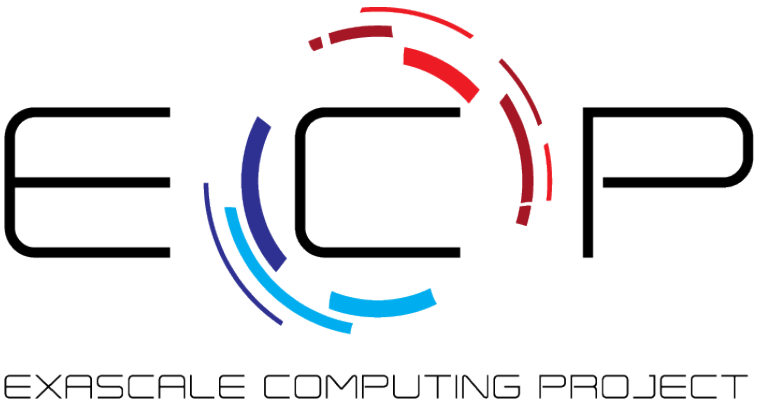
Quantitative Performance Assessment of Proxy Apps and Parents

Report for ECP Proxy App Project Milestone ADCD-504-9

David Richards[1], Omar Aaziz[2], Jeanine Cook[2], Jeffery Kuehn[3], Shirley Moore[4], David Pruitt[5], Courtenay Vaughan[2], and The ECP Proxy App Team[6]

[1]Lawrence Livermore National Laboratory, Livermore, CA
[2]Sandia National Laboratories, Albuquerque, NM
[3]Los Alamos National Laboratory, Los Alamos, NM
[4]Oak Ridge National Laboratory, Oak Ridge, TN
[5]Univeristy of Texas at El Paso, El Paso, TX
[6]https://proxyapps.exascaleproject.org/team

April 2020



LLNL-TR-809403

Contents

1 Executive Summary

The ECP Proxy Application Project has an annual milestone to assess the state of ECP proxy applications and their role in the overall ECP ecosystem. Our FY20 milestone (ADCD-504-9) proposed to:

> Assess the performance and fidelity of proxy applications, including those in the ECP Proxy App Suite, relative to the ECP Application workload on heterogeneous platforms. Use proxy application and selected ECP applications to assess the utility of critical elements of the Exascale toolchain, especially tools used to collect performance data. Identify gaps in coverage and/or common situations in which proxies may fail to adequately represent ECP applications.

This report presents highlights of these efforts.

Section 2 describes work that has been done to compare the performance of proxy applications on AMD MI60 vs. Nvidia V100 GPUs. So far only a small set of ECP proxies are running on AMD GPUs, but we will continue to expand this analysis as additional proxies become available. We find that although the MI60 and V100 have nearly the same measured memory bandwidth, memory bound proxy app kernels perform 20–30% worse on the MI60. Further work is needed to refine these comparisons to determine whether the root cause is due to differences in the hardware, software stack, platform specific optimization, or some combination of the three.

Section 3 describes our continuing effort to find methods to accurately assess the similarity of proxies and parents. We have recently seen very encouraging results using a cosine similarity metric. This technique uses the angle between two vectors of hardware performance counters to characterize the similarity (or difference) between two applications or proxies. We show not only that several widely used proxies are highly similar to their parents, but also that they differ from non-related codes. We also show that cosine similarity can be used to identify gaps and redundacies in suites and even to gain insight into the effects of architectural differences between platforms.

Our work on assessing the Exascale toolchain is ongoing. Our successes with performance measurement tools are evident from the data provided in this report. However, our assessments across the broader tool chain are still too incomplete to provide a meaningful report at this time. We will continue to assess tools and work with vendors and third party developers as issues are identified.

2 Performance Assessment of GPU-enabled Proxy Applications

A small (but growing) set of proxy application have now been ported for execution on both AMD and Nvidia GPUs. This makes it possible to begin comparing performance on DOE proxies across vendors. The GPU performance assessment is carried out by identifying the most important GPU kernels and measuring their performance characteristics. We measure the characteristics of the kernels on two different GPU architectures currently in use in pre-exascale systems. We will evaluate GPU performance on recent NVIDIA (V100) and AMD (MI60) GPUs as implementations for both platforma become available. We use the NVIDIA nvprof and nsight command-line profiling tools to collect performance data on the NVIDIA V100 and the AMD Rocprof profiler to collect data on the AMD MI60.

We attempt to profile the proxy application using one or more input sets representative of the challenge exascale problem, scaled down to represent performance on a single node or small number of nodes. We attempt to use recent versions of the proxy applications that are implemented using the programming model(s) being used by the parent application team as they work towards scaling to exascale. Once the most important GPU kernels have been determined from basic timing profiles, we carry out more detailed performance data collection and analysis for these kernels. We collect data to calculate instruction mix, arithmetic intensity, achieved memory bandwidth, and percentage of peak attained for each kernel. We use the arithmetic intensity to place the kernel on the relevant GPU roofline model.

2.1 GPU Architectures and Programming Models

2.1.1 NVIDIA V100

We used a node of the OLCF Summit supercomputer for our V100 evaluations. The basic building block of Summit is the IBM Power System AC922 node. Each of the approximately 4,600 compute nodes on Summit contains two IBM POWER9 processors and six NVIDIA Tesla V100 accelerators and provides a theoretical double-precision capability of approximately 40 TF. Each POWER9 processor is connected via dual NVLINK bricks, each capable of a 25 GB/s transfer rate in each direction. Nodes contain 512 GB of DDR4 memory for use by the POWER9 processors and 96 GB of High Bandwidth Memory (HBM2) for use by the accelerators. Additionally, each node has 1.6 TB of non-volatile memory that can be used as a burst buffer. The AC922 uses NVIDIA's NVLink interconnect to pass data between GPUs as well as from CPU-to-GPU.

The NVIDIA Tesla V100 accelerator uses 80 Streaming Multiprocessors (SMs), each of which has 64 FP32 cores, 64 INT32 cores, 32 FP64 cores, and 8 Tensor cores. The theoretical peak floating point performance is 7.8 TFLOPS for FP64, 15.7 TFLOPS for FP32, and 125 TFLOPS for Tensor. The V100 has 16GB of 4096-bit HBM2 memory. The capacity of the combined L1 data cache and shared memory is 128 KB/SM. All of it is usable as a cache by programs that do not use shared memory, and shared memory is configurable up to 96 KB. The shared L2 cache has a capacity of 6144 KB. The theoretical peak HBM2 memory bandwidth is 900 KB/sec.

