AIWC: OpenCL based Architecture Independent Workload Characterization

Author details omitted for review

ABSTRACT

OpenCL is an attractive programming model for high performance computing systems composed of heterogeneous compute devices, with wide support from hardware vendors allowing portability of application codes. For accelerator designers and HPC integrators, understanding the performance characteristics of scientific workloads is of utmost importance. However, if these characteristics are tied to architectural features that are specific to a particular system, they may not generalize well to alternative or future systems. A architecture-independent method ensures an accurate characterization of inherent program behavior, without bias due to architectural-dependent features that vary widely between different types of accelerators. This work presents the first architecture-independent workload characterization framework for heterogeneous compute platforms. The tool, AIWC, is capable of characterizing OpenCL workloads currently in use in the supercomputing setting, and is deployed as part of the open source Oclgrind simulator. An evaluation of the metrics collected over a subset of the Extended OpenDwarfs Benchmark Suite is also presented.

1. INTRODUCTION

Cutting-edge high-performance computing (HPC) systems are typically heterogeneous, with a single node comprising a traditional CPU and an accelerator such as a GPU or many-integrated-core device (MIC). High bandwidth, low latency interconnects such as the CRAY XC50 *Aries*, Fujitsu Post-K *Tofu* and IBM Power9 *Bluelink*, support tighter integration between compute devices on a node. Some interconnects support multiple different kinds of devices on a single node, for example *Bluelink* features both NVLINK support for Nvidia GPUs and CAPI for other emerging accelerators such as DSPs, FPGAs and MICs.

The OpenCL programming framework is well-suited to such heterogeneous computing environments, as a single OpenCL code may be executed on multiple different device types. When combined with autotuning, an OpenCL code may exhibit good performance across varied devices. Spafford et al. [16], Chaimov et al. [3] and Nugteren and Codreanu [11] all propose open source libraries capable of performing autotuning of dynamic execution parameters in OpenCL kernels. Additionally, Price and McIntosh-Smith [13] have demonstrated high performance using a general purpose autotuning library [1], for three applications across twelve devices.

Given a diversity of application codes and computational capabilities of accelerators, optimal performance of each code may vary between devices. Hardware designers and HPC integrators would benefit from accurate and systematic performance prediction for combinations of different codes and accelerators, for example, in designing a HPC system, to choose a mix of accelerators that is well-suited to the expected workload.

To this end, we present the Architecture Independent Workload Characterization (AIWC) tool. AIWC simulates execution of OpenCL kernels to collect architecture-independent features which characterize each code, and may also be used in performance prediction. We demonstrate the use of AIWC to characterize a variety of codes in the Extended OpenDwarfs Benchmark Suite [7].

2. RELATED WORK

Oclgrind is an OpenCL device simulator developed by Price and McIntosh-Smith [14] capable of performing simulated kernel execution. It operates on a restricted LLVM IR known as Standard Portable Intermediate Representation (SPIR) [9], thereby simulating OpenCL kernel code in a hardware agnostic manner. This architecture independence allows the tool to uncover many portability issues when migrating OpenCL code between devices. Additionally Oclgrind comes with a set of tools to detect runtime API errors, race conditions and invalid memory accesses, and generate instruction histograms. AIWC is added as a tool to Oclgrind and leverages its ability to simulate OpenCL device execution using LLVM IR codes.

AIWC relies on the selection of the instruction set architecture (ISA)-independent features determined by Shao and Brooks [15], which in turn builds on earlier work in microarchitecture-independent workload characterization. Hoste and Eeckout [6] show that although conventional microarchitecture-dependent characteristics are useful in locating performance bottlenecks [5], they are mis-

leading when used as a basis on which to differentiate benchmark applications. Microarchitecture-independent workload characterization and the associated analysis tool, known as MICA, was proposed to collect metrics to characterize an application independent of particular microarchitectural characteristics. Architecture-dependent characteristics typically include instructions per cycle (IPC) and miss rates – cache, branch misprediction and translation look-aside buffer (TLB) – and are collected from hardware performance counter results, typically PAPI. However, these characteristics fail to distinguish between inherent program behavior and its mapping to specific hardware features, ignoring critical differences between architectures such as pipeline depth and cache size. The MICA framework collects independent features including instruction mix, instruction-level parallelism (ILP), register traffic, working-set size, data stream strides and branch predictability. These feature results are collected using the Pin [10] binary instrumentation tool. In total 47 microarchitecture-independent metrics are used to characterize an application code. To simplify analysis and understanding of the data, the authors combine principal component analysis with a genetic algorithm to select eight metrics which account for approximately 80% of the variance in the data set.

