Application Suitability Assessment for Many-Core Targets

Chris J. Newburn, Jim Sukha(✉), Ilya Sharapov, Anthony D. Nguyen, and Chyi-Chang Miao

Intel Corporation, Hudson, USA
{chris.newburn,jim.sukha,ilya.sharapov,anthony.d.nguyen, chyi-chang.miao}@intel.com

Abstract. Many-core hardware platforms offer a tremendous opportunity for scaling up performance, but not all codes that run on these platforms have been modernized sufficiently to fully utilize the hardware. Assessing whether a code will effectively utilize a given platform can be challenging, particularly for new or potential future platforms where native execution on real hardware is not possible. In this case, one typically relies on architecture simulators and other workload characterization tools, which are often not user-friendly for developers who want to do a quick initial assessment of an application's suitability for a many-core architecture.

To help address this challenge, we present QMSprof, a tool and a set of analyses for an initial assessment of the suitability of a set of applications for a simulated extremely-parallel many-core target. QMSprof automates the process of running a suite of workload binaries through Intel® Software Development Emulator (SDE) and the Sniper multi-core simulator and extracting high-level summary statistics. The tool generates comparative plots summarizing key metrics across the workload suite, including the mix of vector and nonvector instructions, scalability with increasing thread count, memory bandwidth utilization, and statistics on cache misses and working set size. These summary metrics are designed to aid performance tuners in selecting promising codes for a many-core target and in pinpointing opportunities for additional tuning. To illustrate the utility of our tool, we also describe some sample results from characterizing applications on a hypothetical many-core architecture.

Keywords: Many-core · Performance · Characterization · Code modernization

1 Introduction

Not all applications are cut out for execution on extremely-parallel machines like those of the Intel® Xeon Phi^{TM} Processor Family [9], also known as the

This research was, in part, funded by the U.S. Government and DOE. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

"Knights" family. Such machines offer high levels of thread parallelism, vector parallelism and bandwidth. If an application fails to exploit these salient features, its performance will fall far short of the capabilities of these machines.

Those who seek to characterize applications on new architectures often lack the time for detailed analysis, or the expertise, or both. Application developers today typically rely on profiling and analysis tools, such as Intel® VTune^{TM} Amplifier (or VTune^{TM} for short)[8] or Intel® Advisor [6] to understand application behavior on existing hardware. When real hardware is not available however, e.g., because it is a future architecture under development, one typically must resort to using a combination of workload characterization and simulation tools. Unfortunately, since these tools are usually designed for hardware architects rather than application developers, they can overwhelm the uninitiated user with raw statistics, and are not user-friendly for a developer who is looking for a quick high-level profile of their application on a future architecture.

This paper describes *QMSprof*, a tool and set of analyses for assessing how well-suited an application is for extremely-parallel many-core targets, such as the Intel Xeon Phi product line. The name **QMSprof** stems from its implementation, because it derives its results from other **Q**uick **M**ultithreaded **S**imulation and **prof**iling tools. QMSprof runs a suite of workload binaries through two underlying tools, Intel® Software Development Emulator (Intel® SDE) [7] and the Sniper multi-core simulator [2,14], aggregates results, and produces high-level analysis and summary plots. This analysis is designed to provide a high-level summary of application characteristics, enabling users to more easily determine which workloads may be more suitable than others for a many-core architecture. The results produced from QMSprof are expected to be useful as part of an initial triage of applications being brought into tuning sessions, highlighting key characteristics that may warrant deeper analysis or additional optimization.

In this paper, we describe the application of QMSprof and analyses to regions of interest (ROIs) that are marked as important, from eight workloads of general interest to the HPC community, simulated on a hypothetical many-core architecture. The focus of the paper is to demonstrate the *kinds* of analyses supported by QMSprof and what the results may *look* like, not the specific numbers for any workload or architecture. Thus, one should **not** interpret any numerical results on our simulated architectural model in this paper as corresponding to absolute performance on any existing or future Intel silicon. However, we expect the *tools and methodology* to be of interest to those porting their applications to current and future Intel® Xeon Phi^{TM} Processor many-core machines such as Knights Landing (KNL), as well as wide Intel® Xeon Processor machines.