The V100 uses unified memory, which is a single virtual address space that is accessible to any processor (CPU or GPU) within the node. This managed memory is automatically migrated to the accessing processor, eliminating the need for explicit data transfers. Unified memory performance is further improved on the V100 through the use of access counters to automatically tune unified memory by determining where a page is most often accessed.

For programming models on the V100, we used CUDA and RAJA.

2.1.2 AMD MI60

We used a node of the OLCF Lyra, a 10-node AMD cluster, for our MI60 evaluations. At the time we did our evaluations in December 2019, each compute node of Lyra had one AMD EPYC 7451 CPU and two AMD Vega 20 Radeon Instinct MI60 GPUs connected with Infinity Fabric links. The MI60 has 64 Compute Units (CUs), each with theoretical peak floating point performance of 29.5 TFLOPS for FP16, 14.7 TFLOPS for FP32, and 7.4 TFLOPS for FP64. The MI60 has 32 GB of 4096-bit HBM2 memory with a theoretical peak bandwidth of 1 TB/s.

We used the HIP (Heterogeneous-Compute Interface for Portability) programming model on the MI60. HIP is a C++ runtime API that allows code developers to write portable code to run on AMD and NVIDIA GPUs. Since the HIP API is similar to CUDA, porting existing CUDA codes to HIP is fairly straightforward in most cases. HIP uses the underlying Radeon Open Compute (ROCm) or CUDA platform that is installed on the system. HIP is a thin layer and has little to no performance overhead over coding directly in CUDA.

2.2 Proxy Applications

2.2.1 SW4lite

SW4lite is a bare bones version of SW4 intended for testing performance optimizations in a few important numerical kernels of SW4. SW4 stands for Seismic Waves, 4th order. It is a seismic wave propagation code that solves the seismic wave equations in displacement formulation using a node-based finite difference method. We characterized both the CUDA and RAJA versions of SW4lite on the V100. For the MI60, we used a HIP version of SW4lite provided by AMD. We were also able to convert the CUDA version of SW4lite to HIP ourselves using the hipify-perl tool, after making some minor changes to the CUDA code. We were able to compile the same HIP source code to run on either the NVIDIA V100 or the AMD MI60 GPU.

2.2.2 PENNANT

PENNANT is an unstructured mesh physics proxy application that contains mesh data structures and a few physics algorithms adapted from the LANL rad-hydro code FLAG. PENNANT is also intended to give a sample of the typical memory access patterns of FLAG. For the V100, we used the code from the CUDA branch of the PENNANT github repository pointed to from the ECP Proxy Apps catalog. For the MI60, we used a HIP version of PENNANT provided by AMD. The HIP version compiled and ran on Lyra using the hsa, hip, and rocm modules, after we had also installed rocmthrust. The HIP version did not compile for NVIDIA GPUs, and we were unable to convert it ourselves.

2.3 Roofline Modeling Methodology

We used the BabelStream, gumembench, and mixbench low-level benchmarks to construct the rooflines for the V100 and MI60 machine models. Measured achieved memory bandwidth for V100 was 829 GB/s, which is 92% of theoretical peak. Measured peak double precision floating point performance for V100 was 7069 GFLOPS/s, which is 91% of theoretical peak. Measured achieved memory bandwidth for MI60 was 806 GB/s, which is 80% of theoretical peak. Measured double precision floating point performance for MI60 was 7306 GFLOPS, which is 98% of theoretical peak. We constructed the machine roofline models used in section 2.4 from the measured peaks rather than the theoretical peaks.

To calculate the arithmetic intensity of the kernels needed to place them on the roofline models, we needed to measure the number of floating point operations and the number of bytes moved to and from device memory. On the V100, we used the nvprof metric flop_count_dp to measure double precision floating point operations, and we used dram_read_bytes and dram_write_bytes to measure memory traffic. On the MI60, we collected Vega 20 hardware counter data using rocprof. For bytes moved, we used the FetchSize and WriteSize metrics. Because there is no metric available on the MI60 for counting floating point operations, we used the SQ_INSTS_VALU metric to count floating point instructions and used an instruction-based roofline model.

2.4 Results

2.4.1 SW4lite

The gaussianHill-rev.in input is recommended by the SW4lite developers as a scaled-down problem that is intended to be representative of the exascale challenge problem in that it exercises the relevant parts of the code with similar node performance. This input uses an analytical topology, rather than a real t topology read from a file, and exercises the curvilinear kernels that are the most time consuming for the exascale problem. Thus, we use this input for our investigations.

Our previous FY19 investigation of the representativeness of SW4lite with respect to SW4 showed different profiles. With SW4 run on the V100 using a Berkeley input with realistic topology read from a file, the curvilinear kernel took the largest proportion of the runtime, 26%, followed by the supergrid damping kernel at 21% and the stress calculation on the Cartesian grid at 19%. On the gaussianHill-rev.input, both SW4 and SW4lite spend only around 5 to 6 percent of the runtime in the curvilinear kernels and spend the largest proportion of the runtime, around 36% in the supergrid damping kernel. However, given that the curvilinear kernels are considered the most important by the SW4 developers we focused our analysis on the curvilinear kernel.

We ran both the CUDA and RAJA versions of SW4lite on the V100. Although the overall runtime of the CUDA version is somewhat faster, the runtimes of the curvilinear kernel are very close and the roofline plots for this kernel are similar. We also ran the HIP version of SW4lite, compiled for NVIDIA, on the V100, and the runtime averaged 5.8 seconds, only slightly more than the CUDA version. Since the HIP version of SW4lite for AMD is derived from the CUDA version, we used the CUDA version for our comparison. The overall runtime for SW4lite run for 100 steps on the gaussianHill-rev.in input was 5.6 seconds for the CUDA version on the V100 and 8.7 seconds for the HIPified version on the MI60.

The roofline models for the curvilinear kernel (CUDA version) on the V100 and the MI60 are shown in Figures 1 and 2, respectively. The curvilinear kernel is memory bound on both GPU architectures. It runs at about 28% of peak on the V100 and at about 22% of peak on the MI60.

