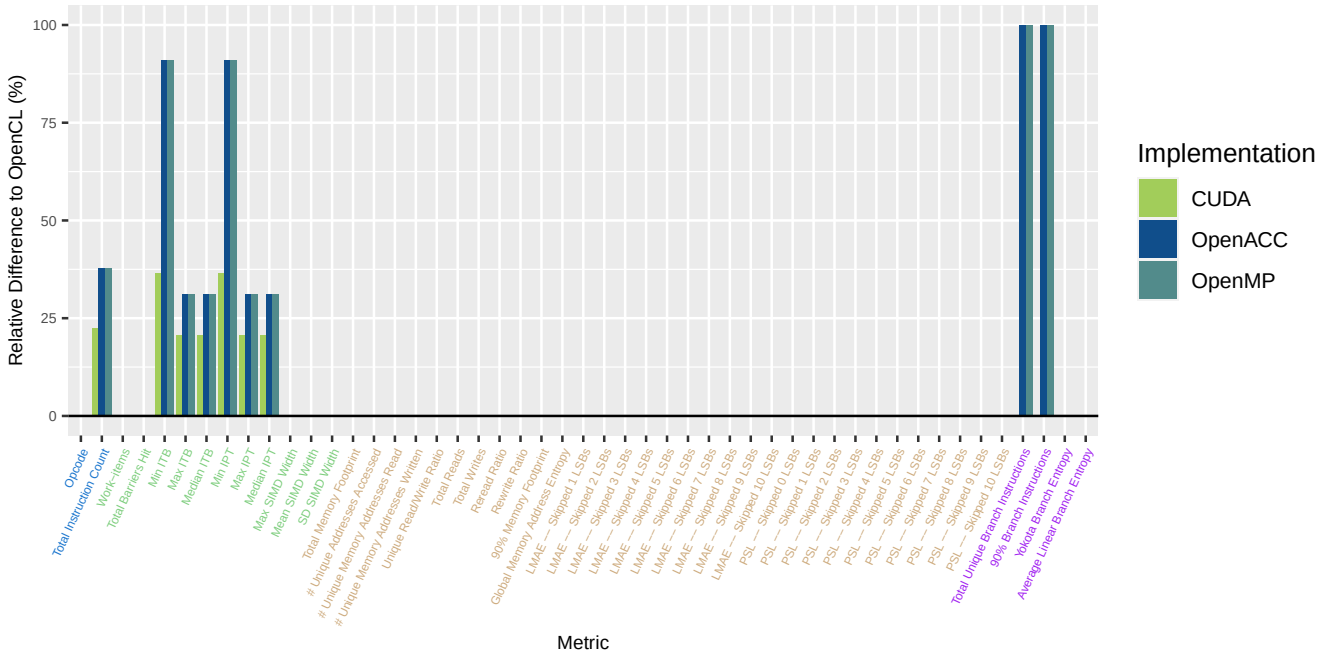

Figure 3: Relative difference in AIWC metrics between each translated Fan1 kernel implementation against the baseline OpenCL.

output from two source code translation tools. We believe this methodology can be boardly more useful in the future development of translators.

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 have found that how they change has enabled us to compare the execution behaviour of codes in response to different compilers performing translation. We have 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, specific accelerators.

i: OpenACC

```
__kernel void Fan1_kernel0(__global float * a,
    ↪ __global float * m, int Size, int
    ↪ fan1_gangs, int fan1_workers, int t)
{
int _ti_100_501;
int lwpriv__i;
_ti_100_501=get_global_id(0);
#pragma acc kernels loop gang(fan1_gangs) worker
    ↪ (fan1_workers) independent copyin(Size, t
    ↪ ) present(a[0:(Size*Size)], m[0:(Size*
    ↪ Size)]) private(i)
if (_ti_100_501<(get_num_groups(0)*fan1_workers)
    ↪ )
{
for (lwpriv__i=(_ti_100_501+0); lwpriv__i<((Size
    ↪ -1)-t); (lwpriv__i+=(get_num_groups(0)*
    ↪ fan1_workers)))
{
m[((Size*((lwpriv__i+t)+1))+t)]=(a[((Size*((
    ↪ lwpriv__i+t)+1))+t)]/a[((Size*t)+t)]);
}
}
}
```

