Characterizing Optimizations to Memory Access Patterns using Architecture-Independent Program Features

Aditya Chilukuri
aditya.chilukuri@anu.edu.au
Australian National University

Josh Milthorpe
josh.milthorpe@anu.edu.au
Australian National University

Beau Johnston
beau.johnston@anu.edu.au
Australian National University

ABSTRACT

HPC developers are faced with the challenge of optimizing OpenCL workloads for high performance on novel architectures. The Architecture Independent Workload Characterisation (AIWC) tool is a plugin for the Oclgrind OpenCL simulator that gathers metrics of OpenCL programs that can be used to understand and predict program performance on an arbitrary given hardware architecture. However, AIWC metrics are not always easily interpreted and do not reflect some important memory access patterns affecting efficiency across architectures. We propose the concept of parallel spatial locality – the closeness of memory accesses simultaneously issued by OpenCL work-items (threads). We implement the parallel spatial locality metric in the AIWC framework, and analyse gathered results on matrix multiply and the Extended OpenDwarfs OpenCL benchmarks. The differences in the obtained parallel spatial locality metric across implementations of matrix multiply reflect the optimizations performed. The new metric can be used to distinguish between the OpenDwarfs benchmarks based on the memory access patterns affecting their performance on various architectures. The improvements suggested to AIWC will help HPC developers better understand memory access patterns of complex codes, a crucial step in guiding optimization of codes for arbitrary hardware targets.

KEYWORDS

Architecture Independent Analysis, Heterogenous Computing, Workload Characterization, Memory Access Patterns

IWOCL'20, April 27–29, 2020, Munich, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Aditya Chilukuri, Josh Milthorpe, and Beau Johnston. 2020. Characterizing Optimizations to Memory Access Patterns using Architecture-Independent Program Features. In *Proceedings of International Workshop on OpenCL (IWOCL'20).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

High-performance computing (HPC) systems are increasingly heterogeneous. A single node on a modern supercomputer may combine traditional CPUs with one or more accelerators such as GPUs, Field Programmable Gate Arrays (FPGAs), or many integrated core devices (MICs). High bandwidth interconnects support tight integration between multiple devices of different types on a single compute node.

The OpenCL programming language is designed to support modern HPC software engineers in writing code that executes on multiple hardware targets. This gives HPC software developers greater flexibility by allowing codes aimed at a range of hardware targets to be written in a single programming language environment.

Application codes differ in resource requirements, control structure and available parallelism. Similarly, compute devices differ in number and capabilities of execution units, processing model, and available resources. These heterogeneous computing environments present opportunities for software engineers and HPC integrators to design highly optimized systems with multiple kernels executing on hardware targets best suited for the diverse computational tasks performed by each kernel [16]. However, this opportunity also presents a challenge in optimizing code to run on diverse architectures. The work reported here aims to help HPC developers understand and optimize memory access patterns to improve performance on heterogeneous systems.

In the evolution of computer architectures over the last few decades, the exponential growth in computational capability has not been matched by proportional increases in memory speeds[3]. Memory accesses pose larger bottlenecks to performance as application demand for main memory scales with the arithmetic capability of computer systems. To mitigate this latency, modern CPU designs have employed a wide range of cache technologies to reduce main memory accesses

using the principle of *spatial locality*: the observation that data that a program accesses close together in time tend to also be close together in memory. On the other hand, GPUs rely on hardware multithreading to hide memory latency, and their architectures favour ALU capability over sophisticated logic to manage a cache hierarchy and out-of-order execution. As a result, the performance of kernels on CPUs and GPUs alike is strongly dependent on memory access patterns intrinsic to the code.

Our aim in this paper is to develop a framework to guide HPC software engineers in hardware-dependent code optimization – specifically by guiding the improvement of memory access patterns. We provide examples of manufacturer-recommended code optimizations to improve memory access patterns on the target architecture, and examine these using the architecture-independent workload characterization (AIWC) [7] plugin for Oclgrind. Our work highlights the benefits and challenges arising from an architecture-independent analysis of memory-based program characteristics. We propose two new metrics to AIWC to better characterize these memory-based optimizations. We measure the presented codes using the new metrics and demonstrate the metrics' effectiveness in capturing the essence of the optimizations performed.

The structure of this paper is as follows. In section 2 we use the example of matrix multiplication on a GPU to showcase the specific vendor-recommended optimizations our work aims to measure. In section 3 we discuss how AIWC and its precursors profiled memory access patterns in an architecture-independent fashion, and relevant architecture dependent approaches to memory access profiling. In section 4 we present a methodology to critically examine the architecture-independent workload characterization tool, which we use in section 5 to measure the accuracy of AIWC in profiling the impact of performance optimizations to various OpenCL codes. In section 6 we discuss these preliminary results, and propose and implement our parallel spatial locality metric into the AIWC framework. In section 7 we present the collected metric for the matrix multiplication kernels and selected OpenDwarfs benchmarks to validate our methodology in this paper. Finally, in section 8 we discuss further avenues to extend our work and conclude.

2 MOTIVATING EXAMPLE

We aim to capture the essence of vendor-recommended optimizations for target architectures in our metrics. We demonstrate the optimization strategies using an OpenCL kernel which multiplies square matrices of order N. We start with a simple unoptimized kernel, and then improve the memory access patterns of this kernel by incrementally performing the optimizations recommended in the CUDA Optimization Handbook [13] for NVIDIA GPUs. These incrementally optimized OpenCL codes are then analysed using AIWC to determine the accuracy in profiling favourable memory access patterns of the current framework and the proposed extensions to AIWC.