2.4.2 PENNANT

We ran the CUDA and HIP versions of PENNANT on the V100 and MI60, respectively, using the sedovbig input. The overall runtime for the CUDA version of PENNANT on the V100 was 5.2 seconds, and the overall runtime of the HIP version on the MI60 was 7.0 seconds. The gpuMain2 kernel takes the majority of the runtime at approximately 60%. The roofline models for the gpuMain2 kernel on the V100 and the MI60 are shown in Figures 3 and 4, respectively. The gpuMain2 kernel is memory bound on both GPU architectures. It runs at about 14% of peak on the V100 and at about 11% of peak on the MI60.

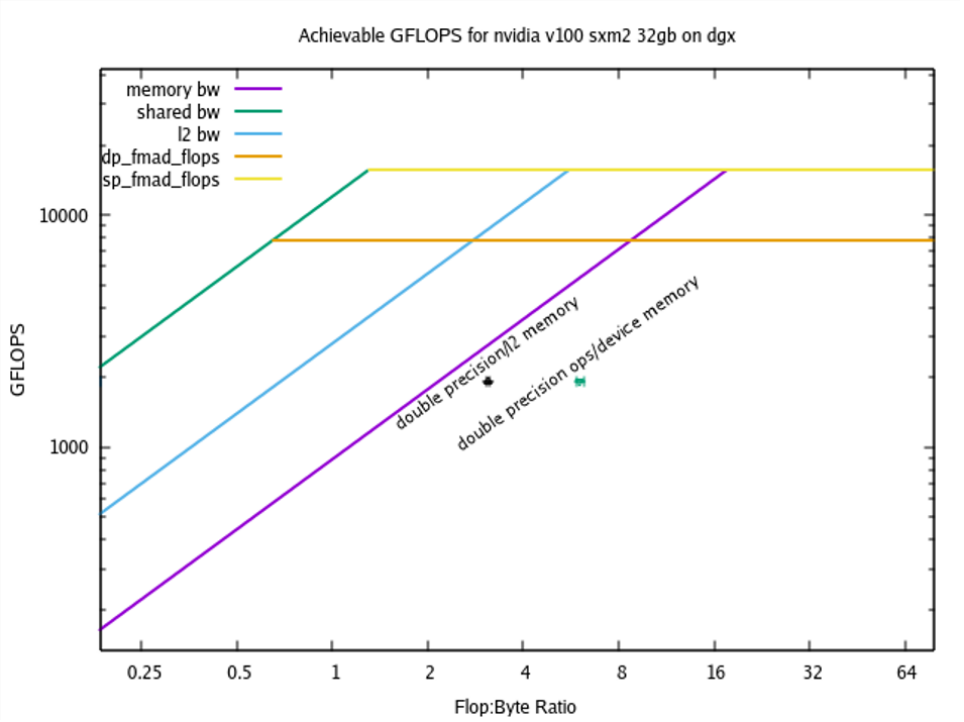


Figure 1: Roofline plot of SW4lite curvilinear kernel on V100 GPU

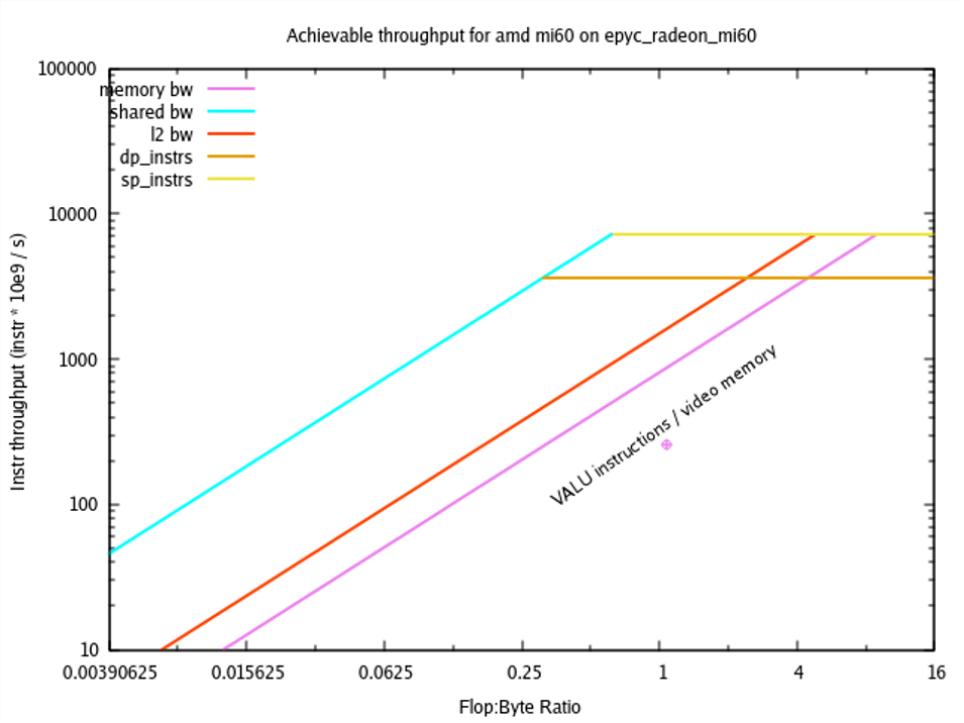


Figure 2: Roofline plot of SW4lite curvilinear kernel on MI60 GPU

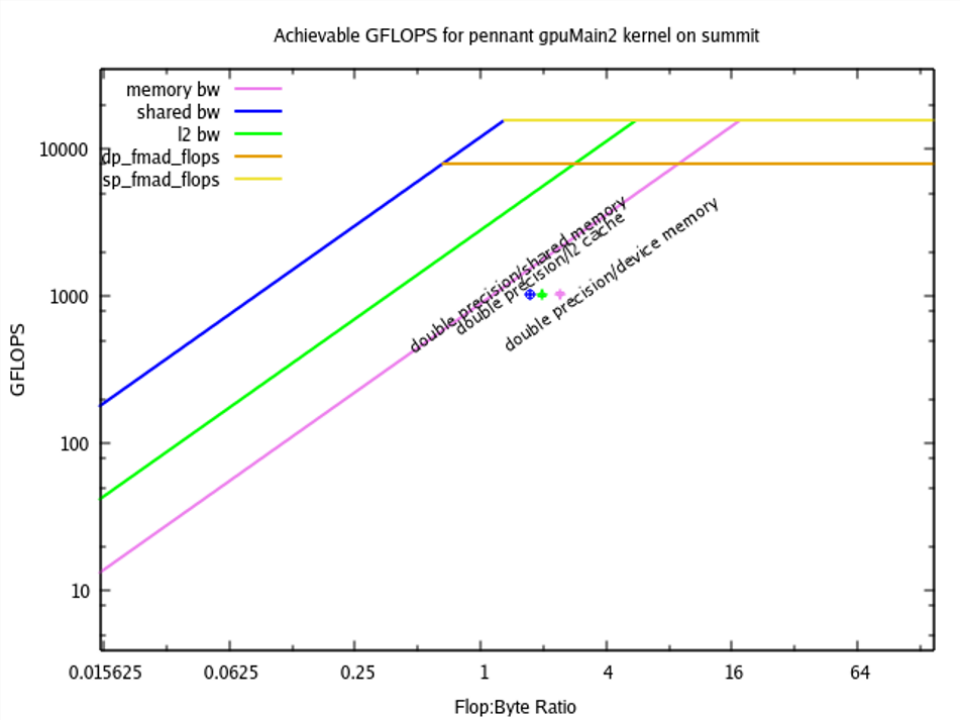


Figure 3: Roofline plot of PENNANT gpuMain2 kernel on V100 GPU

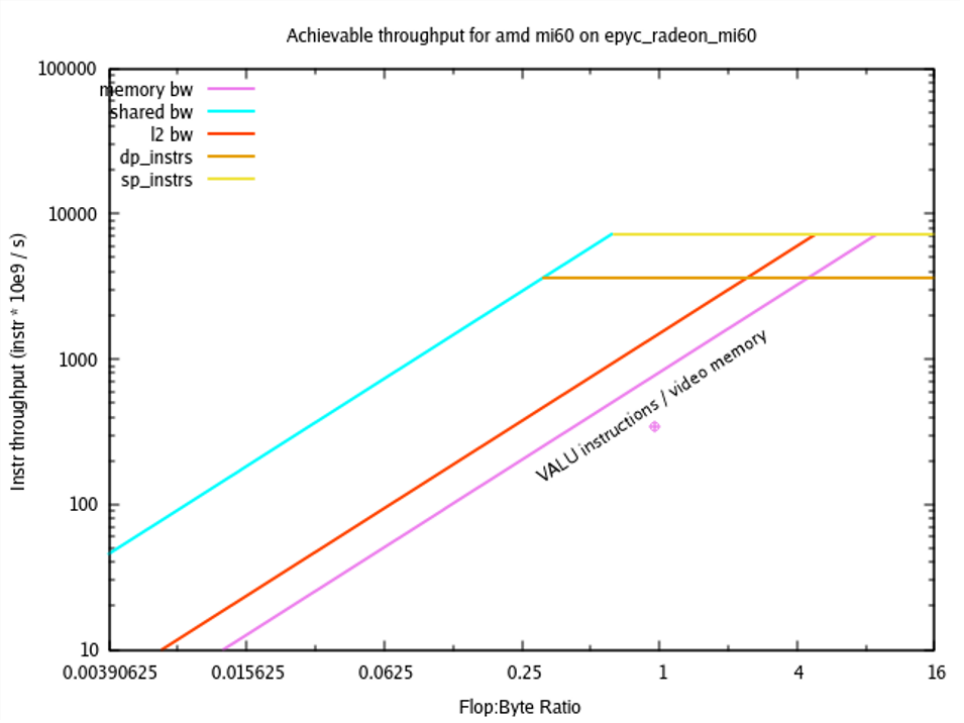


Figure 4: Roofline plot of PENNANT gpuMain2 kernel on MI60 GPU

2.5 Conclusions

The theoretial peak floating point performance and memory bandwidth for the NVIDIA V100 and AMD MI60 GPUs are similar. However, whereas we were able to achieve 92% of the theoretical peak memory bandwidth running BabelStream on the V100, we were able to achieve only 80% on the MI60. The proxy app kernels ran 20 to 30 percent more slowly on the MI60 than on the V100. The difference does not appear to be due to HIP overhead, since the HIP version of SW4lite ran about the same as the CUDA version on the V100. The difference may be due to the lower achievable memory bandwidth on the MI60, since the kernels are memory bound, and due to the proxy applications having been optimized for NVIDIA GPUs (for example, the use of shared memory in the CUDA version of SW4lite).

3 Proxy Application Representativeness

