# Collaborative design and optimization using Collective Knowledge

Anton Lokhmotov[1] and Grigori Fursin[1,2]

[1] dividiti, UK [2] cTuning foundation, France

**Abstract.** Designing faster, more energy efficient and reliable computer systems requires effective collaboration between hardware designers, system programmers and performance analysts, as well as feedback from system users. We present Collective Knowledge (CK), an open framework for reproducible and collaborative design and optimization. CK enables systematic and reproducible experimentation, combined with leading edge predictive analytics to gain valuable insights into system performance. The modular architecture of CK helps engineers create and share entire experimental workflows involving modules such as tools, programs, data sets, experimental results, predictive models and so on. We encourage a wide community, including system engineers and users, to share and reuse CK modules to fuel R&D on increasing the efficiency and decreasing the costs of computing everywhere.

## 1 Introduction

### 1.1 The need for collaboration

Designing faster, more energy efficient and reliable computer systems requires effective collaboration between several groups of engineers, for example:

- *hardware designers* develop and optimize hardware, and provide low-level tools to analyze their behavior such as simulators and profilers with event counters;
- *system programmers* port to new hardware and then optimize proprietary or open-source compilers (e.g. LLVM, GCC) and libraries (e.g. OpenCL,[1] OpenVX,[2] OpenCV,[3] Caffe,[4] BLAS[5]);
- *performance analysts* collect benchmarks and representative workloads, and automate running them on new hardware.

In our experience, the above groups still collaborate infrequently (e.g. on achieving development milestones), despite the widely recognized virtues of hardware/software co-design [1]. Moreover, the effectiveness of collaboration typically

[1] Khronos Group's standard API for heterogeneous systems: `khronos.org/opencl`
[2] Khronos Group's standard API for computer vision: `khronos.org/openvx`
[3] Open library for computer vision: `opencv.org`
[4] Open library for deep learning: `caffe.berkeleyvision.org`
[5] Standard API for linear algebra: `netlib.org/blas`

depends on the proactivity and diligence of individual engineers, the level of investment into collaboration tools, the pressure exerted by customers and users, and so on. Ineffective collaboration could perhaps be tolerated many decades ago when design and optimization choices were limited. Today systems are so complex that any seemingly insignificant choice can lead to dramatic degradation of performance and other important characteristics [2,3,4,5]. To mitigate commercial risks, companies develop proprietary infrastructures for testing and performance analysis, and bear the associated maintenance costs.

For example, whenever a performance analyst reports a performance issue, she should provide the program code along with instructions for how to build and run it, and the experimental conditions (e.g. the hardware and compiler revisions). Reproducing the reported issue may take many days, while omitting a single condition in the report may lead to frustrating back-and-forth communication and further time being wasted. Dealing with a performance issue reported by a user is even harder: the corresponding experimental conditions need to be elicited from the user (or guessed), the program code and build scripts imported into the proprietary infrastructure, the environment painstakingly reconstructed, etc.

Ineffective collaboration wastes precious resources and runs the risk of designing uncompetitive computer systems.

### 1.2 The need for representative workloads

The conclusions of performance analysis intrinsically depend on the workloads selected for evaluation [6]. Several companies devise and license benchmark suites based on their guesses of what representative workloads might be in the near future. Since benchmarking is their primary business, their programs, data sets and methodology often go unchallenged, with the benchmarking scores driving the purchasing decisions both of OEMs (e.g. phone manufacturers) and consumers (e.g. phone users). When stakes are that high, the vendors have no choice but to optimize their products for the commercial benchmarks. When those turn out to have no close resemblance to real workloads, the products underperform.

Leading academics have long recognized the need for representative workloads to drive research in hardware design and software tools [7,8]. With funding agencies increasingly requiring academics to demonstrate impact, academics have the right incentives to share representative workloads and data sets with the community.