A caveat in the MICA approach is that the results presented are not ISA-independent nor independent from differences in compilers. Additionally since the metrics collected rely heavily on Pin instrumentation, characterization of multi-threaded workloads or accelerators are not supported. As such, it is unsuited to conventional supercomputing workloads which make heavy use of parallelism and accelerators.

Shao and Brooks have since extended the generality of the MICA to be ISA independent. The primary motivation for this work was in evaluating the suitability of benchmark suites when targeted on general purpose accelerator platforms. The proposed framework briefly evaluates eleven SPEC benchmarks and examines 5 ISA-independent features/metrics. Namely, number of opcodes (e.g., add, mul), the value of branch entropy – a measure of the randomness of branch behavior, the value of memory entropy – a metric based on the lack of memory locality when examining accesses, the unique number of static instructions, and the unique number of data addresses.

Related to the paper, Shao also presents a proof of concept implementation (WIICA) which uses an LLVM IR Trace Profiler to generate an execution trace, from which a python script collects the ISA independent metrics. Any results gleaned from WIICA are easily reproducible, the execution trace is generated by manually selecting regions of code built from the LLVM IR Trace Profiler. Unfortunately, use of the tool is non-trivial given the complexity of the tool chain and the nature of dependencies (LLVM 3.4 and Clang 3.4). Additionally, WIICA operates on `C` and `C++` code, which cannot be executed directly on any accelerator device aside from the CPU. Our work extends this implementation to the broader OpenCL setting to collect architecture independent metrics from a hardware-agnostic language – OpenCL.

The branch entropy measure used by Shao and Brooks [15] was initially proposed by Yokota [17] and uses Shannon's information entropy to determine a score of Branch History Entropy. De Pestel, Eyerman and Eeckhout [4] proposed an alternative metric, average linear branch entropy metric, to allow accurate prediction of miss rates across a range of branch predictors. As their metric is more suitable for architecture-independent studies, we adopt it for this work.

Caparrós Cabezas and Stanley-Marbell [2] present a framework for characterizing instruction- and thread-level parallelism, thread parallelism, and data movement, based on cross-compilation to a MIPS-IV simulator of an ideal machine with perfect caches and branch prediction and unlimited functional units. Instruction- and thread-level parallelism are identified through analysis of data dependencies between instructions and basic blocks. The current version of AIWC does not perform dependency analysis for characterizing parallelism, however, we hope to include such metrics in future versions.

3. METRICS

For each OpenCL kernel invocation the Oclgrind simulator **AIWC** tool collects a set of metrics, which are listed in Table 1.

The **Opcode**, **total memory footprint** and **90% memory footprint** measures are simple counts. Likewise, **Total Instruction Count** is the number of instructions achieved during a kernel execution. The **global memory address entropy** is an positive real number that corresponds to the randomness of memory addresses accessed. The **local memory address entropy** is computed as 10 separate values according to increasing number of Least Significant Bits (LSB), or low order bits, omitted in calculation. The number of bits skipped ranges from 1 to 10, and a steeper drop in entropy with increasing number of bits indicates greater spatial locality in the address stream.

Both **unique branch instructions** and the associated **90% branch instructions** are counts indicating the count of logical control flow branches encountered during kernel execution. **Yokota branch entropy** ranges between 0 and 1, and offers an indication of a program's predictability as a floating point entropy value. The **average linear branch entropy** metric is proportional to the miss rate in program execution; $p = 0$ for branches always taken or not-taken but $p = 0.5$ for the most unpredictable control flow. All branch-prediction metrics were computed using a fixed history of 16-element branch strings, each of which is composed of 1-bit branch results (taken/not-taken).