For this milestone, our primary goal was to develop an improved methodology for identifying the extent to which a proxy represents its parent with respect to system behavior (e.g., node, memory, communication, network). Our prior work uses classical statistical methods that include principal components analysis (PCA) to reduce data dimensionality and hierarchical clustering to determine similarity. Data is collected from hardware performance counters and used as input to PCA, which is subsequently used as input to the clustering algorithm. Our new methodology uses the same collection mechanism, but uses a cosine similarity metric to quantify the relationship between proxy and parent pairs.

This year we have applied cosine similarity to both node and memory behavior and communication (MPI) patterns for a suite of 10 ECP applications and proxies. This report focuses on the node/memory work that to date has not been submitted for publication. Our work on applying cosine similarity to understand communication differences between proxy and parent applications has been published previously [2].

3.1 Cosine Similarity

Cosine similarity is a machine learning technique that is heavily used in artificial intelligence applications such as automatic document comparison and identification [3–5]. In this work, we use it to quantitatively determine

- if a proxy represents its parent with respect to node and memory behavior,
- a minimal set of proxy applications that represent the composite behavior of the application space,
- a minimal set of proxy applications that are rerpesentative of the behavior of the application space for specific components of the architecture such as memory, cache, and branching.

We can do this for essentially any system, but in this work we initially use two Intel platforms.

Cosine similarity takes two vectors as input and calculates the angle (cosine) between them using properties of the dot product.

$$\cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|} \tag{1}$$

The angle characterizes the distance between the two vectors, which can then be used to indicate the similarity of the two vectors. We use hardware performance counter data to form an application vector or signature. For each system, we use the entire set of available and reliable performance events. We deem an event reliable through testing and inspection of results. Each application signature for each platform comprises hundreds of performance counter events.