Simple Unoptimized Matrix Multiplication

The unoptimized matrix multiplication kernel presented in Appendix A is used as a baseline to validate the performance improvements measured from each following optimization. Each thread of the kernel updates the (*globalRow*, *globalCol*) element of matrix C by computing the dot product of the corresponding row of A and column of B.

Using Shared Memory to Coalesce Global Memory Access to Matrix A

We first notice that the number of global memory accesses to matrices A and B increases as $O(N^3)$ with respect to matrix size N. Global memory is typically located off-chip and accesses induce large delays. NVIDIA GPUs coalesce global memory loads and stores issued within thread-groups into as few DRAM transactions as possible. Multiple global memory loads and stores are coalesced into a single transaction when certain device-specific conditions are met. On most NVIDIA GPUs, data accesses are coalesced when multiple requests are made for memory locations from the same cache line in global memory [14]. Appendix B contains the `coalescedAMultiply` kernel, which coalesces accesses to matrix A by storing *tiles*, or square blocks, of A's values into shared memory (OpenCL `__local` memory).

Using Shared Memory to store Tiles of Matrix B

The code is further optimized by improving how the previous `coalescedAMultiply` kernel reads from Matrix B. While each thread in coalescedAMultiply reads only a single element of matrix A from global memory, each thread reads a full column of matrix B. The repeated reads of elements of matrix B across multiple threads can be alleviated by reading tiles from matrix B into shared memory. Appendix C contains the `coalescedAB` kernel which performs this optimization.

Optimizing handling of shared memory

NVIDIA shared memory is divided into multiple banks – stored in independent memory modules – to allow parallel accesses to shared memory. Bank conflicts occur when shared memory in the same bank is accessed concurrently. The code in `coalescedAB` kernel is further optimized by implicitly transposing tiles of Matrix A while loading from global memory. This improves memory bank utilization during reads to shared memory. The `coalescedABT` kernel demonstrates this optimization by modifying lines 20,21 of `coalescedAB` to:

```
20      ASub[localCol][localRow] = A[tiledRow];
21      BSub[localRow][localCol] = B[tiledCol];
```

Alignment of Memory Allocation

Memory access alignment is important to best utilize all parts of GPU memory architecture. Global memory buffer alignment can allow threads to access blocks of global memory aligned to the nearest cache line. This enables coalescing of memory accesses. If buffers are misaligned, parallel memory requests may cross over cache lines, and may double the number of slow global memory accesses needed as demonstrated in figures 1 and 2.

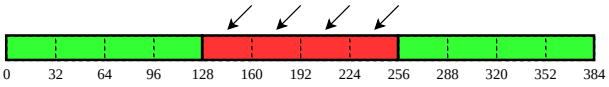

Figure 1: All threads access memory aligned to nearest cache line in parallel [13]

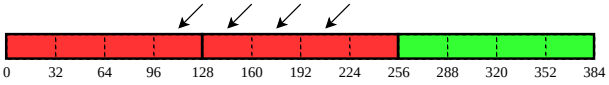

Figure 2: Unaligned sequential memory addresses fit in two cache lines [13]

A similar principle applies when using shared memory. Bank conflicts may be reduced by aligning allocations of shared memory buffers. The `alignedABT` kernel improves the alignment of shared memory tiles in `coalescedABT`. An arbitrary large alignment value of 4096 is chosen as it is larger than the cache line size of modern hardware and attributed to the local array declarations in the existing code as shown below.

```
1       __local float aTile[TILE_DIM][TILE_DIM]
            __attribute__ ((aligned(4096)));
2       __local float bTile[TILE_DIM][TILE_DIM]
            __attribute__ ((aligned(4096)));
```

All the examples above are an incomplete listing of optimization strategies developers can apply to code targeted at NVIDIA GPUs. In section 7, we will compare the collected AIWC metrics with the expected effect of each optimization, to examine how effectively our methodology uncovers underlying bottlenecks and guides optimization efforts.

3 RELATED WORK

Hoste and Eeckout [4] show that while conventional architecture dependent characteristics are useful in locating performance bottlenecks, they can hide the underlying, inherent program behaviour causing performance bottlenecks. A conceptual understanding of the performance characteristics of complex codes is necessary for the programmer to effectively optimize these codes. Architecture dependent characteristics typically include instructions per cycle (IPC), cache and branch prediction miss rates, page faults and DRAM bus data transfer rates. These are typically collected using hardware performance counters available on most target architectures. These performance counters do not serve to guide optimization beyond highlighting potential bottlenecks [2, 4]. Further, in many cases, architecture dependent characteristics usually cannot be directly correlated to specific code patterns. For example, the causes of high cache miss rates in the execution of a program are too complex and widely variant on microarchitecture specific features such as cache size, prefetch behaviour and cache placement policies. In summary, we argue that a HPC developer tasked with optimizing code for a given hardware target would benefit from architecture-independent metrics of the code that can be conceptually correlated to hardware specific optimization patterns. Such architecture-independent metrics will effectively guide the HPC developer's workflow in both finding and fixing bottlenecks in code performance.

The Architecture-Independent Workload Characterization (AIWC) tool [7] relies on a set of instruction set architecture (ISA)-independent features determined by Shao and Brooks [15]. AIWC measures memory based metrics to characterize the target code's memory behaviours. AIWC first collects aggregate simple counts such as the (i) total memory footprint, the total number of unique addresses accessed; and (ii) 90% memory footprint, the number of unique addresses covering 90% of memory accesses. A small ratio of 90% memory footprint to total memory footprint correlates to programs accessing the same small subset of memory addresses repeatedly – which can be linked to increased performance in a cached memory hierarchy.