2 Overview of Tool and Analysis

QMSprof streamlines the process of running a suite of workloads through the Intel Software Development Emulator (SDE) and the Sniper simulator, extracting high-level summary statistics, and generating comparative plots. This section describes the interface and operation of two components of QMSprof, namely the data collector which generates raw SDE and Sniper data for a particular experiment, and the data analyzer which extracts and summarizes the results.

2.1 Data Collector

The *data collector* is a Python module that generates run scripts for a set of SDE and Sniper runs, sweeping across a suite of workloads, Sniper configurations, and different thread counts. Users must provide the following inputs to the collector:

1. **Workload binaries and run parameters**: For each workload in the desired suite, a binary compiled with a special begin and end markers around a single region of interest, and any required input files and command-line arguments.
2. **Sniper Models**: Configuration files for different Sniper models to be run.
3. **Environment Setup**: Other configuration parameters specific to the user's desired run environment (e.g., paths to specific OpenMP runtime libraries or configuration for job managers in a cluster environment).
4. **Experiment Script**: The description of the particular subset of workloads, models, and thread counts to run in a particular experiment.

These four configuration inputs are separable and can be specified mostly independently by different area experts. Workload binaries, input files, and run parameters are typically provided by application experts who are familiar with regions of interest to profile. Sniper models can be provided by hardware architects familiar with specifying relevant architecture parameters in the simulator. The environment setup file can be specified by individuals familiar with details of the installed run environment. Finally, the experiment script is specified by the end user who wants to run particular experiments.

Users configure QMSprof by describing inputs as Python dictionaries and lists, a user interface that is both relatively human-readable and suitable for automation. A detailed example of the interface is illustrated in Appendix A.

Once all the inputs have been specified, the collector takes the list of runs specified in the experiment script and generates a shell script for each Sniper and/or SDE run. These shell scripts can be run directly at a command prompt, or fed into a job manager in a distributed compute environment. QMSprof generates shell scripts, rather than invoking Sniper and SDE directly, because this intermediate step facilitates debugging. When runs fail, one can simply manually edit a generated run script and debug an individual run, without trying to repeat an entire sweep of experiments. In our prototype, we set up QMSprof to run on Intel NetBatch, an internal distributed computing environment which has been used for many years for simulations and other compute-intensive jobs [1,13,21]. It is straightforward, however, to extend QMSprof to submit jobs to other publicly available job managers such as SLURM [19].

QMSprof uses the following Intel SDE and Sniper execution modes to collect raw statistics:

1. **SDE Instruction Mix.** SDE provides an instruction mix tool that produces statistics on the numbers and types of instructions executed.

2. **SDE Footprint Tool.** SDE also provides a tool that estimates the memory footprint of a profiled region in a program, both at cache-line and page granularity. More specifically, SDE tracks the distinct cache lines or pages accessed in a region of interest, and can classify them as either data or code.
3. **Sniper Simulation.** Sniper is an execution-driven high-speed x86 simulator which can be used to characterize a workload by executing the workload on a generic many-core configuration.

Sniper runs slower than SDE, but it collects additional raw statistics that are useful for our analysis. QMSprof adds two flags to each Sniper run to gather additional information: `--profile` to collect information on function calls, including a percentage breakdown of instructions and time spent inside and outside of the OpenMP library, and `--cheetah` to profile the working set of threads executing in the region of interest, using a known technique for efficient simulation of multiple cache sizes in a single run [15].