Although the calculation of cosine similarity does not require reduction of data dimensionality, we apply a simple selectivity metric in order to significantly reduce the amount of data we need to collect. Performance counters that exhibit a relatively low standard deviation across the set of applications are deemed non-selective and are excluded from further data collection and analysis. Our selectivity filter is very similar to techniques such as principal components analysis (PCA). Performing PCA on our data and comparing it to the results using selectivity is part of our planned future work. Even with these reductions, the data used for this work can not be collected in a single run and we must run a single application collection cycle multiple times to ensure statistically acceptable variation. This results in several hundred jobs for the application suite that we currently are examining.

We look at cosine similarity between the the members of our suite using all available and reliable performance events as a signature and we categorize these events based on component or function so that we can look at similarity of applications based on each primary system component. Performance event/component categories/groups include:

- branch
- instruction mix
- cache
- pipeline
- memory
- virtual memory

The pipeline category is further broken down into:

- decode
- issue
- dispatch
- execute
- memory
- retire

Categorizing events into performance groups provides an understanding of which proxies/parents comprise a representative workload for each of these system components. For example, we can pick out a minimal set of proxy and/or parent applications that represent the cache behavior seen across the entire suite. Through this sort of analysis, we can also identify which applications we should recommend to vendors and other researchers for doing explorations of architectural innovations for cache, memory, virtual memory, branch prediction, etc.

Cosine similarity can be used to understand performance differences across many system parameters. For example, one could use it to explore performance difference across operating system versions, compilers, compiler optimizations, or application optimizations, to name a few. We are currently working on comparisons across architectures by creating a full performance vector for each application for each architecture and using that as input to cosine similarity. For example, we would have an ExaMiniMD Broadwell vector and an ExaMiniMD Skylake vector and compute the cosine similarity between the two. This would indicate if the overall behavior on the two systems is different or not. If it is not, then no feature on the newer architecture changed the performance as compared to the older architecture. If there is performance difference (i.e., little similarity) between the two systems, we could look at the cosine similarity using application signatures from the various performance groups to identify which component(s) affects the overall performance change.

3.2 Methodology

To understand proxy representativeness, we use the suite of proxy/parent pairs listed in Table 1. Note that we also include two proxies, *pennant* and *snap*, without their respective parent applications. This table shows the applications and the problems and input sizes used. We adjust problem sizes to use around 40–50% of the node memory for each system. All applications are compiled with the Intel 19 compiler, using vectorization flags at optimization level O3.

We execute all applications using MPI only on 128 ranks. This scale is chosen because it is large enough to observe important communication patterns but not so large that jobs are forced to wait for days in a scheduling queue to execute.

Table 1: Applications and Input Problems

Proxy/Parent	Domain	Input Line	Input File
ExaMiniMD/ LAMMPS	Molecular dynamics	-il in.snap.Ta.mod –comm-type MPI –kokkos-threads=1 -i in.snap.Ta.mod	in.snap.Ta.mod in.snap.Ta.mod
miniQMC/ QMCPack	Quantum Monte Carlo	-r 0.99 -g '2 2 2' NiO-example.in.xml	N/A NiO-example.in.xml; NiO-fcc-supertwist111-supershift000-S64.h5
sw4lite/ SW4	Seismology	new_gh.in	new_gh.inz
SWFFT/ HACC	Cosmology	15 2048 params4x32 -n	N/A params4x32 -n
pennant	Unstructured mesh hydro	leblancx128.pnt	leblancx128.pnt
snap	Neutron particle transport	inh0004t1a out4a	inh0004t1a

We normalize the event counts in each application signature by the total number of cycles executed for each application prior to computing both cosine similarity and selectivity. We experimented with various methods of normalization (including not normalizing) and they all resulted in essentially the same cosine similarity matrix.

We have collected and analyzed data for two systems to date: Intel Broadwell and Intel Skylake. These processor architectures are shown in Figures 5 and 6. A summary of the fundamental differences between these two architectures is that Skylake is characterized by:

- 6 memory channels versus 4 on Broadwell,
- larger memory hierarchy, with more associative caches, larger sizes, and larger cache bandwidths,
- more execution units, particularly to support AVX512 in Skylake,
- a wider decode and front-end with an increased number of in-flight instructions.

We are currently working on the infrastructure for collecting data on IBM Power9 and Nvidia and AMD GPUs.

We use the Light-weight Distributed Metric System (LDMS) monitoring infrastructure from Sandia to collect all performance data (MPI/communication, node, memory, network). We use the PAPI and Aries plug-in samplers to collect respective node and network performance data. In addition to computing cosine similarity, we compute dozens of metrics from the performance counter data to help understand and validate the cosine similarity results. We also do a bottleneck analysis using the Intel Topdown Microarchitecture Analysis (TMA), which we have implemented within LDMS. We do not report this data here for brevity, but this will be made available in the future on the ECP Proxy App Suite website [1].

It required significant effort to classify Intel performance counter events into performance groups (e.g., branch, cache, pipeline). We plan to make this available to the broader community soon through github.

3.3 Results

We define an application vector or signature as the performance data collected for a specific application on a particular architecture. On each architecture, a full application signature comprises around 400 performance event counts. As stated previously, we subsequently separate performance