ii: OpenMP

```
__kernel void Fan1_kernel0(int lfpriv__Size, int
    ↪  lfpriv__t, __global float * a, __global
    ↪ float * m, int fan1_teams, int
    ↪ fan1_threads)
{
int _ti_100_501;
int lwpriv__i;
_ti_100_501=get_global_id(0);
#pragma acc parallel loop num_workers(
    ↪ fan1_threads) gang worker present(a[0:(
    ↪ Size*Size)], m[0:(Size*Size)]) private(i)
    ↪  firstprivate(Size, t) num_gangs(
    ↪ fan1_teams)
if (_ti_100_501<(get_num_groups(0)*fan1_threads)
    ↪ )
{
for (lwpriv__i=(_ti_100_501+0); lwpriv__i<((
    ↪ lfpriv__Size-1)-lfpriv__t); (lwpriv__i+=(
    ↪ get_num_groups(0)*fan1_threads)))
{
m[((lfpriv__Size*((lwpriv__i+lfpriv__t)+1))+
    ↪ lfpriv__t)]=(a[((lfpriv__Size*((lwpriv__i
    ↪ +lfpriv__t)+1))+lfpriv__t)]/a[((
    ↪ lfpriv__Size*lfpriv__t)+lfpriv__t)]);
}
}
}
```

Listing 2: OpenCL kernel representation comparison to translator generated OpenACC and OpenMP kernels of Fan1.

REFERENCES

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

[2] Che, S. et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. *Proceedings of the ieee international symposium on workload characterization (iiswc)* (2009).

[3] Dagum, L. and Menon, R. 1998. OpenMP: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*. 1 (1998), 46–55.

[4] Group, K. 2013. Open Compute Language (OpenCL). [Online]. Available: http://www.khronos.org/opencl/.

[5] Inc., P. 2014. Rodinia benchmark modified to run with enzo and pathcu instead of nvcc cuda compiler. https://github.com/pathscale/rodinia.

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

[7] Johnston, B. and Milthorpe, J. 2018. AIWC: OpenCL-based architecture-independent workload characterization. *2018 ieee/acm 5th workshop on the llvm compiler infrastructure in hpc (llvm-hpc)* (2018), 81–91.

[8] Johnston, B. et al. 2018. OpenCL performance prediction using architecture-independent features. *International Workshop on High Performance and Dynamic Reconfigurable Systems and Networks (DRSN-2018) (in press)*. http://www.milthorpe.org/perf-prediction.

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

[10] Nickolls, J. et al. 2008. Scalable parallel programming with cuda. *Queue*. 6, 2 (2008), 40–53.

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

[12] Price, J. and McIntosh-Smith, S. 2015. Oclgrind: An extensible OpenCL device simulator. *Proceedings of the 3rd international workshop on OpenCL* (2015), 12.

i: OpenCL

```
; Function Attrs: nounwind
define void @Fan1(float* nocapture %m_dev, float* nocapture readonly %a_dev, float* nocapture readnone %b_dev, i32 %size, i32 %t) local_unnamed_addr #0 !kernel_arg_addr_space !1 !kernel_arg_access_qual !2 !kernel_arg_type !3 !kernel_arg_base_type !3 !kernel_arg_type_qual !4 {
  %call = tail call i32 (i32, ...) bitcast (i32 (...)* @get_local_id to i32 (i32, ...)*)(i32 0) #3
  %call1 = tail call i32 (i32, ...) bitcast (i32 (...)* @get_group_id to i32 (i32, ...)*)(i32 0) #3
  %call2 = tail call i32 (i32, ...) bitcast (i32 (...)* @get_local_size to i32 (i32, ...)*)(i32 0) #3
  %mul = mul nsw i32 %call2, %call1
  %add = add nsw i32 %mul, %call
  %sub = add nsw i32 %size, -1
  %sub3 = sub i32 %sub, %t
  %cmp = icmp slt i32 %add, %sub3
  br i1 %cmp, label %1, label %4

; <label>:1: ; preds = %0
  %add4 = add i32 %t, 1
  %add5 = add i32 %add4, %add
  %mul6 = mul nsw i32 %add5, %size
  %idx.ext = sext i32 %mul6 to i64
  %add.ptr = getelementptr inbounds float, float* %a_dev, i64 %idx.ext
  %idx.ext7 = sext i32 %t to i64
  %add.ptr8 = getelementptr inbounds float, float* %add.ptr, i64 %idx.ext7
  %2 = load float, float* %add.ptr8, align 4, !tbaa !5
  %mul9 = mul nsw i32 %t, %size
  %idx.ext10 = sext i32 %mul9 to i64
  %add.ptr11 = getelementptr inbounds float, float* %a_dev, i64 %idx.ext10
  %add.ptr13 = getelementptr inbounds float, float* %add.ptr11, i64 %idx.ext7
  %3 = load float, float* %add.ptr13, align 4, !tbaa !5
  %div = fdiv float %2, %3, !fpmath !9
  %add.ptr18 = getelementptr inbounds float, float* %m_dev, i64 %idx.ext
  %add.ptr20 = getelementptr inbounds float, float* %add.ptr18, i64 %idx.ext7
  store float %div, float* %add.ptr20, align 4, !tbaa !5
  br label %4

; <label>:4: ; preds = %1, %0
  ret void
}
```

