OAK RIDGE National Laboratory

IRIS Scheduling Experiments with DAGGER

IRIS Mini Workshop CY22

Beau Johnston
Oak Ridge National Laboratory

4 Jan 2022

ORNL is managed by UT-Battelle, LLC for the US Department of Energy

U.S. DEPARTMENT OF ENERGY

https://csmd.ornl.gov/group/programming-systems

johnstonbe@ornl.gov

IRIS

- The Intelligent Runtime System (IRIS) is a powerful tool, it enables:
  - Multiple languages and backends to be used in a common runtime.
  - Queues can be shared between devices. (Task queue to Device queue).
  - It promotes a genuine heterogeneous workflow — Systems with multiple accelerators from different vendors exist, but runtimes to leverage them historically lacking.
  - It handles dependencies between tasks
  - manages underlying language implementations of tasks

A whole can of worms!

- However this exposes the additional complexity around how to schedule! How do we:
  - Target an optimal device for a given task? (given increasingly heterogeneous systems)
  - Memory transfers still expensive (since most heterogenous systems utilize PCI-E to communicate with devices), this latency is high and implies memory transfers should be avoided.
  - As applications have more complex workflows (the chains of tasks to schedule gets longer) the more complexity we have to manage.

DAGGER

- Directed Acyclic Graph Generator for Evaluating Runtimes
- Simple Python Program which generates a task-graph of arbitary length and complexity
- Generated DAGs plug directly into IRIS (as JSON)
- Used to validate and test different schedulers, and generally stress-test IRIS

Usage:

```
DAGGER: Directed Acyclic Graph Generator for Evaluating Runtimes

optional arguments:
  -h, --help            show this help message and exit
  --kernels KERNELS     The kernel names --in the current directory-- to
                        generate tasks, presented as a comma separated value
                        string e.g. 'process,matmul'
  --kernel-split KERNEL_SPLIT
                        The percentage of each kernel being assigned to the
                        task, presented as a comma separated value string e.g.
                        '80,20'.
  --depth DEPTH         Depth of tree, e.g. 10.
  --num-tasks NUM_TASKS
                        Total number of tasks to build in the DAG, e.g. 100.
  --min-width MIN_WIDTH
                        Minimum width of the DAG, e.g. 1.
  --max-width MAX_WIDTH
                        Maximum width of the DAG, e.g. 10.
  --cdf-mean CDF_MEAN   Mu of the Cumulative Distribution Function, default=0
  --cdf-std-dev CDF_STD_DEV
                        Sigma^2 of the Cumulative Distribution Function,
                        default=0.2
  --skips SKIPS         Maximum number of jumps down the DAG levels (Delta)
                        between tasks, default=1
```

Sample Usage:

To generate a linear/sequential series of tasks---5 tasks long (excluding start and stop nodes), with the sole kernel function called `process`:

```
./dagger_generator.py --kernels="process" --kernel-split='100' --depth=5 --num-tasks=5 --min-width=1 --max-width=1
```

The kernel-split argument specify the probability (as a percentage) of the corresponding kernel name being used.

Figure 1: CLI DAGGER usage.

Quick Compositions (I)

```
In [77]: ! ./dagger_generator.py --kernels="process" --kernel-split='100' --depth=5 --num-tasks=5 --min-width=1 --max-width=1

         from IPython.display import Image
         Image("./dag.png", width=600, height=600)

         done

Out[77]:
```

Figure 2: Demo of using DAGGER to change parameters.

Quick Compositions (II)

- Each task in the DAG to represent a kernel task.
- Changing the `min` and `max` widths can increase the opportunities for concurrency:

```
In [52]: ! ./dagger_generator.py --kernels="process" --kernel-split='100' --depth=1 --num-tasks=5 --min-width=5 --max-width=5

         #from IPython.display import IFrame
         #IFrame("./dag.pdf", width=600, height=600)
         from IPython.display import Image
         Image("./dag.png", width=600, height=600)

         done

Out[52]:
```

Figure 3: Demo of using DAGGER to change parameters to increase potential concurrency.