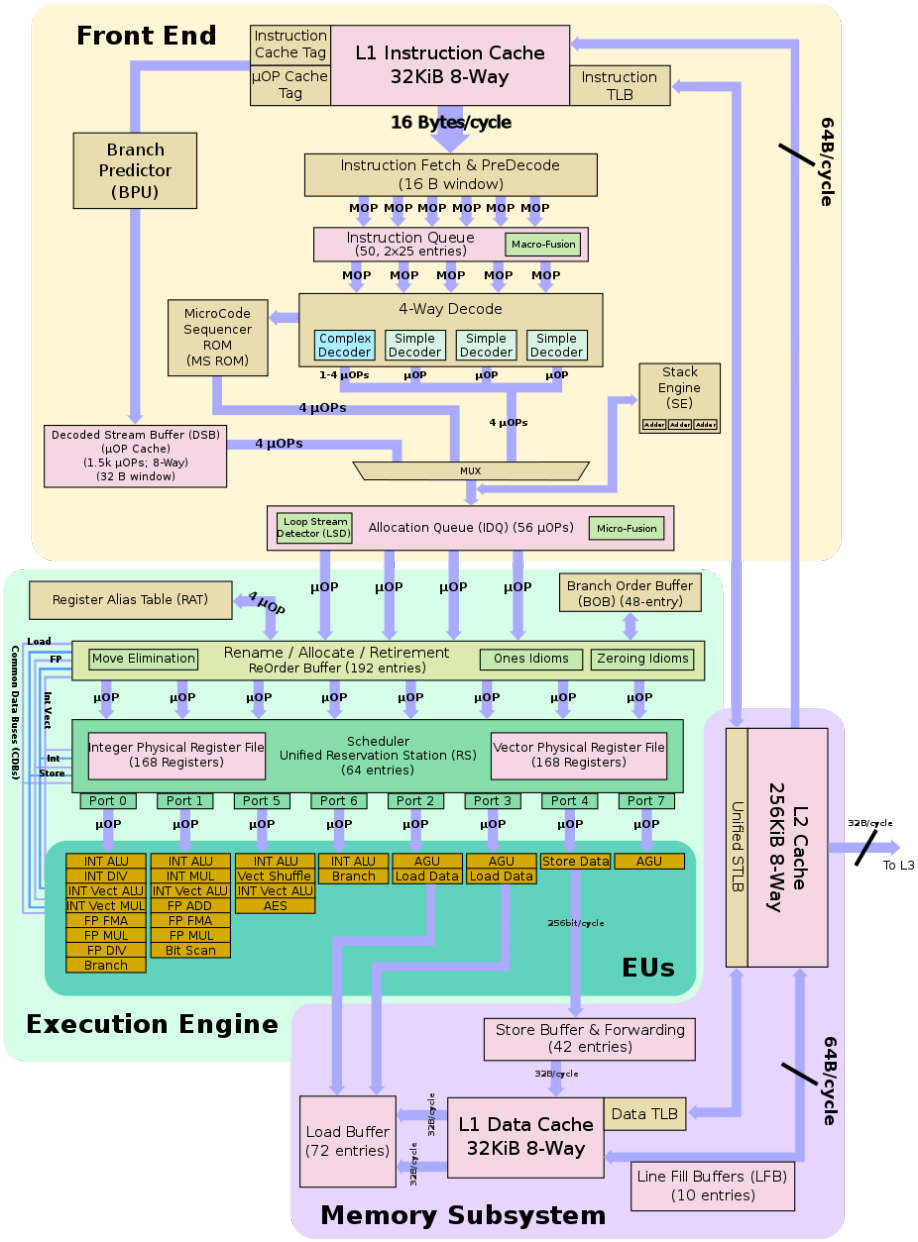


Figure 5: Intel Broadwell Processor Architecture

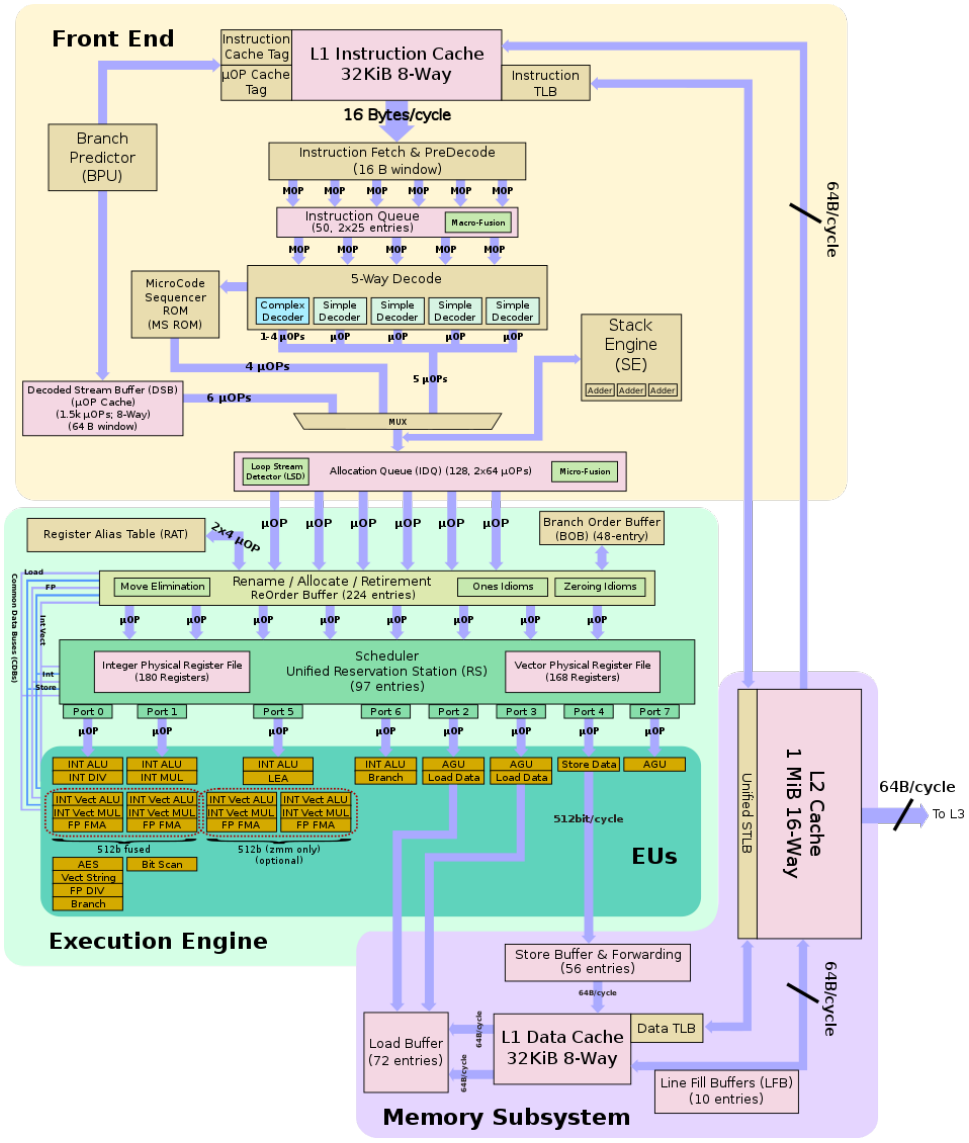


Figure 6: Intel Skylake Processor Architecture

events into categories that pertain to major components in the architecture such as memory, cache, and branching. These form smaller performance signatures of specific components of the architecture that we again use to extract cosine similarity. We report results for Intel Skylake and Broadwell architectures using full application signatures and only significant results when using signatures from specific performance groups.