ii: CUDA

```
; Function Attrs: nounwind
define void @_Z4Fan1PfS_ii(float* nocapture, float* nocapture readonly, i32, i32) local_unnamed_addr #1 {
  %5 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x() #3, !range !5 @thread_id
  %6 = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x() #3, !range !6 @block_id
  %7 = tail call i32 @llvm.nvvm.read.ptx.sreg.ntid.x() #3, !range !7 @block_dim
  %8 = mul i32 %7, %6
  %9 = add i32 %8, %5
  %10 = add nsw i32 %2, -1
  %11 = sub i32 %10, %3
  %12 = icmp slt i32 %9, %11
  br i1 %12, label %13, label %30

; <label>:13: ; preds = %4
  %14 = add i32 %3, 1
  %15 = add i32 %14, %9
  %16 = mul nsw i32 %15, %2
  %17 = sext i32 %16 to i64
  %18 = getelementptr inbounds float, float* %1, i64 %17
  %19 = sext i32 %3 to i64
  %20 = getelementptr inbounds float, float* %18, i64 %19
  %21 = load float, float* %20, align 4, !tbaa !8
  %22 = mul nsw i32 %3, %2
  %23 = sext i32 %22 to i64
  %24 = getelementptr inbounds float, float* %1, i64 %23
  %25 = getelementptr inbounds float, float* %24, i64 %19
  %26 = load float, float* %25, align 4, !tbaa !8
  %27 = fdiv float %21, %26
  %28 = getelementptr inbounds float, float* %0, i64 %17
  %29 = getelementptr inbounds float, float* %28, i64 %19
  store float %27, float* %29, align 4, !tbaa !8
  br label %30

; <label>:30: ; preds = %13, %4
  ret void
}
```

Listing 3: OpenCL compared to the CUDA implementations generated LLVM-IR/SPIR of the Fan1 kernel.

i: OpenCL

```
; Function Attrs: nounwind
define void @Fan1(float* nocapture %m_dev, float* nocapture readonly %a_dev, float* nocapture readnone %b_dev, i32 %size, i32 %t) local_unnamed_addr #0 !kernel_arg_addr_space !1 !kernel_arg_access_qual !2 !kernel_arg_type !3 !kernel_arg_base_type !3 !kernel_arg_type_qual !4 {
  %call = tail call i32 (i32, ...) bitcast (i32 (...)* @get_local_id to i32 (i32, ...)*)(i32 0) #3
  %call1 = tail call i32 (i32, ...) bitcast (i32 (...)* @get_group_id to i32 (i32, ...)*)(i32 0) #3
  %call2 = tail call i32 (i32, ...) bitcast (i32 (...)* @get_local_size to i32 (i32, ...)*)(i32 0) #3
  %mul = mul nsw i32 %call2, %call1
  %add = add nsw i32 %mul, %call
  %sub = add nsw i32 %size, -1
  %sub3 = sub i32 %sub, %t
  %cmp = icmp slt i32 %add, %sub3
  br i1 %cmp, label %1, label %4

; <label>:1: ; preds = %0
  %add4 = add i32 %t, 1
  %add5 = add i32 %add4, %add
  %mul6 = mul nsw i32 %add5, %size
  %idx.ext = sext i32 %mul6 to i64
  %add.ptr = getelementptr inbounds float, float* %a_dev, i64 %idx.ext
  %idx.ext7 = sext i32 %t to i64
  %add.ptr8 = getelementptr inbounds float, float* %add.ptr, i64 %idx.ext7
  %2 = load float, float* %add.ptr8, align 4, !tbaa !5
  %mul9 = mul nsw i32 %t, %size
  %idx.ext10 = sext i32 %mul9 to i64
  %add.ptr11 = getelementptr inbounds float, float* %a_dev, i64 %idx.ext10
  %add.ptr13 = getelementptr inbounds float, float* %add.ptr11, i64 %idx.ext7
  %3 = load float, float* %add.ptr13, align 4, !tbaa !5
  %div = fdiv float %2, %3, !fpmath !9
  %add.ptr18 = getelementptr inbounds float, float* %m_dev, i64 %idx.ext
  %add.ptr20 = getelementptr inbounds float, float* %add.ptr18, i64 %idx.ext7
  store float %div, float* %add.ptr20, align 4, !tbaa !5
  br label %4

; <label>:4: ; preds = %1, %0
  ret void
}
```