Quick Compositions (III)

- Support for multiple different kernel tasks—each task is coloured uniquely according to the kernel name.

```
In [36]: ! ./dagger_generator.py --kernels="process,foo,bar" --kernel-split='50,35,15' --depth=100 --num-tasks=50 --min-width=1 --max-width=25

         #from IPython.display import IFrame
         #IFrame('./dag.pdf', width=600, height=600)
         from IPython.display import Image
         Image('./dag.png', width=600, height=600)

         done

Out[36]:
```

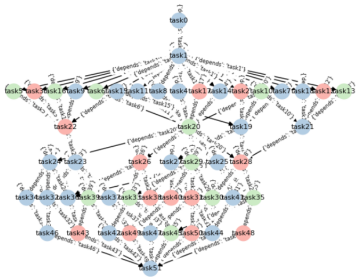


Figure 4: Demo of using DAGGER to change parameters to increase potential concurrency.

Quick Compositions (IV)

- There is an option to provide a mean and standard-deviation to the Cumulative Density Function to change the shape of the DAG, for instance:

```
In [45]: ! ./dagger_generator.py --kernels="process" --kernel-split='100' --depth=25 --num-tasks=100 --min-width=1 --max-width=5 --cdf-mean=1 --cdf-std-dev=1
         from IPython.display import Image
         Image("./dag.png", width=600, height=600)

         done

Out[45]:
```

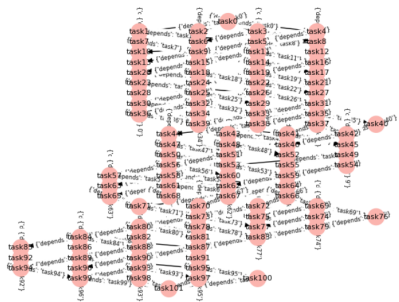


Figure 5: Demo of using DAGGER to change CDF parameters to change shape.

Quick Compositions (V)

- The number of skips allows us to skip up to the maximum number of levels between tasks to have as a dependency, for instance:

```
In [47]: ! ./dagger_generator.py --kernels='process' --kernel-split='100' --depth=25 --num-tasks=100 --min-width=1 --max-width=25 --cdf-mean=2 --cdf-std-dev=5 --skips=5
         from IPython.display import Image
         Image('./dag.png', width=600, height=600)

         done

Out[47]:
```

Figure 6: Demo of using DAGGER to increase levels between dependencies.

DAGGER to IRIS

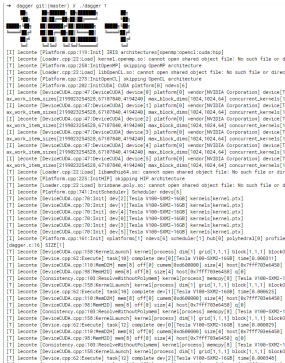


Figure 7: DAGGER generated output running in IRIS.

DAGGER Recap

- DAGGER gives us a simple tool to generate different DAGs to examine the performance of (and stress-test) IRIS.
- Let's apply some interesting payloads...

Experimental Systems

System	GPU Vendor	GPU Series	# of GPUs	Runtimes
Oswald00	Nvidia	P100	1	CUDA
				OpenCL
Equinox	Nvidia	V100	4	CUDA
				OpenCL
Leconte	Nvidia	V100	6	CUDA
Radeon	AMD	Vega20	1	HIP
				OpenCL
Explorer	AMD	M160	2	HIP
				OpenCL

Figure 8: ExCL Systems running IRIS.

The Case for a Smart IRIS Scheduling Policy

- Consider one of the simplest DAGs:

Figure 9: Simple Linear-10 DAG.

- When run on IRIS...