Incentives to share representative workloads may be somewhat different for industry. Consider the example of Realeyes,[6] a participant in the EU CARP project.[7] Recognizing the value of collaborative R&D, Realeyes released under a permissive license a benchmark comprised of several standard image processing algorithms used in their pipeline for evaluating human emotions [9]. Now

[6] `realeyesit.com`
[7] `carpproject.eu`

Realeyes enjoy the benefits of our research on run-time adaptation (§3) and accelerator programming ([10]) that their benchmark enabled.

We thus have reasons to believe that the expert community can tackle the issue of representative workloads. The challenge for vendors and researchers alike will be to keep up with the emerging workloads, as this will be crucial for competitiveness.

### 1.3 The need for predictive analytics

While traditionally performance analysts would only obtain benchmarking figures, recently they also started performing more sophisticated analyses to detect unexpected behavior and suggest improvements to hardware and system software engineers. Conventional labour-intensive analysis (e.g. frame by frame, shader by shader for graphics) is not only extremely costly but is simply unsustainable for analyzing hundreds of real workloads (e.g. most popular mobile games).

Much of success of companies like Google, Facebook and Amazon can be attributed to using statistical ("machine learning", "predictive analytics") techniques, which allow them to make uncannily accurate predictions about users' preferences. Whereas most people would agree with this, the same people would resist the idea of using statistical techniques in their own area of expertise. A litmus test for our community is to ask ten computer engineers whether statistical techniques would help them design better processors and compilers. In our own experience, only one out of ten would say yes, while others would typically lack interdisciplinary knowledge.

We have grown to appreciate the importance of statistical techniques over the years. (One of us actually flunked statistics at university.) We constantly find useful applications of predictive analytics in computer engineering. For example, identifying a minimal set of representative programs and inputs has many benefits for design space exploration, including vastly reduced simulation time.

### 1.4 Our humble proposal for solution

We present Collective Knowledge, a simple and extensible framework for collaborative and reproducible R&D ([11], §2). With Collective Knowledge, engineers can systematically investigate design and optimization choices using leading edge statistical techniques, conveniently exchange experimental workflows across organizational boundaries (including benchmarks), and automatically maintain programming tools and documentation.

Several performance-oriented open-source tools exist including LLVM's LNT,[8] ARM's Workload Automation,[9] and Phoronix Media's OpenBenchmarking.[10] These tools do not provide, however, robust mechanisms for reproducible experimentation and capabilities for collaborative design and optimization. We