AIWC also records the global memory address entropy (GMAE), a positive real number corresponding to the randomness of the memory access distribution of a program. To measure locality of memory accesses, AIWC collects the local memory address entropy (LMAE) of memory addresses accessed after dropping n least significant bits of all memory addresses accessed by the program. To calculate this, AIWC collects a frequency distribution of all non-register memory accesses by all threads in the target kernel. Using the collected frequency distribution, AIWC calculates 10 separate local memory address entropy (LMAE) values according

to increasing number of least significant bits (LSB) skipped using the explicit formula for the n-bits skipped LMAE:

$$LMAE_{n-bits} = \sum_{a \in A_n} p_a \log_2(p_a^{-1}) \quad (1)$$

- A_n is the set of all addresses accessed after skipping n LSBs of each address.
- $p_a := \frac{\#access_a}{\#access_{total}}$ is the probability (calculated as relative frequency) at which each memory address is accessed.

LMAE measures the locality of memory accesses performed over the full execution of a program. A steeper drop in entropy with increasing number of bits may correlate to more localized memory accesses over the program's execution.

Limitations of the memory footprint and local memory address entropy values as memory access pattern metrics are further discussed by this work.

Advancements in Memory Access Pattern Characterization

We provide a brief summary of past work in architecture independent program characterization and Memory Access Pattern analysis:

- The Microarchitecture Independent Workload Characterization Tool (MICA) [4] was the first microarchitecture independent workload characterization tool for single threaded programs.
- The Workload ISA-Independent Characterization for Applications (WIICA) [15] extends MICA to present a framework to analyse single threaded programs independent of the ISA.
- CuMAPz: CuMAPz [10] is a CUDA memory access profiling tool and the first to present metrics for CUDA codes which can be generalized to any given NVIDIA GPU.

MICA

The MICA framework was developed by Hoste and Eeckout [4] in response to their findings on the limitations of conventional microarchitecture dependent characteristics. Hoste and Eeckout discovered that performance counter based approaches to profiling codes often struggled in finding underlying program features that map to improved or worsened usage of performance critical hardware features of the target architecture. The MICA framework is a holistic characterization tool, and thus collects features including instruction mix, instruction-level parallelism, register traffic, data stream strides and branch predictability.

Of these metrics, data stream strides are of particular interest in memory access pattern profiling. MICA's stride length metric measures the distance between consecutive memory accesses in a single threaded application. For CPU architectures running single threaded applications, this metric correlates to the spatial locality of memory accesses – a measure of how closely bunched are memory access in nearby times. This is directly correlated to cache reuse rates, critical to code performance on CPUs [5].

The MICA approach was tailored for single-threaded applications as the metrics collected rely heavily on Pin instrumentation [12]. As such, MICA is unsuited to analysing conventional supercomputing workloads with heavy use of parallelism.

CuMAPz

Kim and Shrivastava [10] present CuMAPz, a CUDA memory access pattern analysis tool to guide NVIDIA GPU optimizations. CuMAPz focuses on the problem of improving CUDA application performance using NVIDIA memory-based optimizations. CuMAPz analyses CUDA codes structurally and simulates code execution on the memory hierarchy of specific NVIDIA GPU models.

During the simulated run of the target code, CuMAPz records all memory accesses in various buffers to simulate the global and shared memories on NVIDIA GPUs. Using this detailed simulation data, CuMAPz can estimate the performance critical (i) shared memory data reuse profit, (ii) profit from coalesced access (iii) memory channel skew cost and (iv) bank conflict cost characteristics of the target code.

The simulation environment described in CuMAPz and the attached analysis framework is highly specific to CUDA enabled GPUs. Replicating the CuMAPz framework for all target architectures is challenging. However, CuMAPz is an interesting simulator from the standpoint of this paper's work since it adheres to the core parts of NVIDIA GPUs' memory models in its analysis, while allowing the user to specify their GPU model specific hardware information.

Our approach is an architecture independent analysis of memory access patterns to provide metrics correlating to similar performance critical memory access optimizations as CuMAPz. We aim to further the state of the art by providing a framework to guide developers in optimizing OpenCL codes for any given target architecture. To the best our knowledge, none of the previous works present a set of performance metrics that accurately characterize memory access patterns of parallel applications independent of the target architecture.

4 METHODOLOGY

To provide memory access pattern metrics that represent vendor recommended optimizations on GPUs, we improved AIWC by adding our first new metric: relative local memory usage. This measures the proportion of all memory accesses from the symbolic execution of the kernel that occurred

to memory allocated as `__local`. On NVIDIA GPUs, this memory address space is mapped to fast on-chip shared memory.

Relative local memory usage is an example of a metric that is useful to measure performance critical access patterns on some architectures such as GPUs, and not others, such as CPUs. CPUs do not typically have a notion of user controlled on-chip memory shared between hardware threads such as NVIDIA GPUs' shared memory. This is a natural consequence of programming for a heterogeneous system. Specific code patterns may translate to performance improvements only on certain hardware.

In the original AIWC tool [9], global and local memory address entropy (MAE) were calculated using physical addresses of memory used by the Oclgrind simulator back-end of AIWC. This caused irregularities in entropy calculations across multiple runs of the same simulation. We improved the calculation of memory address entropy by using *virtual addresses* to calculate MAE values using an abstract ideal address space on which all memory accesses by the kernel occur. This allows AIWC to accurately abstract over the hardware and ISA-specific differences in memory layouts across the diverse hardware targets.

5 PRELIMINARY RESULTS

Figure 3 shows that across varying problem sizes, the vendor recommended optimizations to the matrix multiplication code lead to increased performance.

Table 1 summarizes AIWC memory metrics collected for each of the matrix multiply kernels in section 2. Note that only two LMAE values are shown for brevity. We now analyse the effectiveness of AIWC metrics in profiling the NVIDIA recommended memory optimizations applied to the matrix multiplication kernel.