Figure 7 shows the cosine similarity (COS) results for the Intel Broadwell system using all available and reliable performance event counts (full vector) in the application signature. In this figure (and all of the following COS figures), colors are interpreted as follows:

- Green — indicates good similarity. As the similarity decreases (i.e., the angle between applications becomes larger), the shade of green becomes lighter.
- Yellow — indicates some similarity, but also some difference. Whether this is acceptable or not is really based on the intended use of either the proxy or the parent application.
- Orange — indicates poor similarity. The two applications are more different than they are similar. The shade of orange darkens as the similarity decreases (i.e., difference increases).
- Red — indicates almost no similarity. These applications exhibit very different behavior on this architecture or for this component.

The scale here was somewhat arbitrarily assigned. In researching methods for defining such a scale, we found no rigorous quantitative methods. Validating and improving our assignment scheme will be included in future work.

First notice in Figure 7, that the diagonal reflects self-similarity of each application (e.g., the similarity between ExaMiniMD and ExaMiniMD) in that the angle between them is zero. The next observation is that all of the proxy/parent pairs have good similarity—they are all some shade of green. SWFTT and HACC have the highest similarity. ExaMiniMD and LAMMPS are similar to each other but are very different compared to any other application. This is also the case for miniQMC and QMCPack. SW4 and sw4lite have some similarity with SWFFT, HACC and snap (yellow entries). We refer to this as redundancy because if we choose one of these applications and discard the other two, we would still represent all of the unique behavior of the entire suite. Note that pennant and snap are not similar to each other, as expected.

Looking at the COS data from the Skylake system in Figure 8, we see that the similarity between proxy and parent pairs remains good (green), although the magnitude of the angle between the application signatures changes. For instance, the angle between ExaMiniMD and LAMMPS is smaller (more similar), that between miniQMC and QMCPack and SWFFT and HACC is significantly larger, and that between sw4lite and SW4 is essentially the same. Also note that sw4lite, SW4, SWFFT, HACC, pennant, and snap are less divergent from ExaMiniMD and LAMMPS than they were on the Broadwell architecture—this area in the matrix is characterized more by yellow and orange than by red blocks. MiniQMC remains a distinct outlier compared to the other applications, but here QMCPack has more similarity than before (although it is still more different than similar) to sw4lite, SW4, HACC, and pennant. We also can see that on Skylake versus Broadwell, snap is now much more similar to SWFTT, HACC, and pennant, transitioning from red to green and yellow. These changes in similarity between application signatures (i.e., behavior) are all in response to changing architectural constraints when going from Broadwell to Skylake.

Using the similarity data from both the Broadwell and Skylake platforms, we propose a minimal proxy application suite that covers the composite execution behavior across all of the applications and proxies. This is shown in Table 2. This suggests that the behavior of all 10 of the original members of the suite can be represented by these four proxies.

We can use COS to define a minimal set of proxies for each component represented by a

	ExaMiniMD	LAMMPS	MiniQMC	QMCPack	sw4lite	sw4	SWFFT	HACC	pennant	snap
ExaMiniMD	0.00	10.24	84.61	83.55	61.94	64.17	86.71	85.58	75.88	44.50
LAMMPS	10.24	0.00	75.12	73.95	53.63	56.50	79.66	78.51	70.97	34.97
MiniQMC	84.61	75.12	0.00	5.97	42.91	47.75	51.57	51.28	66.16	43.41
QMCPack	83.55	73.95	5.97	0.00	37.71	42.28	45.85	45.52	60.31	40.89
sw4lite	61.94	53.63	42.91	37.71	0.00	6.47	27.99	26.86	30.17	24.55
sw4	64.17	56.50	47.75	42.28	6.47	0.00	23.59	22.42	23.83	29.89
SWFFT	86.71	79.66	51.57	45.85	27.99	23.59	0.00	1.22	18.65	51.79
HACC	85.58	78.51	51.28	45.52	26.86	22.42	1.22	0.00	18.14	50.70
pennant	75.88	70.97	66.16	60.31	30.17	23.83	18.65	18.14	0.00	51.63
snap	44.50	34.97	43.41	40.89	24.55	29.89	51.79	50.70	51.63	0.00

Figure 7: Broadwell Cosine Similarity in Degrees, Full Vector

	ExaMiniMD	LAMMPS	MiniQMC	QMCPack	sw4lite	sw4	SWFFT	HACC	pennant	snap
ExaMiniMD	0.00	8.97	81.96	68.83	38.66	39.55	28.51	37.76	43.58	22.20
LAMMPS	8.97	0.00	81.38	68.47	38.60	39.33	29.50	38.49	42.40	20.45
MiniQMC	81.96	81.38	0.00	16.35	47.28	47.63	58.78	49.85	46.58	65.55
QMCPack	68.83	68.47	16.35	0.00	36.05	36.40	46.19	37.82	36.33	53.30
sw4lite	38.66	38.60	47.28	36.05	0.00	4.05	20.56	17.09	12.89	21.69
sw4	39.55	39.33	47.63	36.40	4.05	0.00	19.82	15.87	11.91	22.79
SWFFT	28.51	29.50	58.78	46.19	20.56	19.82	0.00	10.33	24.49	21.44
HACC	37.76	38.49	49.85	37.82	17.09	15.87	10.33	0.00	19.92	26.67
pennant	43.58	42.40	46.58	36.33	12.89	11.91	24.49	19.92	0.00	25.00
snap	22.20	20.45	65.55	53.30	21.69	22.79	21.44	26.67	25.00	0.00