2.2 Data Analysis

The *data analyzer* in QMSprof is a set of scripts that extracts data from an experiment run by the collector, and generates various output summary plots. In particular, QMSprof can generate the following summary data:

1. **Vector Instruction Mix**: A breakdown of the types of vector instructions executed by each run.
2. **Thread Scalability**: Running time of benchmarks variants in the experiment as a function of thread count.
3. **Memory bandwidth**: A measure of the average memory bandwidth utilization of the application.
4. **Cache Miss Statistics**: A plot of the miss rates and misses per 1 K instructions for the last-level cache.
5. **Working set size**: An analysis of the last-level cache sizes needed to achieve a given miss rate.
6. **OpenMP overhead**: The fraction of total execution time spent in the OpenMP runtime, which can indicate overheads from fork/join and dynamic scheduling.
7. **Memory footprint**: A measure of the new number of distinct pages of memory required to execute each workload.

The analyzer produces two forms of output, namely (1) comma-separated value files amenable to import into a spreadsheet, or (2) data and plot files for gnuplot, version 4.6 [18]. In subsequent sections of this paper, we describe the summary plots generated by the analyzer in greater detail and explain why they are useful for understanding the suitability of a workload for a many-core architecture. We also performed an analysis of the granularity of OpenMP parallel regions, but since it is not fully automated, we do not show that here. Table 1 summarizes the execution mode of Sniper or SDE used for each analysis, and identifies comparable analyses from Advisor and VTune, if they exist.

In general, compared to Advisor and VTune, QMSprof is optimized for different purposes. Both Advisor and VTune are primarily designed for characterization of full-scale workloads, running natively on existing hardware, while QMSprof is designed to extract important high-level summary statistics from slower but more detailed simulations of targeted regions of interest in a workload. Native execution has a few limitations compared to a simulation-based approach, which include a lack of accurate FLOPS counting and a limited ability to estimate performance on future hardware that may have different characteristics. QMSprof is able to use Sniper to simulate models of hardware that do not exist today, or even models that are impractical to build but can provide interesting insights through limit studies and other hypothetical what-if scenarios. Similarly, its use of SDE allows for more detailed accounting of dynamically executed instructions which are difficult to do with current hardware. Additionally, although running workloads through Sniper or SDE is slower than native execution, the resulting statistics tend to be less affected by measurement noise and thus more amenable to comparison across workloads.

Table 1. Execution modes from SDE and Sniper used by the analyses in QMSprof. The table also indicates which analyses are supported by the Advisor and VTune tools. A ✓ and ∼ indicate full and partial support, respectively.

Characterization	SDE		Sniper			Advisor	VTune
Mode	-mix	-footprint	Default	–profile	–cheetah		
Vector Instruction Mix	✓					∼	∼
Thread Scalability			✓			✓	
Memory Bandwidth Utilization			✓				✓
Cache Miss Statistics			✓			∼	∼
Working Set Size			✓		✓		
OpenMP Runtime Fraction			✓	✓			✓
Memory Footprint		✓				✓	

As indicated in Table 1, Advisor provides some analyses similar to QMSprof. Intel® Advisor 2016/2017 offers vectorization, FLOPS and roofline analysis capabilities, for both Xeon and Xeon Phi. These capabilities provide per-loop and optionally per-program information on the following data ingredients in ways that are similar to four of the numbered QMSprof features above: (1) static and limited dynamic instruction mixes, (2) thread scalability, (4) memory bandwidth and (7) memory footprint. Advisor analysis is targeted towards end-user complex code modernization, offering insight at the loop/function granularity, with low runtime overhead and multiple data representations and data sources, such as compiler opt-reports, the access pattern profiler, or trip count/FLOPS analysis. Unlike QMSprof, Advisor does not focus on contrasting aggregated program-level characteristics across workloads or platforms. Also, with the exception of Thread Scalability and AVX-512 codepath projection features, it does not model platforms other than currently-available silicon. Instruction mix and footprint data is currently not a first class citizen information in Advisor; it is not always exposed in detail and is sometimes provided with lower accuracy to minimize runtime overhead.