As the OpenCL programming model is targeted at parallel architectures, any workload characterization must consider exploitable parallelism and associated communication and synchronization costs. We characterize thread-level parallelism (TLP) by the number of **work-items** executed by each kernel, which indicates the maximum number of threads that can be executed concurrently.

Work-item communication hinders TLP, and in the OpenCL setting takes the form of either local communication (within a work-group) using local synchronization (barriers) or globally by dividing the kernel and invoking the smaller kernels on

Table 1: **AIWC** tool metrics.

Type	Metric	Description
Compute	opcode	# of unique opcodes required to cover 90% of dynamic instructions
Compute	Total Instruction Count	Total # of instructions executed
Parallelism	Work-items	# of work-items or threads executed
Parallelism	Total Barriers Hit	maximum # of instructions executed until a barrier
Parallelism	Min ITB	minimum # of instructions executed until a barrier
Parallelism	Max ITB	maximum # of instructions executed until a barrier
Parallelism	Median ITB	median # of instructions executed until a barrier
Parallelism	Max SIMD Width	maximum number of data items operated on during an instruction
Parallelism	Mean SIMD Width	mean number of data items operated on during an instruction
Parallelism	SD SIMD Width	standard deviation across the number of data items affected
Memory	Total Memory Footprint	# of unique memory addresses accessed
Memory	90% Memory Footprint	# of unique memory addresses that cover 90% of memory accesses
Memory	Global Memory Address Entropy	measure of the randomness of memory addresses
Memory	Local Memory Address Entropy	measure of the spatial locality of memory addresses
Control	Total Unique Branch Instructions	# unique branch instructions
Control	90% Branch Instructions	# unique branch instructions that cover 90% of branch instructions
Control	Yokota Branch Entropy	branch history entropy using Shannon's information entropy
Control	Average Linear Branch Entropy	branch history entropy score using the average linear branch entropy

the command queue. Both local and global synchronization can be measured in **instructions to barrier** by performing a running tally of instructions executed per work-item until a barrier is encountered under which the count is saved and resets; this count will naturally include the final (implicit) barrier at the end of the kernel. **Min**, **Max** and **Median ITB** are reported to understand synchronization overheads as well as load imbalance, as a large difference between min and max ITB may indicate an irregular workload.

To characterize data parallelism, we examine the number and width of vector operands in the generated LLVM IR, reported as **Max SIMD Width**, **Mean SIMD Width** and **SD SIMD Width**. Some of the other metrics are highly dependent on workload scale, so **work-items** may be used to normalize between different scales. For example, **total memory footprint** can be divided by **work-items** to give the total memory footprint per work-item, which indicates the memory required per processing element.

4. METHODOLOGY – WORKLOAD CHARACTERIZATION BY TOOLING OCLGRIND

AIWC verifies the architecture independent metrics since they are collected on a tool chain and in a language actively executed on a wide range of accelerators – the OpenCL runtime supports execution on CPU, GPU, DSP, FPGA, MIC and ASIC hardware architectures. The intermediate representation of the OpenCL kernel code is a subset of LLVM IR known as SPIR-V – Standard Portable Intermediate Representation. This IR forms a basis for Oclgrind to perform OpenCL device simulation which interprets LLVM IR instructions.

Migrating the metrics presented in the ISA-independent workload characterization paper [15] to the Oclgrind tool offers a accessible, high-accuracy and reproducible method to acquire these AIWC features. Namely:

- Accessibility: since the Oclgrind OpenCL kernel debugging tool is one of the most adopted OpenCL debugging tools freely available to date, having AIWC metric generation included as a Oclgrind plugin allows rapid workload characterization.
- High-Accuracy: evaluating the low level optimized IR does not suffer from a loss of precision since each instruction is instrumented during its execution in the simulator, unlike with the conventional metrics generated by measuring architecture driven events – such as PAPI and MICA analysis.
- Reproducibility: each instruction is instrumented by the AIWC tool during execution, there is no variance in the metric results presented between OpenCL kernel runs.

The caveat with this approach is the overhead imposed by executing full solution HPC codes on a slower simulator device. However, since AIWC metrics do not vary between runs, this is still a shorter execution time than the typical number of iterations required to get a reasonable statistical sample when compared to a MICA or architecture dependent analysis.