The Case for a Smart IRIS Scheduling Policy (continued)

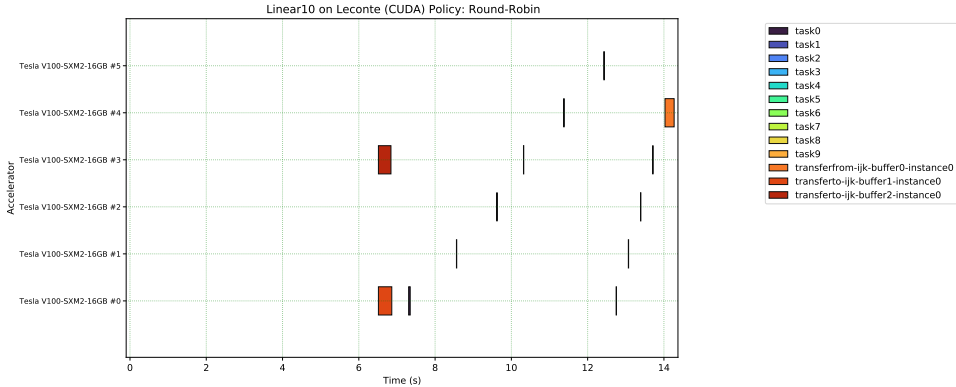


Figure 10: Roundrobin Linear-10 DAG (filtered).

The Case for a Smart IRIS Scheduling Policy (continued)

- Argh Memory Transfers!

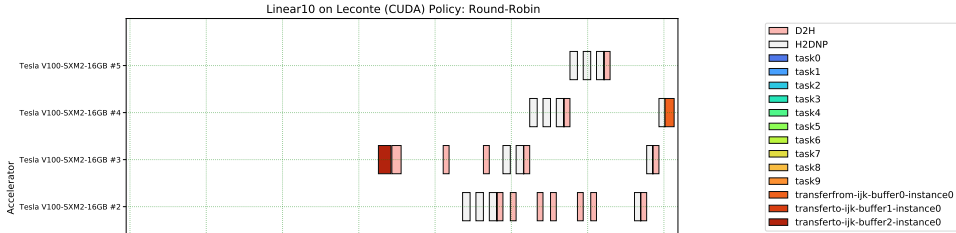


Figure 11: Roundrobin Linear-10 DAG (less filtered).

IRIS Scheduling Policies

IRIS built-in policies include:

1) All,
2) Any,
3) Random,
4) Profiling-Aware
5) Locality-Aware
6) Device number/type

Choosing the right policy for the job...

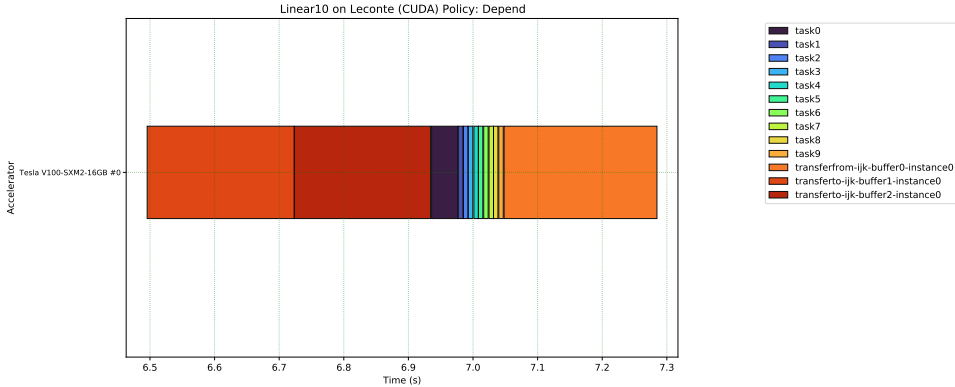


Figure 12: Linear-10 DAG with a dependency policy.

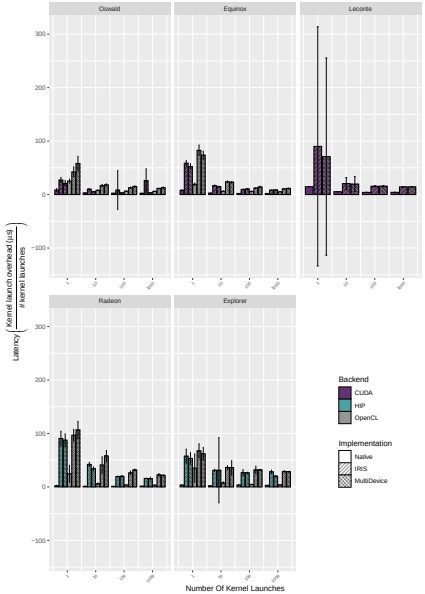
There can't be only one!