[8] `llvm.org/docs/lnt`

[9] `github.com/ARM-software/workload-automation`

[10] `openbenchmarking.org`

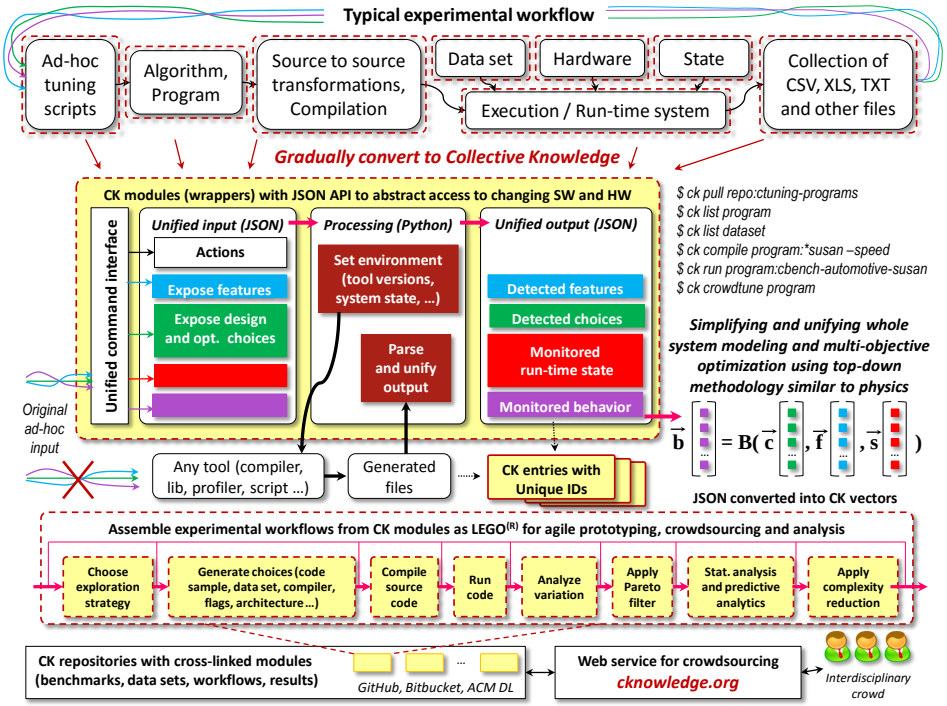


**Fig. 1.** Converting a typical experimental workflow to the Collective Knowledge format.

demonstrate some of these mechanisms and capabilities on a computationally intensive algorithm from the Realeyes benchmark (§3). We believe that Collective Knowledge can be combined with open-source and proprietary tools to create robust, cost-effective solutions to accelerate computer engineering.

## 2 Collective Knowledge

Fig. 1 shows how a typical experimental workflow can be converted into a collection of CK modules such as programs (e.g. benchmarks), data sets, tools (e.g. compilers and libraries), scripts, experimental results, predictive models, articles, etc. In addition, CK modules can abstract away access to hardware, monitor run-time state, apply predictive analytics, etc.

Each CK module has a class. Classes are implemented in Python, with a JSON[11] meta description, JSON-based API, and unified command line interface. New classes can be defined as needed.

Each CK module has a DOI-style unique identifier (UID). CK modules can be referenced and searched through by their UIDs using Hadoop-based Elas-

[11] JavaScript Object Notation: `json.org`

ticsearch.[12] CK modules can be flexibly combined into experimental workflows, similar to playing with LEGO® modules.

Engineers can share CK workflows complete with all their modules via repositories such as GitHub. Other engineers can reproduce an experiment under the same or similar conditions using a single CK command. Importantly, if the other engineers are unable to reproduce an experiment due to uncaptured dependencies (e.g. on run-time state), they can "debug" the workflow and share the "fixed" workflow back (possibly with new extensions, experiments, models, etc.)

Collaborating groups of engineers are thus able to gradually expose in a unified way multi-dimensional design and optimization choices **c** of all modules, their features **f**, dependencies on other modules, run-time state **s** and observed behavior **b**, as shown in Fig. 1 and described in detail in [12,13]. This, in turn, enables collaboration on the most essential question of computer engineering: how to optimize any given computation in terms of performance, power consumption, resource usage, accuracy, resiliency and cost; in other words, how to learn and optimize the behavior function $B$:

$$\mathbf{b} = B(\mathbf{c}, \mathbf{f}, \mathbf{s})$$

### 2.1 Systematic benchmarking

Collective Knowledge supports systematic benchmarking of a program's performance profile under reproducible conditions, with the experimental results being aggregated in a local or remote CK repository. Engineers gradually improve reproducibility of CK benchmarking by implementing CK modules to set run-time state and monitor unexpected behavior across participating systems.

For example, on mobile devices, unexpected performance variation can often be attributed to dynamic voltage and frequency scaling (DVFS). Mobile devices have power and temperature limits to prevent device damage; in addition, when a workload's computational requirements can still be met at a lower frequency, lowering the frequency conserves energy. Further complications arise when benchmarking on heterogeneous multicore systems such as ARM big.LITTLE: in a short time, a workload can migrate between cores having different microarchitectures, as well as running at different frequencies. Controlling for such factors (or at least accounting for them with elementary statistics) is key to meaningful performance evaluation on mobile devices.

## 3 Example

Systematically collecting performance data that can be trusted is essential but does not by itself produce insights. The Collective Knowledge approach permits to seamlessly apply leading edge statistical techniques on the collected data, thus converting "raw data" into "useful insights".