5. IMPLEMENTATION

AIWC is implemented as a plugin for Oclgrind, which simulates kernel execution on an ideal compute device. OpenCL kernels are executed in series, and Oclgrind generates notification events which AIWC handles to populate data structures for each workload metric. Once each kernel has completed execution, AIWC performs statistical summaries of the collected metrics by examining these data structures.

The **Opcode** diversity metric updates a counter on an unordered map during each `workItemBegin` event, the type of operation is determined by examining the opcode name using the LLVM Instruction API.

The number of **work-items** is computed by incrementing a global counter – accessible by all work-item threads – once a

`workItemBegin` notification event occurs.

TLP metrics require barrier events to be instrumented within each thread. Instructions To Barrier **ITB** metrics require each thread to increment a local counter once every `instructionExecuted` has occurred, this counter is added to a vector and reset once the work-item encounters a barrier. The **Total Barriers Hit** counter also increments on the same condition. Work-items are executed sequentially within all work-items in a work-group, if a barrier is hit the queue moves onto all other available work-items in a ready state. Collection of the metrics post barrier resumes during the `workItemClearBarrier` event.

ILP **SIMD** metrics examine the size of the result variable provided from the `instructionExecuted` notification, the width is then added to a vector for the statistics to be computed once the kernel execution has completed.

Total Memory Footprint 90% Memory Footprint and Local Memory Address Entropy **LMAE** metrics require the address accessed to be stored during kernel execution and occurs during the `memoryLoad`, `memoryStore`, `memoryAtomicLoad` and `memoryAtomicStore` notifications.

Branch entropy measurements require a check during `instructionExecuted` event on whether the instruction is a branch instruction, if so a flag indicating a branch operation has occurred is set and both LLVM IR labels – which correspond to branch targets – are recorded. On the next `instructionExecuted` the flag is queried and reset while the current instruction label is examined and is stored around which of the two targets were taken. The branch metrics can then be computed. The **Total Unique Branch Instructions** is a count of the absolute number of unique locations that branching occurred, while the **90% Branch Instructions** indicates the number of unique branch locations that cover 90% of all branches. **Yokota** from Shao [15], and **Average Linear Branch Entropy**, from De Pestel [4] and have been computed and are also presented based on this implementation. `workGroupComplete` events trigger the collection of the intermediate work-item and work-group counter variables to be added to the global suite, while `workGroupBegin` events reset all the local/intermediate counters.

Finally, `kernelBegin` initializes the global counters and `kernelEnd` triggers the generation and presentation of all the statistics listed in Table 1. The source code is available at the GitHub Repository [8].

6. DEMONSTRATION

We now demonstrate the use of **AIWC** with a few example scientific application kernels selected from the Extended OpenDwarfs Benchmark Suite [7]. These benchmarks were extracted from and are representative of general scientific application codes. Our selection is not intended to be exhaustive, rather, it is meant to illustrate how key properties of the codes are reflected in the metrics collected by **AIWC**.

AIWC is run on full application codes, but it is difficult to present an entire summary due to the nature of OpenCL. Computationally intensive kernels are simply selected regions of the full application codes and are invoked separately for device execution. As such, the AIWC metrics can either be shown per kernel run on a device, for all kernel runs on the device, or as the summation of all metrics for a kernel for a full application at a given problem size – we chose the latter. Additionally, given the number of kernels presented we believe AIWC will generalise to full codes in other domains.

We present metrics for 11 different application codes – which includes 37 kernels in total. Each code was run with four different problem sizes, called **tiny**, **small**, **medium** and **large** in the Extended OpenDwarfs Benchmark Suite; these correspond respectively to problems that would fit in the L1, L2 and L3 cache or main memory of a typical current-generation CPU architecture. As simulation within Oclgrind is deterministic, all results presented are for a single run for each combination of code and problem size.