Relative local memory usage: As reliance on NVIDIA GPUs' shared memory increases in each kernel from `simpleMultiply` to `coalescedAB`, we find that the ratio of OpenCL local memory usage increases as expected.

Global and local MAE: Entropy measurements decrease from `simpleMultiply` to `coalescedAB`. This is explained by the optimizations performed incrementally reducing accesses to matrix A and B, and instead performing the bulk of computations on smaller tiles in local memory. We observe an almost completely uniform distribution of memory accesses in `simpleMultiply`, where the program makes N loads to each element of A and B with N the dimension of the matrices. The frequency distribution of memory addresses accessed become increasingly non-uniform as we perform less accesses to matrices A and B and offset this by increasing the frequency of accesses to smaller, more localized local memory buffers. The non-uniformity of relative frequency with which each memory cell is accessed by the program translates directly to decreases in local and global memory entropy values as reliance on small local memory buffers increases from `simpleMultiply` to `coalescedAB`.

Memory Footprint: Similar to trends in global and local MAE, we find that the ratio of 90% memory footprint to total memory footprint decreases from 60.12% for `simpleMultiply` to 0.25% (`coalescedAB`). Increased utilization of local memory in the optimized kernels means that the local memory buffers make up a greater proportion of total memory accesses in the program. As the local memory buffers are small and reused within a workgroup, the memory footprint of the local memory buffers is also smaller.

AIWC's metrics strongly reward optimizations that tend to localize memory accesses. Local memory buffers are typically smaller than global memory arrays when programming for GPUs due to hardware limitations on sizes of shared memory [14]. However, the metrics currently measured by AIWC do not have a direct causal relation to code patterns that optimize memory accesses on GPUs. The proposed relative local memory usage metric is the first to correspond to a recommended optimization strategy of using fast on-chip shared memory. Further, we find that all current AIWC metrics do not measure any sizeable difference between `coalescedAB`, `coalescedABT` and `alignedABT` codes. We address the root cause of this – a lack of temporal information about memory accesses, in the next section.

6 DISCUSSION

AIWC metrics are inherently lossy. Given the set of metrics calculated for an unknown kernel, we can not reconstruct the kernel based on these metrics. Specifically to memory access pattern analysis, the calculation of memory address entropy of a kernel relies only on the frequency distribution of memory accesses to all addresses accessed by the kernel. We observe that calculation of memory address entropy does not make use of temporal information. The order of sequential memory accesses performed by each thread, as well as addresses requested in parallel by multiple threads in an OpenCL thread group, are both vital in accurately characterizing parallel codes.

We propose a new architecture independent metric to measure memory access patterns of parallel programs. The proposed metric is inspired by CuMAPz' direct approaches to measuring optimization specific characteristics of CUDA codes.

AIWC currently collects a list of all memory accessed by each thread of execution. In the OpenCL programming model, threads within thread groups execute the same code and are allowed to cooperate on *local memory* addresses. We can group together threads by their work groups to align histories of memory accesses of threads in a group to analyse memory accesses performed by thread groups at each logical

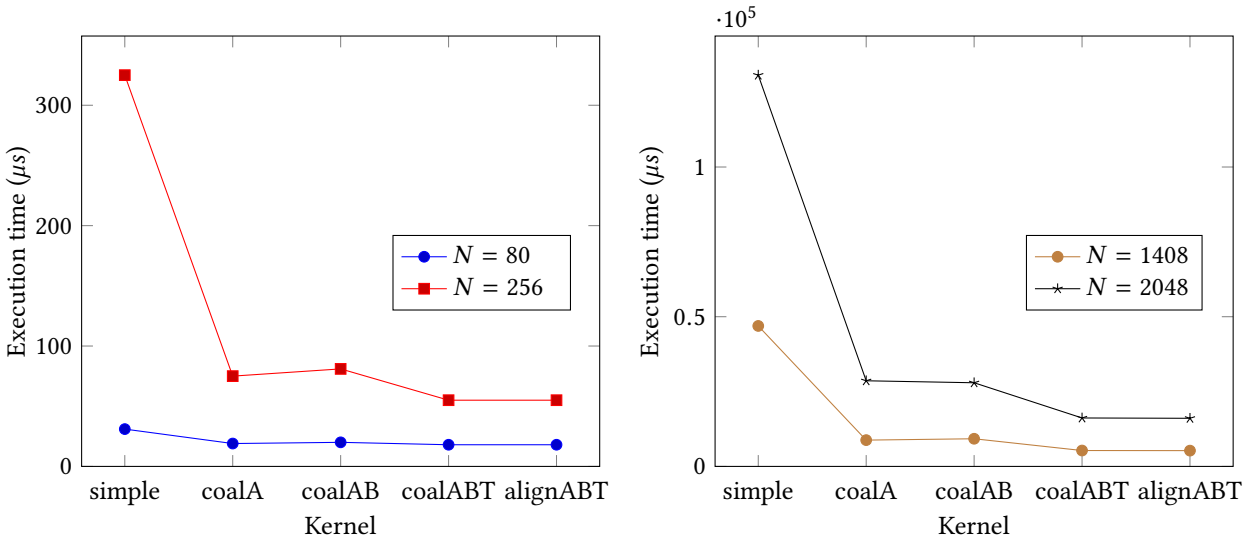

Figure 3: NVIDIA Tesla P100 Runtime performance of section 2 codes averaged over 100 runs

	simple	coalescedA	coalescedAB	coalescedABT	alignedABT
Total memory footprint	196608	196608	196608	196608	196608
90% Memory Footprint	118196	56176	489	489	489
Global MAE	17.02	13.18	9.78	9.78	9.78
LMAE #bits=3	16.02	12.18	8.78	8.78	8.78
LMAE #bits=10	9.02	5.18	1.78	1.78	1.78
Relative Local Memory Usage	0	49.91	94.28	94.28	94.28

Table 1: Selection of AIWC[7] metrics for 256 × 256 matrix multiplication

time step in the symbolic execution of the code. We argue that analysing the accesses occuring at the same logical time step is an accurate, conceptually sound method of predicting the extent of memory access coalescing and bank conflicts we are likely to find upon executing the code on a GPU.