[12] Open-source distributed real-time search and analytics: `elastic.co`

| **Platform** | **CPU** (ARM) | **GPU** (ARM) |
|---|---|---|
| Chromebook 1 | Cortex-A15×2 | Mali-T604×4 |
| Chromebook 2 | Cortex-A15×4 | Mali-T628×4 |

**Table 1.** Experimental platforms: Samsung Chromebooks 1 (XE303C12, 2012) and 2 (XE503C12, 2014). Notation: "processor architecture" × "number of cores".

Consider the Histogram of Oriented Gradients (HOG), a widely used computer vision algorithm for detecting objects [14]. Realeyes deploy HOG in several stages of their image processing pipeline. Different stages use different "flavours" of HOG, considerably varying in their computational requirements. For example, one stage of the pipeline may invoke HOG on a small-sized image but with a high amount of computation per pixel ("computational intensity"); another stage, may invoke HOG on a medium-sized image but with low computational intensity. In addition, the Realeyes pipeline may be customized differently for running on mobile devices (e.g. phones), personal computers (e.g. laptops) or in the cloud.

In this paper, we use two versions of HOG: an OpenCV-based CPU implementation (with TBB parallelization) and a hand-written OpenCL implementation (data parallel kernel).[13] Suppose we are interested in optimizing the execution time of HOG.[14] Computing HOG on the GPU is typically faster than on the CPU. The *total* GPU execution time (including the memory transfer overhead), however, may exceed the CPU execution time.

Figure 2 shows a performance surface plot for one flavour of HOG with DVFS disabled and the processors' frequencies controlled for. The $X$ and $Y$ axis show the CPU and the GPU frequencies, while the $Z$ axis shows the CPU execution time divided by the *total* GPU execution time. When this ratio is greater than 1 (the light pink to bright red areas), using the GPU is faster than using the CPU, despite the memory transfer overhead. A sensible scheduling decision, therefore, is to schedule the workload on the GPU.

While it may be possible to infer when to use the GPU from this plot (just avoid the light blue to navy areas), what if the performance also depends on other factors as well as the processors' frequencies? Will we still be able to make sensible scheduling decisions most of the time?

To answer this question, we conducted multiple experiments with HOG ($1 \times 1$ cells) on two Chromebook platforms (see Table 1). The experiments covered the Cartesian product of the CPU and GPU frequencies available on both platforms (CPU: 1600 MHz, 800 MHz; GPU: 533 MHz, 266 MHz), 3 block size (16, 64, 128), 23 images (in different shapes and sizes), for the total of 276 samples (with 5 repetitions each).

[13] The related CK repository is at `github.com/ctuning/reproduce-carp-project`.

[14] We can also consider multi-objective optimization e.g. finding appropriate trade-offs between execution time vs. energy consumption vs. cost.