In a cursory breakdown 4 selected metrics are presented in Figure~1. Each of the 4 metrics were chosen as one of each of the main categories, namely, Opcode, Barriers Per Instruction, Global Memory Address Entropy, Branch Entropy (Linear Average). Each category has also been segmented by colour: blue results represent *compute* metrics, green represent metrics that indicate *parallelism*, beige represents *memory* metrics and purple bars represent *control* metrics. Median results are presented for each metric – while there is no variation between invocations of AIWC, certain kernels are iterated multiple times and over differing domains / data sets. Each of the 4 sub-figures shows all kernels over the over 4 different sized problems.

All kernels are presented along the x-axis, whereas the normalized percentage of each category is presented in the y-axis.

Generally, problem size primarily affects the memory category – where global memory address entropy increases with problem size – while the generic shape of the other categories is fixed per kernel. Note, there are fewer kernels presented over the **medium** and **large** problem sizes due to the difficulties in determining arguments and the ability of some benchmark applications to run correctly over the larger problem sizes.

Following Shao and Brooks [15], we present the AIWC metrics for a kernel as a kiviat or radar diagram, for each of the problem sizes. Unlike Shao and Brooks, we do not perform any dimensionality reduction, but choose to present all collected metrics. The ordering of the individual spokes is not chosen to reflect any statistical relation between the metrics, however, they have been grouped into four main categories: green spokes represent metrics that indicate *parallelism*, blue spokes represent *compute* metrics, beige spokes represent *memory* metrics and purple spokes represent *control* metrics. For clarity of visualization, we do not present the raw AIWC metrics, but instead normalize or invert the metrics to produce a scale from 0 to 1. The parallelism metrics presented are the inverse values of the metrics collected by AIWC, i.e. **granularity** = 1/**work-items** ; **barriers per instruction** = 1/**mean ITB** ; **instructions per operand** = $1/\sum$ **SIMD widths**. All other values are normalized according to the maximum value measured across all kernels

Figure 1: Selected AIWC metrics from each category over all kernels and 4 problem sizes.