The precise definition of the proposed metric is provided in the following section.

A Parallel Spatial Locality Metric

There are three steps involved in generating an AIWC metric: *recording*, *calculating* and *summarizing* data collected from the symbolic execution of the kernel under inspection.

Record: we first record memory accesses performed by each thread in an OpenCL thread group as described above to achieve a global ordering of all memory accesses performed in a thread group. This ordering is collected in the form of logical timestamps ($t_0..t_{last}$) at which memory accesses occur.

Calculate: for each timestamp $t = t_0..t_{last}$, calculate the n-bits dropped entropy of memory addresses accessed by all threads in a work group within the timestamp t. Here n can range between 0 and 10 as was the case for *LMAE*.

Summarize: average the collected entropy values across all the timestamps to calculate the parallel spatial locality metrics for one thread group. We then average the n-bits dropped entropy summaries across thread groups to obtain the *n-bits dropped parallel spatial locality metrics* for the kernel's execution.

The proposed metric is a parallel computing analogue for MICA's data stride metric that measures the distance between consecutive data accesses in a single threaded environment. In parallel programs, to accurately measure spatial locality of accesses, we must consider memory accesses performed by multiple threads in a close temporal scope. The proposed metric calculates the locality of accesses in

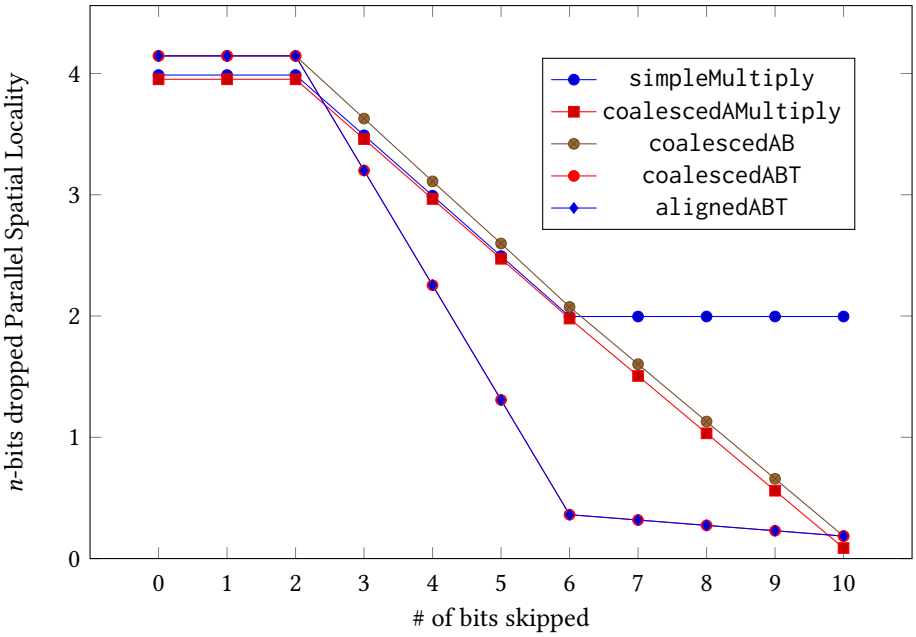

Figure 4: Parallel spatial locality metric obtained from AIWC for matrix multiply kernels for 256×256 matrix multiplication

each time step of the program's execution and steeper drop-offs in n-bits dropped parallel spatial locality scores will be observed in programs that often access nearby memory addresses within the same timestamp. Such programs will perform better on GPUs, as they will make better use of both global memory access coalescing and shared memory bank structures. To a lesser extent, the proposed metric captures performance-critical memory access patterns on CPUs, as pulling a single cacheline from global memory services the threads running on all CPU cores.

7 RESULTS

In figure 4 we see the proposed parallel spatial locality metric obtained using the AIWC tool for each of the matrix multiplication kernels from section 2. In particular we observe that the `coalescedABT` and `alignedABT` kernels have the steepest drop-offs in entropy as the number of bits skipped is increased, which correlates to better locality of parallel memory accesses. It is expected for these kernels to exhibit better parallel spatial locality than `simpleMultiply`, as `coalescedABT` and `alignedABT` make use of local memory, where accesses tend to be localized simply due to the small size of shared on-chip memory typically available on GPUs. Further we find that the proposed metric successfully distinguishes between the `coalescedAB` and `coalescedABT` kernels. It accurately depicts a steeper drop-off for the more optimized `coalescedABT` kernel, where a larger proportion of parallel memory accesses make better use of the memory bank structure of GPU shared memory than all previous kernels. This is a significant improvement over the state-of-the-art AIWC metrics in characterizing how codes localize simultaneous memory accesses to better use the hardware provided.