ii: OpenACC and OpenMP

```
; Function Attrs: nounwind
define void @Fan1_kernel0(float* nocapture readonly %a, float* nocapture %m, i32 %Size, i32 %fan1_gangs, i32 %fan1_workers, i32 %t) local_unnamed_addr #0 !kernel_arg_addr_space !1 !kernel_arg_access_qual !2 !kernel_arg_type !3 !kernel_arg_base_type !3 !kernel_arg_type_qual !4 {
  %call = tail call i32 (i32, ...) bitcast (i32 (...)* @get_global_id to i32 (i32, ...)*)(i32 0) #3
  %call1 = tail call i32 (i32, ...) bitcast (i32 (...)* @get_num_groups to i32 (i32, ...)*)(i32 0) #3
  %mul = mul nsw i32 %call1, %fan1_workers
  %cmp = icmp slt i32 %call, %mul
  br i1 %cmp, label %.preheader, label %.loopexit

  .preheader: ; preds = %0
  %sub = add nsw i32 %Size, -1
  %sub2 = sub i32 %sub, %t
  %cmp338 = icmp slt i32 %call, %sub2
  br i1 %cmp338, label %.lr.ph, label %.loopexit

  .lr.ph: ; preds = %.preheader
  %add4 = add i32 %t, 1
  %mul8 = mul nsw i32 %t, %Size
  %add9 = add nsw i32 %mul8, %t
  %idxprom10 = sext i32 %add9 to i64
  %arrayidx11 = getelementptr inbounds float, float* %a, i64 %idxprom10
  br label %1

  ; <label>:1: ; preds = %.lr.ph, %1
  %lwpriv__i.039 = phi i32 [ %call, %.lr.ph ], [ %add20, %1 ]
  %add5 = add i32 %add4, %lwpriv__i.039
  %mul6 = mul nsw i32 %add5, %Size
  %add7 = add nsw i32 %mul6, %t
  %idxprom = sext i32 %add7 to i64
  %arrayidx = getelementptr inbounds float, float* %a, i64 %idxprom
  %2 = load float, float* %arrayidx, align 4, !tbaa !5
  %3 = load float, float* %arrayidx11, align 4, !tbaa !5
  %div = fdiv float %2, %3, !fpmath !9
  %arrayidx17 = getelementptr inbounds float, float* %m, i64 %idxprom
  store float %div, float* %arrayidx17, align 4, !tbaa !5
  %call18 = tail call i32 (i32, ...) bitcast (i32 (...)* @get_num_groups to i32 (i32, ...)*)(i32 0) #3
  %mul19 = mul nsw i32 %call18, %fan1_workers
  %add20 = add nsw i32 %mul19, %lwpriv__i.039
  %cmp3 = icmp slt i32 %add20, %sub2
  br i1 %cmp3, label %1, label %.loopexit.loopexit

  .loopexit.loopexit: ; preds = %1
  br label %.loopexit

  .loopexit: ; preds = %.loopexit.loopexit, %.preheader, %0
  ret void
}
```

Listing 4: OpenCL compared to the OpenMP and OpenACC implementations generated LLVM-IR/SPIR of the Fan1 kernel.

i: OpenCL	ii: CUDA	iii: OpenACC	iv: OpenMP
	getelementptr		
	bitcast		
call	call	call	call
	trunc	trunc	trunc
		sext	sext
call	call	call	call
	trunc		
call	call	sext	sext
	trunc		
mul	mul	mul	mul
add	add		
trunc			
add	add		
sub	sub		
icmp	icmp	icmp	icmp
br	br	br	br
	getelementptr		
	bitcast		
add	add	add	add
		sub	sub
add	add	add	add
mul	mul	mul	mul
		add	add
sext	sext	sext	sext
getelementptr		getelementptr	getelementptr
sext	sext	br	br
	getelementptr		
getelementptr	getelementptr	phi	phi
load	load	icmp	icmp
		br	br
		add	add
mul	mul	mul	mul
		add	add
sext	sext	sext	sext
getelementptr	getelementptr	getelementptr	getelementptr
getelementptr	getelementptr	load	load
load	load	load	load
fdiv	fdiv	fdiv	fdiv
getelementptr	getelementptr	getelementptr	getelementptr
getelementptr	getelementptr		
store	store	store	store
		zext	zext
		add	add
		trunc	trunc
		br	br
		phi	phi
		icmp	icmp
		br	br
br	br	br	br
ret	ret	ret	ret

Listing 5: Trace of instructions executed by thread 0 of Fan1 kernel for each of the language implementations.