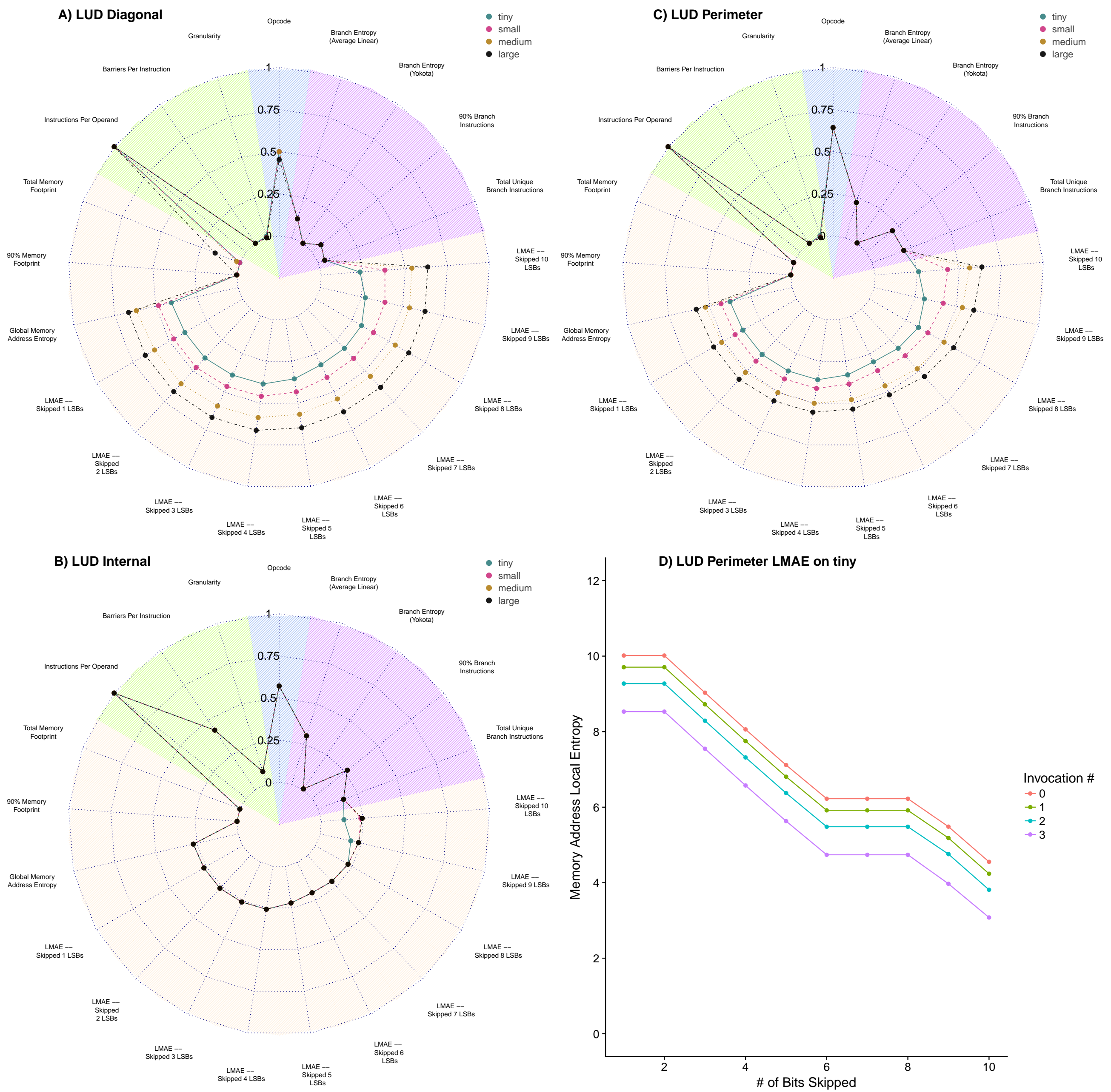

Figure 2: A) B) and C) show the AIWC features of the `diagonal`, `internal` and `perimeter` kernel of the LUD application over all problem sizes. D) shows the corresponding Local Memory Address Entropy for the `perimeter` kernel over the tiny problem size.

examined – and on all problem sizes. This presentation allows a quick value judgement between kernels, as values closer to the center (0) generally have lower hardware requirements, for example, smaller entropy scores indicate more regular memory access or branch patterns, requiring less cache or branch predictor hardware; smaller granularity indicates higher exploitable parallelism; smaller barriers per instruction indicates less synchronization; and so on.

The **lud** benchmark application comprises three major kernels, **diagonal**, **internal** and **perimeter**, corresponding to updates on different parts of the matrix. The AIWC metrics for each of these kernels are presented – superimposed over all problem sizes – in Figure~2 A) B) and C) respectively. Comparing the kernels, it is apparent that the diagonal and perimeter kernels have a large number of branch instructions with high branch entropy, whereas the internal kernel has few branch instructions and low entropy. This is borne out through inspection of the OpenCL source code: the internal kernel is a single loop with fixed bounds, whereas diagonal and perimeter kernels contain doubly-nested loops over triangular bounds and branches which depend on thread id. Comparing between problem sizes (moving across the page), the large problem size shows higher values than the tiny problem size for all of the memory metrics, with little change in any of the values.

The visual representation provided from the kiviat diagrams allows the characteristics of OpenCL kernels to be readily assessed and compared.

Finally, we examine the linear memory access entropy (LMAE) presented in the kiviat diagrams in greater detail. Figure~2 D) presents a sample of the linear memory access entropy, in this instance of the LUD Perimeter kernel collected over the tiny problem size. The kernel is launched 4 separate times during a run of the tiny problem size, this is application specific and in this instance each successive invocation operates on a smaller data set per iteration. Note there is steady decrease in starting entropy, and each successive invocation of the LU Decomposition Perimeter kernel the lowers the starting entropy. However the descent in entropy – which corresponds to more bits being skipped, or bigger the strides or the more localized the memory access – shows that the memory access patterns are the same regardless of actual problem size.

7. CONCLUSIONS AND FUTURE WORK

We have presented the Architecture-Independent Workload Characterization tool (AIWC), which supports the collection of architecture-independent features of OpenCL application kernels. These features can be used to predict the most suitable device for a particular kernel, or to determine the limiting factors for performance on a particular device, allowing OpenCL developers to try alternative implementations of a program for the available accelerators – for instance, by reorganizing branches, eliminating intermediate variables et cetera. The additional architecture independent characteristics of a scientific workload will be beneficial to both accelerator designers and computer engineers responsible for ensuring a suitable accelerator diversity for scientific codes on supercomputer nodes.