Figure 8: Skylake Cosine Similarity in Degrees, Full Vector

Table 2: Potential Minimal Proxy Application Suites

Minimal Proxy App Suite
ExaMiniMD
miniQMC
sw4lite
snap

	ExaMiniMD	LAMMPS	MiniQMC	QMCPack	sw4lite	sw4	SWFFT	HACC	pennant	snap
ExaMiniMD	0.00	6.21	10.96	13.31	13.83	13.85	12.73	12.37	14.25	5.75
LAMMPS	6.21	0.00	7.10	7.82	8.74	8.77	8.07	7.17	8.39	1.58
MiniQMC	10.96	7.10	0.00	10.34	11.83	11.87	12.24	10.90	8.38	6.11
QMCPack	13.31	7.82	10.34	0.00	1.66	1.70	2.86	1.70	3.36	8.91
sw4lite	13.83	8.74	11.83	1.66	0.00	0.05	1.82	1.72	4.81	9.93
sw4	13.85	8.77	11.87	1.70	0.05	0.00	1.79	1.74	4.86	9.96
SWFFT	12.73	8.07	12.24	2.86	1.82	1.79	0.00	1.56	6.18	9.40
HACC	12.37	7.17	10.90	1.70	1.72	1.74	1.56	0.00	4.85	8.44
pennant	14.25	8.39	8.38	3.36	4.81	4.86	6.18	4.85	0.00	9.02
snap	5.75	1.58	6.11	8.91	9.93	9.96	9.40	8.44	9.02	0.00

Figure 9: Skylake COS in Degrees, Branch Behavior

	ExaMiniMD	LAMMPS	MiniQMC	QMCPack	sw4lite	sw4	SWFFT	HACC	pennant	snap
ExaMiniMD	0.00	5.02	54.54	38.73	11.70	12.49	6.58	6.38	13.21	7.13
LAMMPS	5.02	0.00	54.69	38.62	15.66	16.27	4.87	6.38	13.60	10.88
MiniQMC	54.54	54.69	0.00	17.15	47.12	46.08	50.02	48.98	42.16	49.15
QMCPack	38.73	38.62	17.15	0.00	32.64	31.67	33.92	32.94	26.29	33.78
sw4lite	11.70	15.66	47.12	32.64	0.00	1.15	13.41	11.40	11.15	5.07
sw4	12.49	16.27	46.08	31.67	1.15	0.00	13.74	11.70	10.69	5.69
SWFFT	6.58	4.87	50.02	33.92	13.41	13.74	0.00	2.24	9.09	8.80
HACC	6.38	6.38	48.98	32.94	11.40	11.70	2.24	0.00	7.86	6.87
pennant	13.21	13.60	42.16	26.29	11.15	10.69	9.09	7.86	0.00	9.37
snap	7.13	10.88	49.15	33.78	5.07	5.69	8.80	6.87	9.37	0.00

Figure 10: Broadwell COS in Degrees, Cache Behavior

corresponding performance group. For example, we can determine which proxy(ies) to use when interested solely in memory behavior. This can be useful when vendors or other researchers are developing or investigating architectural features pertaining to a specific component. Figure 9 shows COS for branch behavior on our Intel Skylake platform. We see here that all 10 applications and proxies have very similar branch behavior. In this case, we suggest using only one (maybe two) proxies to cover the space of behavior. We would choose either snap or {snap, ExaMiniMD} for branch behavior studies.

Figure 10 shows cosine similarity for cache behavior on our Intel Broadwell platform. We see here that although all proxy behavior is very similar to that of respective parent behavior, miniQMC and QMCPack have starkly different cache behavior compared to all of the other applications. In this case, it would be unwise to use miniQMC as the only representative of application behavior and we would probably want to choose miniQMC and one other proxy (it does not really matter which, so we might choose the one with shortest execution time) to represent the cache behavior of the entire application suite.

Although we do not show all of the performance group COS data here, it is similar to that above in that some applications have very different behavior than others for certain components in the processor architecture. COS enables us to quantitatively choose a minimal set of proxies that represent the behavior of the entire application suite.

3.4 Conclusion

We present here a simple and intuitive methodology based on cosine similarity to understand the extent to which a proxy represents its parent in the context of performance on the underlying hardware. We apply this to 4 ECP proxy/parent pairs on two different Intel platforms and show that all of the proxies are good representatives of their parents across both platforms with respect to node behavior (memory and compute). We also show that we can define a minimal set of proxy applications that represent the behavior of a larger suite using cosine similarity. Finally, we show that cosine similarity can be used to understand which proxies are good for representing behavior of particular architectural components such as cache or memory. By classifying all of a processor's performance events into groups (we call performance groups) that count events on particular architecture components, we can see which are good proxies for certain component behaviors and we can define a minimal set of proxies that cover the behavior space for each particular component. We show that for branching behavior, one proxy can be used to represent the behavior of all 10 members of our suite.

3.5 Future Work

Work for the next FY on assessing proxy representativeness will focus on the following:

- Classify hardware performance counter events of IBM Power9 processor into performance groups to do cosine similarity.
- Further develop our selectivity metric for determining which performance events are most relevant to distinguish applications. Compare this to principal components analysis.
- Develop a more quantitative method for imposing a scale on cosine similarity angles that determines good versus acceptable versus poor similarity between applications.
- Apply cosine similarity to our application suite where for each application, we execute either a completely different problem and/or a different input. The goal is to understand if a change of problem/input results in very different behavior.
- Add more proxy/parent pairs to our analysis.
- Apply cosine similarity to GPU performance for proxies and parents that have GPU implementations.

4 Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear Security Administration under Contract DE-AC52-07NA27344.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Oak Ridge National Laboratory is managed by UT-Battelle, LLC, for the U.S. Department of Energy Under contract DE-AC05-00OR22725.

Los Alamos National Laborary is operated by Triad National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under Contract No. 89233218CNA000001.

Argonne National Laboratory is managed by UChicago Argonne LLC for the U.S. Department of Energy under contract DE-AC02-06CH11357.

The Lawrence Berkeley National Laboratory is a national laboratory managed by the University of California for the U.S. Department of Energy under Contract Number DE-AC02-05CH11231.

References

[1] *ECP Proxy Applications*, 2020. URL: https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/.

[2] O. Aaziz, C. Vaughan, J. Cook, J. Cook, J. Kuehn, and D. Richards. Fine-grained analysis of communication similarity between real and proxy applications. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 93–102, 2019.

[3] Minh Duc Cao and Xiaoying Gao. Combining contents and citations for scientific document classification. In Shichao Zhang and Ray Jarvis, editors, *AI 2005: Advances in Artificial Intelligence*, pages 143–152, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[4] Ramesh Nallapati, Daniel McFarland, and Christopher Manning. Topicflow model: Unsupervised learning of topic-specific influences of hyperlinked documents. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 543–551, 2011.

[5] R. Wagh and D. Anand. Application of citation network analysis for improved similarity index estimation of legal case documents : A study. In *2017 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC)*, pages 1–5, 2017.