- Just one example of how different DAG structures generated from DAGGER will favor different built-in IRIS policies.
- For brevity, I omit studies into all the IRIS systems and Scheduling policies—lots of permutations!
- **But**, can we avoid having the user specify the policy?
  - It requires them know the shape of the DAG,
  - Type of operations (each kernel performs/which accelerator type should be used),
  - Available accelerators and general system configuration (this become increasingly system specific),
  - Scales poorly (human error and manual) as we move IRIS codes between multiple systems,
  - Size of the memory?
- What would we need to consider to have a truly *smart* policy?

Factors for Smart IRIS Scheduling Policies

- Aaron's *HUNTER* can validate correctness of IRIS scheduler trace, and serve as oracle for scheduling time(s).
- Baseline system performance would need be collected to determine relative score.
  - Task latency
  - Accelerator performance over the system (Peak GFLOPs)
  - Memory transfer-time (or latency and bandwidth)
- Prediction of execution time of each kernel/task run
  - Could be addressed with the performance policy (but requires a kernel be run on all available accelerators for the initial comparison)
  - AIWC & Random-Forest/ML methods could be applied
- Thankfully, complexity of dependencies already preserved (in task-dag), so at runtime we know how immutable task-chain/description.
  - How far can we reliably look-ahead for scheduling?

Microbenchmarks – Task Latency



Microbenchmarks – System Performance (Peak GFLOPs)

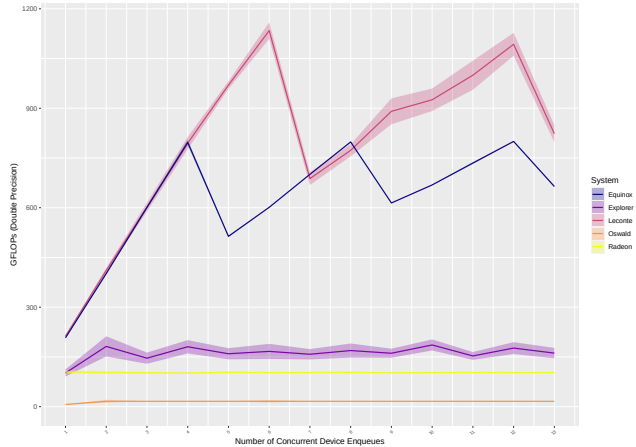
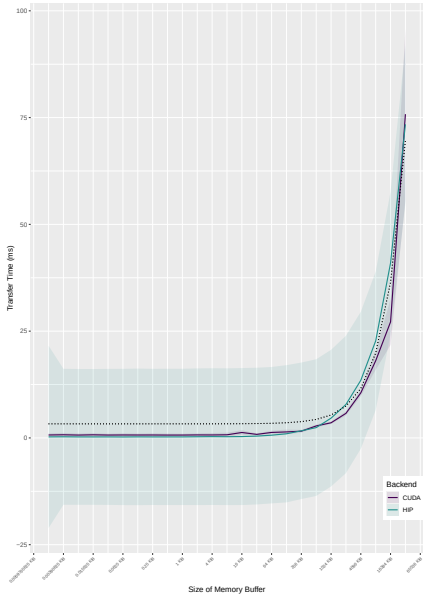


Figure 14: Performance Scaling from increasing devices on a compute-intensive DGEMM kernel.