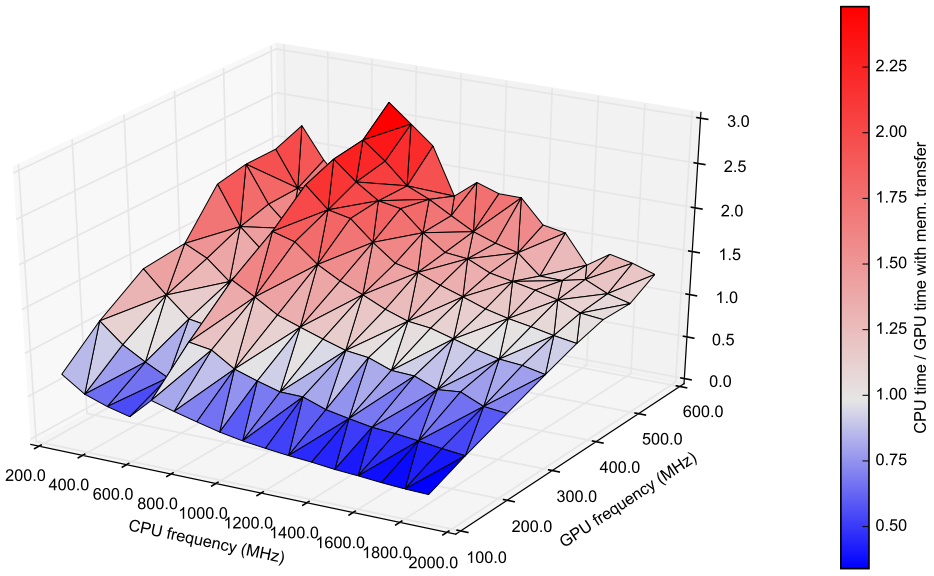


**Fig. 2.** Platform: Chromebook 2. Program: HOG $4 \times 4$; block size: 64. $X$ axis: CPU frequency (MHz); $Y$ axis: GPU frequency (MHz); $Z$ axis: CPU execution time divided by GPU [kernel + memory transfer] execution time.

To analyze the collected experimental data, we use decision trees, a popular supervised learning method for classification and regression.[15] We build decision trees using a Collective Knowledge interface to the Python `scikit-learn` package.[16] We thus obtain a predictive model that tells us whether it is faster to execute HOG on the GPU or on the CPU by considering several features of a sample (experiment). In other words, the model classifies a sample by assigning to it one of the two labels: "YES" means the GPU should be used; "NO" means the CPU should be used. We train the model on the experimental data, by labelling a sample with "YES" if the CPU execution time exceeds the GPU execution time by at least 7% (to account for variability), and with "NO" otherwise.

Figure 3 shows a decision tree of depth 1 built from the experimental data obtained on Chromebook 1 using just one feature: the block size (designated as 'worksize' in the figure), which, informally, determines the computational intensity of the algorithm. The root node divides the training set of 276 samples into two subsets. For 92 samples in the first subset, represented by the left leaf node ("L1"), the worksize is less than or equal to 40 (i.e. 16). For 184 samples in the second subset, represented by the right leaf node ("L2"), the worksize is greater than 40 (i.e. 64 and 128).