The Extended OpenDwarfs benchmark suite is a set of OpenCL workload classified according to the computation patterns or Dwarfs. First presented in 2006, the 13 Berkeley Dwarfs are common computational and communication patterns that aim to capture the landscape of parallel computing workloads. The Extended OpenDwarfs benchmark suite presents a set of highly diverse workloads that each fit the programming paradigm of one of the 13 Berkeley Dwarfs. The workloads presented are not optimized for any specific architecture – hence optimizations using OpenCL local memory (which translates to CUDA shared memory) are not performed. Many of these benchmarks can be run with up to four *problem sizes* based on the sizes of caches found in modern CPU memory heirarchies [8]. We present the results of running the AIWC tool on selected benchmarks from the Extended OpenDwarfs benchmark suite [1, 8] with a problem size setting of *medium*, except for the GEM benchmark, which is run at a setting of *tiny*. For benchmarks such as BFS with multiple kernel invocations per run, the AIWC parallel spatial locality metrics of the invocation with the highest executed LLVM-SPIR instructions are presented for analysis. Note that the presented benchmarks use 32 bit numbers (OpenCL `int` and `float` types), and so dropping up to

2 bits of the memory addresses accessed will not change the parallel spatial locality, since any addresses accessed are at least 4 bytes apart.

N-body methods: GEM

The OpenDwarfs benchmark GEM computes the electrostatic potential of a biomolecule by calculating the sum of charges contributed by all atoms in the biomolecule at each specific surface vertex. This is an embarrassingly parallel problem. Each OpenCL work-item operates on a single surface vertex, finding the electrostatic potential generated by looping through every atom in the biomolecule in a global order. The computation pattern is highly regular and memory accesses are perfectly synchronized. Atom data is accessed consecutively, with all work-items simultaneously accessing each single atom's data. The parallel spatial locality metric reflects this pattern of efficient memory utilization (single loads from global memory servicing all OpenCL work-items). The recorded parallel spatial locality approaches the theoretical limit of 0 (0.0124 at 10 bits dropped). This indicates that almost all memory accesses made by the OpenCL kernel are perfectly synchronized between OpenCL threads.

Performance results [8, 11] show that GPUs perform significantly better than CPUs for GEM, as memory unit stalls are at low levels for both CPUs and GPUs due to the highly efficient memory utilization of this benchmark. As memory operations do not present a major slowdown, superior FPU compute capability of GPUs leads to far better performance on GPUs than CPUs [11].

Dense Linear Algebra: Lower-upper decomposition (LUD)

LUD in OpenDwarfs is a program to decompose an input $N \times N$ matrix as the product of one lower and one upper diagonal matrix. Memory access patterns in a dense linear algebra workload such as LUD are typically highly regular and deterministic for each OpenCL work item, based on the matrix dimension and offset parameters. The OpenDwarfs implementation of LUD [1] splits the LUD computation into three kernels: `LUD perimeter` and `LUD diagonal` kernels spawn work-items in a single work dimension; while `LUD internal` performs work in both dimensions of the input matrix.

The LUD kernels partially benefit from memory access coalescing on GPUs, for the lines of code where all the work-items access contiguous memory along a row of the input matrix. However, when threads simultaneously access a column of the input matrix, multiple memory requests are made as addresses accessed are too distant to be coalesced into a single memory transaction on GPUs. This is reflected in the drop in parallel spatial locality for the LUD kernels to approximately half the value at 0 bits skipped. Taking `LUD internal` as an example, the 0-bits skipped parallel spatial locality metric is 4.101, while the 10-bits skipped parallel spatial locality metric is 2.053. The swift decline of the metric to 2.053 from 2 to 6-bits skipped parallel spatial locality indicates that approximately half the parallel memory accesses in `LUD internal` are highly localized. This occurs when work-items simultaneously access contiguous memory along a matrix row. Similar to the presented matrix multiply codes, LUD can be optimized for GPUs by effective utilization of shared memory [1].

Dynamic Programming: Needleman-Wunsch (NW)

Needleman-Wunsch is a dynamic programming algorithm used to perform protein sequence alignment by identifying the similarity level between two strings of Amino acids. Elements in a matrix are computed, with each element dependent on its west, north and north-west neighbours. This dependency enforces a wavefront computation pattern, travelling along the main diagonal of the matrix. Each iteration of the kernels computes over an antidiagonal of the matrix, starting at its top left corner, and finishing at its bottom right corner.

In particular, the wavefront pattern of computations causes each work-item to request distant memory addresses in a parallel fashion. Thus, parallel memory requests into the input matrix prohibit locality. We note the lack of any dropoff in parallel spatial locality as the number of bits dropped increases. This rightly indicates that memory addresses accessed at each logical timestamp are very distant. On GPUs this translates to poor utilization of both memory access coalescing and caching [11]. This trend in the parallel spatial locality metric suggests that a possible improvement for GPU performance would be to load blocks of the input matrix into on-chip local memory to reduce the number of global memory requests – this is typically performed in GPU optimized implementations of the Needleman-Wunsch algorithm [1].

Structured Grids: Speckle Reducing Anisotropic Diffusion (SRAD)

SRAD removes locally correlated noise from images by following a repeated grid update computation pattern on image pixel grids. Conditional statements in the code cause thread divergence, potentially within a work-group, to handle boundary conditions. These boundaries constitute a small portion of the executed work-items, especially on larger datasets and so the effective thread divergence is minimal. Memory access patterns in SRAD are statically determined and relatively localized. Similar to the LUD and `simpleMultiply` kernels, both SRAD kernels observe a rapid drop-off in parallel spatial locality (figure 5) as the number of bits skipped is increased, with the metric stabilizing at approximately half its original value when 0 bits are skipped.