Microbenchmarks – Transfer Time



Prototype – Prediction per Task

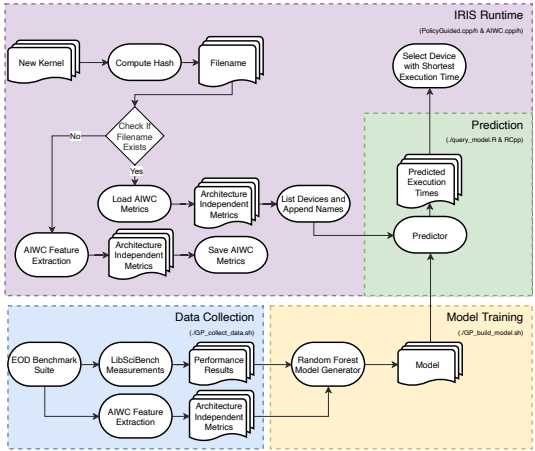


Figure 16: Workflow of performance predictions in IRIS.

Conclusions

- IRIS is a powerful tool, it enables portability of codes originally written for specific accelerators—also complex iterations between these tasks.
- DAGGER can generate synthetic payloads to test IRIS (stress and correctness), and develop scenarios to handicap specific policies.
- Prebuilt policies are a necessary step in evaluating IRIS—but there may be a better option! Let's use these to see!

Future Work

- Comprehensive study to show shortcomings of each built-in scheduler policy—yet to determine the best DAG for each policy.
- In IRIS, broader tasks can be composed of multiple (sub)commands, such as, memory transfers and kernel executions. Do we treat these as "hints" as largely atomic/indivisible chunks of work which should be completed on a single accelerator?
- Apply AIWC and Random-Forest predictive policy (to see relative performance).

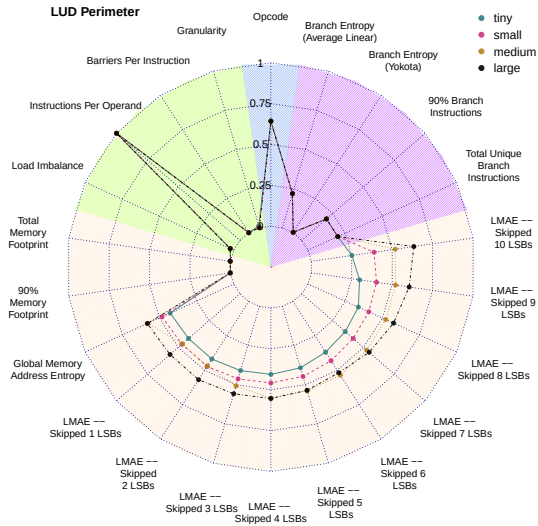
Workload characterization with AIWC

- Architecture-Independent Workload Characterization (AIWC)
- Plugin for OclGrind – an Extensible OpenCL device simulator[1]
- Beta available – https://github.com/ANU-HPC/Oclgrind – and will be merged into default OclGrind
- Simulation of OpenCL kernels occur on LLVM IR – SPIR
- AIWC tracks and measures hardware agnostic events
- Metrics carefully selected and collected during simulator execution
- Large number of metrics collected (28)
- Over a wide spectrum computation, thread communication and memory access patterns
- Supports parallel workloads
- Accessible – as part of OclGrind
- High-accuracy – full resolution, not interrupt/sample driven

[1] J. Price and S. McIntosh-Smith, "Oclgrind: An extensible opencl device simulator," in Proceedings of the 3rd International Workshop on OpenCL, 2015, p. 12.

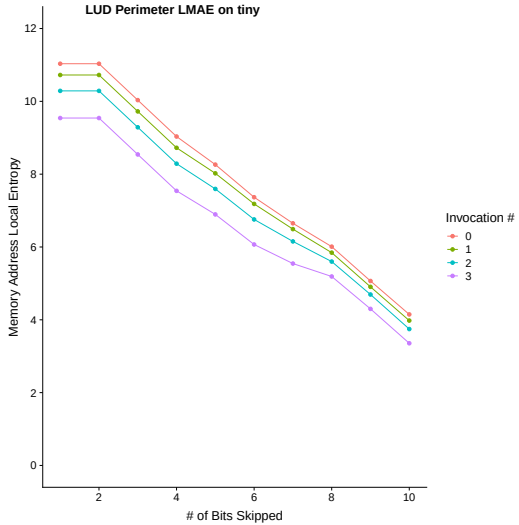
AIWC Example

- Four major classes: Compute, Parallelism, Memory, Control
- Different statistics per metric – distributions, entropy and absolute counts