[15] `en.wikipedia.org/wiki/Decision_tree_learning`
[16] `scikit-learn.org`

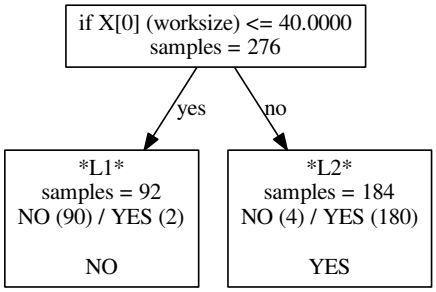


**Fig. 3.** Platform: Chromebook 1. Model: feature set: 1; depth: 1.

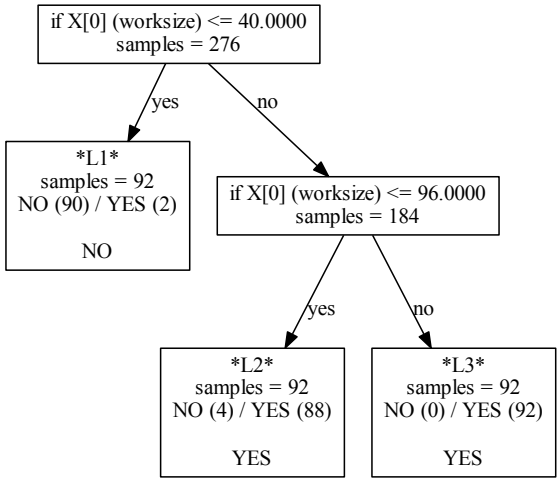


**Fig. 4.** Platform: Chromebook 1. Model: feature set: 1; depth: 2.

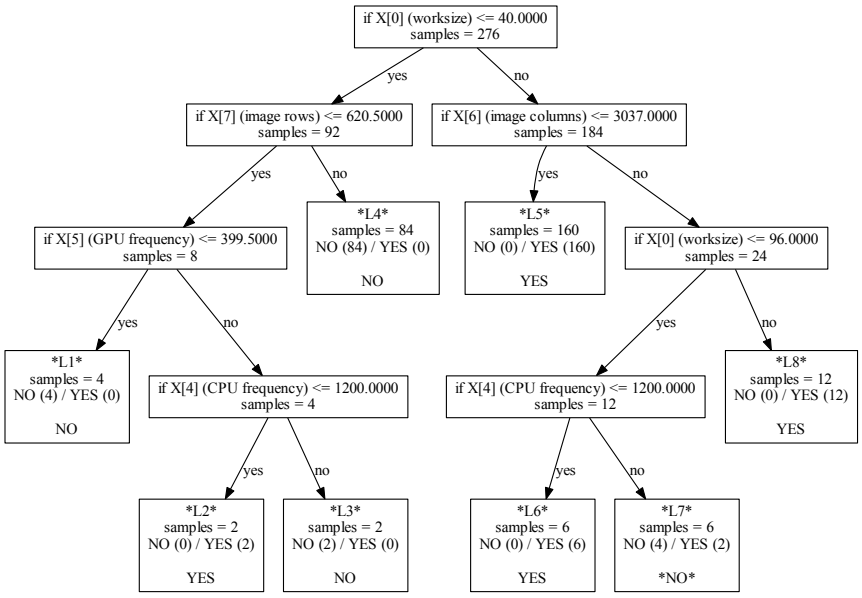


**Fig. 5.** Platform: Chromebook 1. Model: feature set: 2; depth: 4.

| Id | Features |
|---|---|
| FS1 | worksize [block size] |
| FS2 | all features from FS1, CPU frequency, GPU frequency, image rows ($m$), image columns ($n$), image size ($m \times n$), (GWS0, GWS1, GWS2) [OpenCL global work size] |
| FS3 | all features from FS2, CPU frequency / GPU frequency, image size / CPU frequency, image size / GPU frequency, |

**Table 2.** Feature sets: simple (FS1); natural (FS2); designed (FS3).

In the first subset, 90 samples are labelled with "NO" and 2 samples are labelled with "YES". Since the majority of the samples are labelled with "NO", the tree predicts that the workload for which the worksize is less than or equal to 40 should be executed on the CPU. Similarly, the workload for which the worksize is greater than 40 should be executed on the GPU. Intuitively, this makes sense: the workload with a higher computational intensity (a higher value of the worksize) should be executed on the GPU, despite the memory transfer overhead.

For 6 samples out of 276, the model in Figure 3 mispredicts the correct scheduling decision. (We say that the rate of correct predictions is 270/276 or 97.8%.) For example, for the two samples out of 92 in the subset for which the worksize is 16 ("L1"), the GPU was still faster than the CPU. Yet, based on labelling of the majority of the samples in this subset, the model mispredicts that the workload should be executed on the CPU.

Figure 4 shows a decision tree of depth 2 using the same worksize feature. The right child of the root now has two children of its own. All the samples in the rightmost leaf ("L3") for which the worksize is greater than 96 (i.e. 128) are labelled with "YES". This means that at the highest computational intensity, the GPU was always faster than the CPU, thus confirming our intuition. However, the model in Figure 4 still makes 6 mispredictions. To improve the prediction rate, we build models using more features, as well as having more levels. In Table 2, we consider two more sets of features.

The "natural" set is constructed from the features that we expected would impact the scheduling. Figure 5 shows a decision tree of depth 4 built using the natural feature set. This model uses 4 additional features (the GPU frequency, the CPU frequency, the number of image columns, the number of image rows) and has 8 leaf nodes, but still results in 2 mispredictions ("L7"), achieving the prediction rate of 99.3%. This model makes the same decision on the worksize at the top level, but better fits the training data at lower levels. However, this model is more difficult to grasp intuitively and may not fit new data well.

The "designed" set can be used to build models achieving the 100.0% prediction rate. A decision tree of depth 5 (not shown) uses all the new features from the designed set. With 12 leaf nodes, however, this model is even more difficult to grasp intuitively and exhibits even more overfitting than the model in Figure 5.