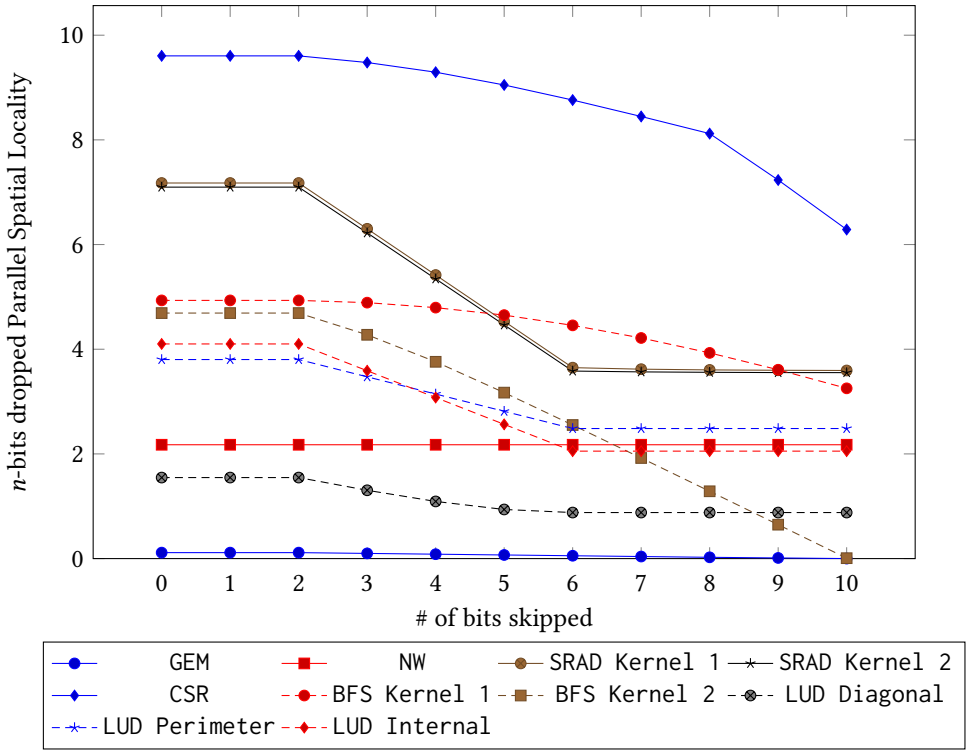

Figure 5: Parallel spatial locality metric for selected OpenDwarfs benchmark kernels

At each point in the kernels' memory access profile, each work-item with global ID (i, j) in an OpenCL work-group accesses the $(i, j)^{th}$ elements of various matrices. The sequential memory access pattern is non-linear since different matrices are accessed consecutively by the kernel, prohibiting ideal caching [11]. However, memory requests made simultaneously by a work-group always fall within a rectangular block of one of the matrices. This allows memory access coalescing on GPUs for OpenCL work-items accessing contiguous matrix data, greatly reducing the number of memory requests made on each line of the kernel code on GPUs. However, work-items accessing data along a column of the matrix do not observe memory access coalescing. Thus we observe that while cache hit rates are typically low on this benchmark, particularly on GPUs [11], GPUs can hide the latency of global memory accesses through memory access coalescing to some extent.

Graph Traversal: BFS

Graph traversal algorithms require pointer chasing operations to traverse nodes of a graph and perform calculations. The Breadth First Search implemented in OpenDwarfs calls two OpenCL kernels to traverse nodes immediately connected to the list of nodes at each depth starting from the root node. BFS is characterized by an imbalanced workload per kernel launch depending on the degree of the nodes being operated on, with only a proportion of launched nodes performing meaningful work. As such, the AIWC features collected for each kernel invocation vary significantly. The parallel spatial locality metric collects the entropy of memory accesses at each time step, and workload imbalances across work-items are dealt with by averaging entropy scores across all the time steps in the execution of an OpenCL work-group.

Of the two BFS kernels, it is `BFS kernel 1` that performs the graph traversal necessary to generate a list of neighbours at each node. The memory access patterns in `BFS kernel 1` are irregular. Work-items fetch discontinuous memory locations attributed to any particular node in the graph, depending on the connectivity of the node they operate on. The program structure involves multiple levels of pointer-chasing and thus the precise memory address accessed are runtime dependent. This leads to poor parallel spatial locality. Thus, we see a slow dropoff in parallel spatial locality for `BFS kernel 1`.

Sparse Linear Algebra: CSR

Compressed Sparse Row Matrix-Vector Multiplication (CSR) computes the sum of each of a matrix's row's elements after it is multiplied by a given vector. The matrix is stored in a compressed row storage sparse matrix format, which is very

efficient for storage when the number of zero elements is far greater than the number of non-zero elements. Three inputs are provided to the CSR kernel. The non-zero elements of a matrix are stored in row-major order in *Ax*, along with separate arrays *Aj*, *Ap* indicating the position of each non-zero element in the matrix.

In this particular implementation, a single OpenCL work-item is assigned to compute over each row of the input sparse matrix. The locations of matrix data read by each work-item are dependent on the number and position of non-zero values in the sparse-matrix, which are decided by the values in *Aj* and *Ap*. Similar to BFS, this means memory access patterns are runtime-dependent due to indirect addressing. This pattern of indirect addressing is typical of applications in the Spare Linear Algebra Dwarf, which severely hinders locality of memory accesses performed. The collected parallel spatial locality metric reflects this trend. Figure 5 shows the gradual decline in parallel spatial locality as the number of bits dropped is increased.

8 CONCLUSIONS AND FUTURE WORK

To the best of our knowledge this work is the first to critically examine AIWC's measurement of memory access patterns to guide hardware-specific optimization. We implemented and justified the addition of relative local memory usage to AIWC to help characterize memory-based GPU code optimizations. We proposed the concept of parallel spatial locality to capture the idea of closeness of memory accesses made by parallel OpenCL workloads, and implemented a new parallel spatial locality metric to AIWC. We ran the modified AIWC tool on matrix multiply kernels and selected OpenDwarfs benchmarks, presenting results and analysis to validate our methodology.