Caparrós Cabezas and Stanley-Marbell [2] examine the Berkeley dwarf taxonomy by measuring instruction-level parallelism, thread parallelism, and data movement. They propose a sophisticated metric to assess ILP by examining the data dependency graph of the instruction stream. Similarly, Thread-Level-Parallelism was measured by analysing the block dependency graph. Whilst we propose alternative metrics to evaluate ILP and TLP – using the max, mean and standard deviation statistics of SIMD width and the total barriers hit and Instructions To Barrier metrics respectively – a quantitative evaluation of the dwarf taxonomy using these metrics is left as future work. We expect that the additional AIWC metrics will generate a comprehensive feature-space representation which will permit cluster analysis and comparison with the dwarf taxonomy.

References

[1] Ansel, J. et al. 2014. OpenTuner: An extensible framework for program autotuning. *International conference on parallel architectures and compilation techniques (PACT)* (Edmonton, Canada, August 2014).

[2] Caparrós Cabezas, V. and Stanley-Marbell, P. 2011. Parallelism and data movement characterization of contemporary application classes. *Proceedings of the twenty-third annual ACM symposium on parallelism in algorithms and architectures* (New York, NY, USA, 2011), 95–104.

[3] Chaimov, N. et al. 2014. Toward multi-target autotuning for accelerators. *IEEE international conference on parallel and distributed systems (ICPADS)* (2014), 534–541.

[4] De Pestel, S. et al. 2017. Linear branch entropy: Characterizing and optimizing branch behavior in a micro-architecture independent way. *IEEE Transactions on Computers.* 66, 3 (Mar. 2017), 458–472. DOI:https://doi.org/10.1109/TC.2016.2601323.

[5] Ganesan, K. et al. 2008. A performance counter based workload characterization on Blue Gene/P. *International conference on parallel processing (ICPP)* (2008), 330–337.

[6] Hoste, K. and Eeckhout, L. 2007. Microarchitecture-independent workload characterization. *IEEE Micro.* 27, 3 (2007).

[7] Johnston, B. and Milthorpe, J. 2017. Dwarfs on accelerators: Extending OpenCL benchmarking for heterogeneous computing architectures. *unpublished.* (2017).

[8] Johnston, B. et al. 2017. BeauJoh/Oclgrind: Adding AIWC – An Architecture Independent Workload Characterisation Plugin. https://doi.org/10.5281/zenodo.1134175.

[9] Kessenich, J. 2015. A Khronos-Defined Intermediate Language for Native Representation of Graphical Shaders and Compute Kernels.

[10] Luk, C.-K. et al. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN notices* (2005), 190–200.

[11] Nugteren, C. and Codreanu, V. 2015. CLTune: A

generic auto-tuner for OpenCL kernels. *IEEE international symposium on embedded multicore/many-core systems-on-chip (MCSoC)* (2015), 195–202.

[12] Prakash, T.K. and Peng, L. 2008. Performance characterization of SPEC CPU2006 benchmarks on Intel Core 2 Duo processor. *ISAST Trans. Comput. Softw. Eng.* 2, 1 (2008), 36–41.

[13] Price, J. and McIntosh-Smith, S. 2017. Analyzing and improving performance portability of OpenCL applications via auto-tuning. *Proceedings of the 5th international workshop on OpenCL* (2017), 14.

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

[15] Shao, Y.S. and Brooks, D. 2013. ISA-independent workload characterization and its implications for specialized architectures. *IEEE international symposium on performance analysis of systems and software (ISPASS)* (2013), 245–255.

[16] Spafford, K. et al. 2010. Maestro: Data orchestration and tuning for OpenCL devices. *Euro-Par 2010-Parallel Processing.* (2010), 275–286.

[17] Yokota, T. et al. 2007. Introducing entropies for representing program behavior and branch predictor performance. *Proceedings of the 2007 workshop on experimental computer science* (2007), 17.