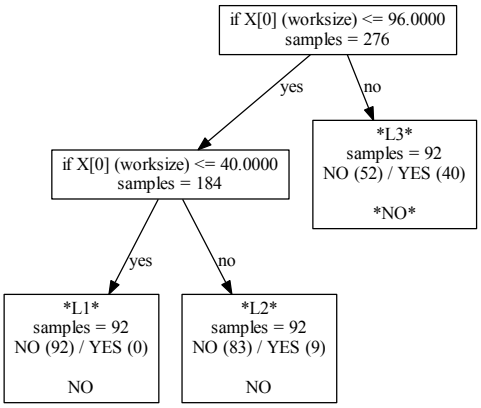


**Fig. 6.** Platform: Chromebook 2. Model: feature set: 1; depth: 2.

Now, if we use a simple model trained on data from Chromebook 1 (Figure 4) for predicting scheduling decisions on Chromebook 2, we only achieve a 51.1% prediction rate (not shown). A similar model retrained on data from Chromebook 2 (Figure 6) achieves a 82.3% prediction rate. Note that the top-level decision has changed to the worksize being less than 96. In other words, up to that worksize the CPU is generally faster than the GPU even as problems become more computationally intensive. This makes sense: the CPU of Chromebook 2 has 4 cores, whereas the CPU of Chromebook 1 has 2 cores. This demonstrates the importance of retraining models for different platforms.

As before, using more features and levels can bring the prediction rate to 100.0%. For example, using the natural feature set improves the prediction rate to 90.2% (Figure 7). Note that the top-level decision no longer depends on the worksize but on the first dimension of the OpenCL global work size.

For brevity, we omit a demonstration of the importance of using more data for training. For example, to build more precise models, we could have added experiments with the worksize of 32 to determine if that would still be considered non-intensive as the worksize of 16. The Collective Knowledge approach allows to crowdsource such experiments and rebuild models as more mispredictions are detected and more data becomes available.

High-level programming frameworks for heterogeneous systems such as Android's RenderScript,[17] Qualcomm's Symphony,[18] and Khronos's OpenVX[19] can be similarly trained to dispatch tasks to system resources efficiently.

[17] `developer.android.com/guide/topics/renderscript`
[18] `developer.qualcomm.com/symphony` (formerly known as MARE)
[19] `khronos.org/openvx`

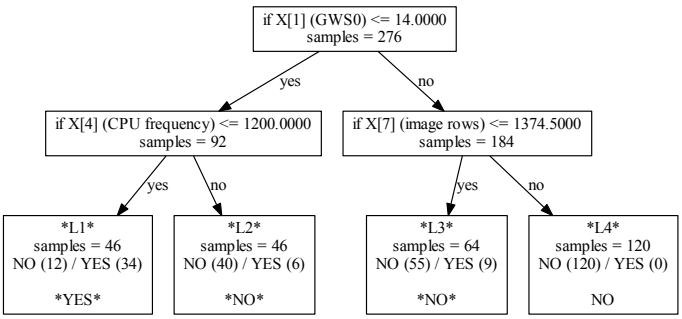


**Fig. 7.** Platform: Chromebook 2. Model: feature set: 2; depth: 2.

## 4 Conclusion

We have presented Collective Knowledge, an open methodology that enables collaborative design and optimization of computer systems. This methodology encourages contributions from the expert community to avoid common benchmarking pitfalls (allowing, for example, to fix the processor frequency, capture run-time state, find missing software/hardware features, improve models, etc.)

### 4.1 Representative workloads

We believe the expert community can tackle the issue of representative workloads as well as the issue of rigorous evaluation. The community will both provide representative workloads and rank them according to established quality criteria. Furthermore, a panel of recognized experts could periodically (say, every 6 months) provide a ranking to complement commercial benchmark suites.