Our proposed parallel spatial locality metric may also correlate to some memory-based optimization strategies for CPUs. Future work would apply the approach followed by this paper to optimizations for CPU and FPGA architectures to critically evaluate the viability of AIWC in analyzing memory access patterns in codes targeted to these architectures.

Our work in this paper intended to guide optimization efforts. Another potential use case for AIWC is providing architecture-independent performance predictions for OpenCL kernels by generating machine learning models based on AIWC metrics [6]. Future work would modify the presented metrics and develop new memory metrics with the specific intent of being fed to machine learning models to predict kernel performance on arbitrary and novel architectures.

REFERENCES

[1] Extended OpenDwarfs. https://github.com/ANU-HPC/OpenDwarfs/commit/dee488cac9833f029dfada35ff6ae4077b68c4b5, Jan 2020.

[2] Karthik Ganesan, Lizy John, Valentina Salapura, and James Sexton. A performance counter based workload characterization on Blue Gene/P. In *International Conference on Parallel Processing (ICPP)*, pages 330–337. IEEE, 2008.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[4] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3), 2007.

[5] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. 2016.

[6] Beau Johnston, Gregory Falzon, and Josh Milthorpe. Opencl performance prediction using architecture-independent features. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*, pages 561–569. IEEE, 2018.

[7] Beau Johnston and Josh Milthorpe. AIWC: opencl based architecture independent workload characterisation. *CoRR*, abs/1805.04207, 2018.

[8] Beau Johnston and Josh Milthorpe. Dwarfs on accelerators: Enhancing OpenCL benchmarking for heterogeneous computing architectures. In *Proceedings of the* 47^{th} *International Conference on Parallel Processing Companion*, ICPP '18, pages 4:1–4:10, New York, NY, USA, 2018. ACM.

[9] Beau Johnston, James Price, Moritz Pflanzer, Petros Kalos, Tom Deakin, Nido Media, and Daniel Saier. BeauJoh/Oclgrind: Adding AIWC – An Architecture Independent Workload Characterisation Plugin. https://doi.org/10.5281/zenodo.1134175, December 2017.

[10] Yooseong Kim and Aviral Shrivastava. Cumapz: a tool to analyze memory access patterns in CUDA. In Leon Stok, Nikil D. Dutt, and Soha Hassoun, editors, *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pages 128–133. ACM, 2011.

[11] Konstantinos Krommydas, Wu-chun Feng, Christos D Antonopoulos, and Nikolaos Bellas. Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *Journal of Signal Processing Systems*, 85(3):373–392, 2016.

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN notices*, volume 40, pages 190–200. ACM, 2005.

[13] NVIDIA Corporation. CUDA C Best Practices Guide. 2019.

[14] NVIDIA Corporation. CUDA C Programming Guide. 2019.

[15] Yakun Sophia Shao and David Brooks. Isa-independent workload characterization and its implications for specialized architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–255. IEEE, 2013.

[16] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: data orchestration and tuning for OpenCL devices. *Euro-Par 2010-Parallel Processing*, pages 275–286, 2010.

A SIMPLEMULTIPLY KERNEL

```
__kernel void simpleMultiply(__global float *A,
     __global float *B,
     __global float *C,
     int N)
{
   const int globalRow = get_global_id(0); //
       Row ID of C (0..N-1)
   const int globalCol = get_global_id(1); //
       Col ID of C (0..N-1)
```

```
        // Compute a single element of C (loop over K)
        float acc = 0.0f;
        for (int k = 0; k < N; ++k) {
           acc += B[k * N + globalCol] * A[globalRow * N + k];
        }
        // Store the result
        C[globalRow * N + globalCol] = acc;
     }
```

B COALESCEDAMULTIPLY KERNEL

```
__kernel void coalescedAMultiply(
      const __global float* A,
      const __global float* B,
      __global float* C,
      const int N)
{
   __local float aTile[TILE_DIM][TILE_DIM];

   const int localRow = get_local_id(0);
   const int localCol = get_local_id(1);

   const int globalRow = get_global_id(0);
   const int globalCol = get_global_id(1);

   __private float sum = 0.0f;

   const int numTiles = N / TILE_DIM;
   __private const int tiledRow = globalRow*N+localCol;
   for (int i = 0; i < numTiles; i++) {
      aTile[localRow][localCol] = A[tiledRow+i*TILE_DIM];
      barrier(CLK_LOCAL_MEM_FENCE);
      for (int k = 0; k < TILE_DIM; k++) {
         sum += aTile[localRow][k] * B[(i*TILE_DIM+k)*N+globalCol];
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }
   C[globalRow*N+globalCol] = sum;
}
```

C COALESCEDAB KERNEL

```
__kernel void coalescedAB(
      const __global float* A,
      const __global float* B,
      __global float* C,
      const int N) {
   __local float ASub[TILE_DIM][TILE_DIM];
   __local float BSub[TILE_DIM][TILE_DIM];

   const int localRow = get_local_id(0);
   const int localCol = get_local_id(1);
   const int globalRow = get_global_id(0);
   const int globalCol = get_global_id(1);

   float acc = 0.0f;
   const int numTiles = N/TILE_DIM;

   for (int i = 0; i < numTiles; i++) {
      const int tiledRow = globalRow*N+i*TILE_DIM + localCol;
      const int tiledCol = globalCol + (TILE_DIM*i + localRow)*N;
      ASub[localRow][localCol] = A[tiledRow];
      BSub[localRow][localCol] = B[tiledCol];

      barrier(CLK_LOCAL_MEM_FENCE);

      for (int k=0; k<TILE_DIM; k++) {
         acc += ASub[localRow][k] * BSub[k][localCol];
      }

      barrier(CLK_LOCAL_MEM_FENCE);
   }
   C[globalRow*N + globalCol] = acc;
}
```