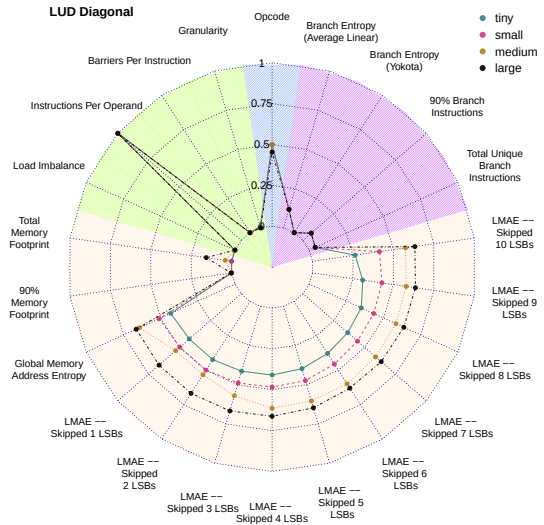


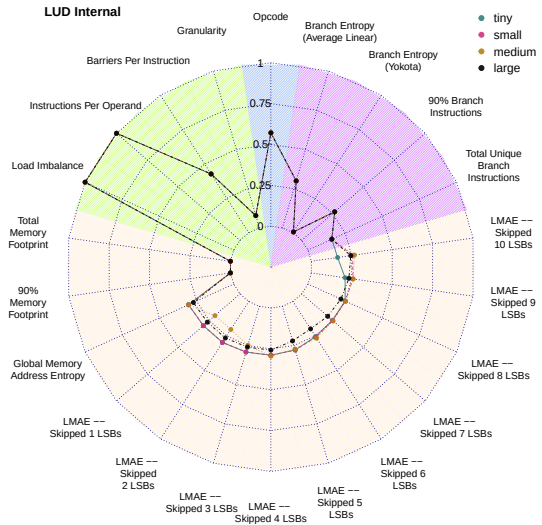
AIWC Example II

- Local Memory Address Entropy
- Kernel launched 4 times – over different problem sizes
- Starting entropy changes with problem size, but same gradient → memory access patterns are the same regardless of actual problem size
- Steeper descent → more localised memory access → better cache utilization



AIWC Example III





Subset of AIWC Metrics

Table 1: Sample of metrics collected by AIWC – ordered by type.

Type	Metric	Description
Compute	Opcode	total # of unique opcodes required to cover 90% of dynamic instructions
Compute	Total Instruction Count	total # of instructions executed
Parallelism	Work-items	total # of work-items or threads executed
Parallelism	Total Barriers Hit	total # of barrier instructions
Parallelism	Median ITB	median # of instructions executed until a barrier
Parallelism	Max IPT	maximum # of instructions executed per thread
Parallelism	Mean SIMD Width	mean # of data items operated on during an instruction
Memory	Total Memory Footprint	total # of unique memory addresses accessed
Memory	90% Memory Footprint	# of unique memory addresses that cover 90% of memory accesses
Memory	Unique Read/Write Ratio	indication of workload being (unique reads / unique writes)
Memory	Reread Ratio	indication of memory reuse for reads (unique reads/total reads)
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	total # of unique branch instructions
Control	90% Branch Instructions	# of 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

AIWC Usage

```
oclgrind --aiwc ./kmeans -p 0 -d 0 -t 0 -- -g -p 256 -f 30
```

- The collected metrics are logged as text in the command line interface during execution and also in a csv file, stored separately for each kernel and invocation.
- Files can be found in the working directory with the naming convention `aiwc_`α`_`β`.csv`.
- Where α is the kernel name and β is the invocation count – the number of times the kernel has been executed.