The success will depend on establishing the right incentives for the community. As the example of Realeyes shows, even when commercial sensitivity prevents a company from releasing their full application under an open-source license, it may still be possible to distill a performance-sensitive portion of it into a standalone benchmark. The community can help the company to optimize their benchmark (for free or for fee), thus improving the overall performance of their full application.[20] Some software developers will just want to see their benchmark appear in the ranked selection of workloads, highlighting their skill and expertise (similar to "kudos" for open-source contributions).

### 4.2 Predictive analytics

We believe that the Collective Knowledge approach convincingly demonstrates that statistical techniques can indeed help computer engineers do a better job

[20] The original HOG paper [14] has over 12500 citations. Just imagine this community combining their efforts to squeeze out every gram of HOG performance across different "flavours", data sets, hardware platforms, etc.

in many practical scenarios. Why do we think it is important? Although we are not suggesting that even most advanced statistical techniques can ever substitute human expertise and ingenuity, applying them can liberate engineers from repetitive, time-consuming and error-prone tasks that machines are better at. Instead, engineers can unleash their creativity on problem solving and innovating. Even if this idea is not particularly novel, Collective Knowledge brings it one small step closer to reality.

### 4.3 Trust me, I am a catalyst!

We view Collective Knowledge as a catalyst for accelerating knowledge discovery and stimulating flows of reproducible insights across largely divided hardware/-software and industry/academia communities. Better flows will lead to breakthroughs in energy efficiency, performance and reliability of computer systems. Effective knowledge sharing and open innovation will enable new exciting applications in consumer electronics, robotics, automotive and healthcare—at better quality, lower cost and faster time-to-market.

## 5 Acknowledgements

We thank the EU FP7 609491 TETRACOM Coordination Action for funding initial CK development. We thank the CK community for their encouragement, support and contributions. In particular, we thank our partners and customers for providing us valuable opportunities to improve Collective Knowledge on real-world use cases.

## References

1. J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, May 2012.
2. John L. Hennessy and David A. Patterson. *Computer architecture, a quantitative approach (second edition)*. Morgan Kaufmann publishers, 1995.
3. R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the Conference on High Performance Networking and Computing*, 1998.
4. B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, P.M.W. Knijnenburg, M.F.P O'Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E.A. Stöhr, M. Verhoeven, and H.A.G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In *Proc. Euro-Par 97*, volume 1300 of *Lecture Notes in Computer Science*, pages 1351–1356, 1997.
5. K.D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1), 2002.
6. Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. May 1991.

7. Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.
8. Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham Riley, Nigel Topham, and Steve Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*, May 2015. arXiv:1410.2167.
9. Elnar Hajiyev, Róbert Dávid, László Marák, and Riyadh Baghdadi. Realeyes image processing benchmark. `https://github.com/Realeyes/pencil-benchmarks-imageproc`, 2011–2015.
10. Riyadh Baghdadi, Ulysse Beaugnon, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Javed Absar, Sven van Haastregt, Alexey Kravets, Robert David, Elnar Hajiyev, Adam Betts, Jeroen Ketema, Albert Cohen, Alastair Donaldson, and Anton Lokhmotov. PENCIL: a platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT'15)*, September 2015.
11. Grigori Fursin, Anton Lokhmotov, and Ed Plowman. Collective Knowledge: towards R&D sustainability. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'16)*, March 2016.
12. Grigori Fursin, Renato Miceli, Anton Lokhmotov, Michael Gerndt, Marc Baboulin, D. Malony, Allen, Zbigniew Chamski, Diego Novillo, and Davide Del Vento. Collective Mind: Towards practical and collaborative auto-tuning. *Scientific Programming*, 22(4):309–329, July 2014.
13. Grigori Fursin, Abdul Memon, Christophe Guillon, and Anton Lokhmotov. Collective Mind, Part II: Towards performance- and cost-aware software engineering as a natural science. In *Proceedings of the 18th International Workshop on Compilers for Parallel Computing (CPC'15)*, January 2015.
14. Navneet Dalal and Triggs Bill. Histograms of oriented gradients for human detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 886–893, 2005.