AIWC Overheads

Table 2: Overhead of the **AIWC** tool on the `fft` benchmark and the Intel i7-6700K CPU.

	time			memory		
	usage (ms)		increase	usage (MB)		increase
	without AIWC	with AIWC		without AIWC	with AIWC	
tiny	0.04	73.4	≈1830×	80.0	85.9	1.07×
small	0.2	427.8	≈2800×	75.9	149.0	1.96×
medium	2.9	12420	≈4300×	101.4	636.8	6.28×
large	19.6	69300	≈3540×	203.8	2213.2	10.86×

- large problems, may exmaine fewer iterations, or run on machines with more virtual memory.
- Envisaged use is that AIWC is only run once, for instance, to examine the characteristics of the kernel in order to identify suitability for accelerators or verify that a high degree of SIMD vectorization had been achieved.

AIWC Example IV

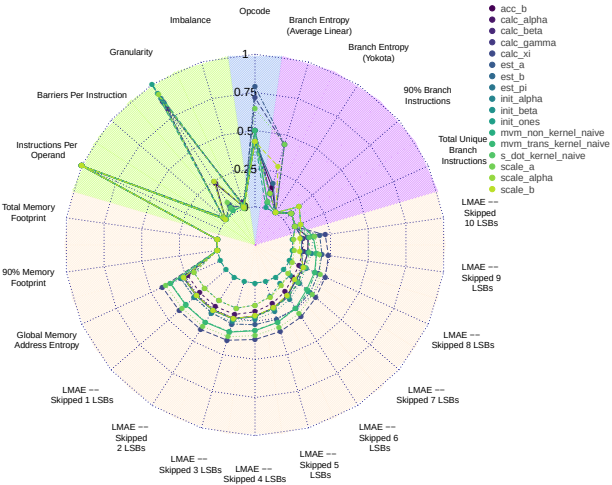
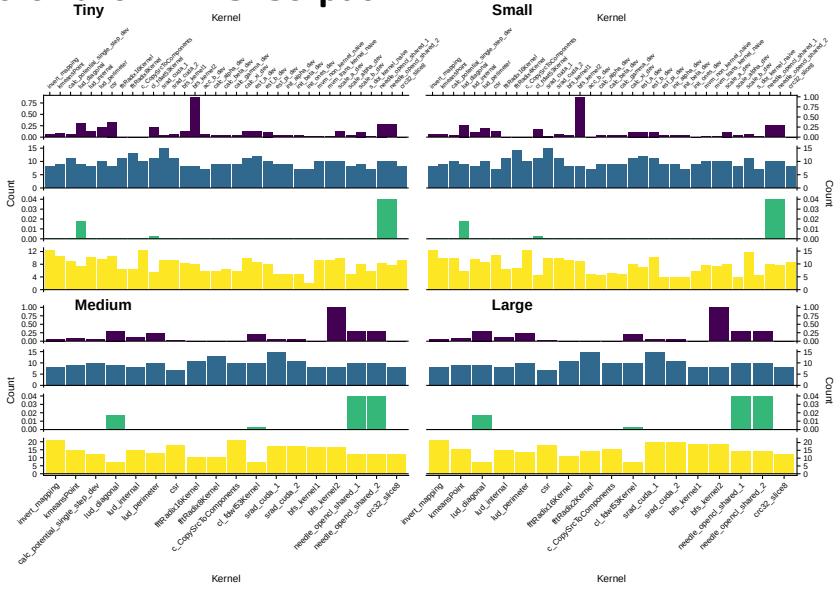


Figure 21: Highlighting the variance in features between kernels in the hmm benchmark.

A larger AIWC Corpus

- Just a sample of 2 of 11 applications
- Similar breakdown of 37 more kernels from the remainder of EOD suite
- Presented in a docker & jupyter artefact

Sample of the AIWC Corpus



OAK RIDGE National Laboratory

Itemize Test

- What happens now?
  - if we use sublists?
  - and other things...
- Test again
  - and now
    - one more unto the breach,
    - dear friends,
    - once more

1) What happens now?
   1) if we use sublists?
   2) and other things...
2) Test again
   1) and now – why worry about counting yourself?
      1) blah
      2) blah
      3) blah