VTune supports many detailed analyses of a single workload, and is generally optimized for a deep dive into the behavior of a few workloads on real silicon, rather than QMSprof which is optimized for a quick initial comparison of select statistics across a large set of workloads. VTune provides information on code performance through several predefined analysis types. For example, VTune includes algorithmic analysis types, such as hotspot analysis and threading concurrency analysis with locks and waits profiling to find synchronization bottlenecks. It also includes microarchitecture analysis types, such as general exploration analysis with a hierarchical organization of event-based metrics for identifying the dominant performance bottlenecks in an application, memory access analysis showing processor stalls by memory hierarchy, memory bandwidth information, and a correlation of memory objects with memory performance metrics. VTune's statistic collection methods include hardware performance monitoring, binary instrumentation, instrumented threading runtimes, and static analysis. VTune covers the following analyses in QMSprof listed above: (1) vector instruction mix based on KNL's limited hardware profiling that are mitigated by static analysis, (3) memory bandwidth including DRAM, MCDRAM and QPI bandwidth types, (4) cache miss statistics, (6) OpenMP serial time, imbalance and overhead with cause, and MPI time spent spinning in active waiting for hybrid MPI + OpenMP applications. VTune is evolving to offer a combination of thread scalability, memory and FPU utilization aspects in one analysis type called HPC Performance Characterization.

2.3 Prototype Implementation

We have implemented a prototype of QMSprof that works with SDE and an Intel-internal version of Sniper. We use an internal version of Sniper primarily because it supports execution of binaries compiled for the AVX512 instruction set, a feature currently not available in the public version of Sniper. Currently, there exists a formal process for developers with appropriate restricted-secret NDA approvals to access the Intel-internal version of Sniper, and our scripts and configuration files can be made available to those who have access to the Intel-internal version of Sniper.

For the empirical results presented in the rest of this paper, since SDE is publicly available, the analyses in QMSprof that are based on SDE can be reproduced by all users. The analysis based on Sniper could also be repeated using the public version of Sniper, using only AVX128 vectors, but this change in Sniper versions would affect the instruction mix and potentially the interactions with the memory system.

3 A Case Study

In the remainder of this paper, we demonstrate QMSprof by applying it to a case study that evaluates 8 HPC workloads on a generic many-core micro-architecture model. This section describes the workloads and the Sniper model used for our case study. Although detailed simulation results are not the focus of this paper, we describe our experimental methodology to provide some context for understanding QMSprof's output.

We tested regions of interest (ROIs) chosen from eight representative HPC workloads: BlackScholes [10], Himeno [4], LULESH [3], miniFE [3], SimpleMOC kernel [16], SNAP [3], SMC [17,20], and WSM5 [5]. We also tested two microbenchmarks: PeakFLOPS, a synthetic kernel created to achieve near maximal performance on floating-point computations, and STREAM [11,12], a synthetic kernel designed to achieve maximal memory bandwidth usage. BlackScholes, Himeno, and SimpleMOC (kernel), are relatively simple kernels, while the remaining codes are regions taken from more complex proxy apps. All codes are strong-scaled with OpenMP.

We spent little or no effort tuning these workloads, instead taking the binaries generated by the compiler mostly as-is. Frankly, scenarios with poorly-tuned workloads are more common than those with extensive tuning and optimization. *The condition of the workloads and the actual conclusions based on the data are not the focus of the paper; instead, the analysis methodology and tools are.* This usage model matches a typical work flow for a performance modeling expert or an architecture expert, who can experiment with changes in architecture, but is often not in a position to optimize workloads.

For each workload, we chose an ROI which executes on a scaled data set for somewhere between 100 to 500 million instructions. This choice is driven in part by the simulation speed of Sniper, which simulated on roughly 0.1 to 1 million instructions per second in our study. This speed is obviously too slow to estimate the performance of a multicore applications on a full production-size input. We believe it is sufficient, however, for understanding the impact of changes in architectural parameters such as cache sizes, prefetchers, out-of-order execution width, etc., provided that workload experts can provide representative scaled-down inputs. The analyses that are based on SDE and Sniper share the same binaries and hence have the same ROI markers. The binaries were compiled with the -xMIC-AVX512 flag using version 16.0 of the Intel compiler.

The SDE-based analyses are target independent. For Sniper, we model a many-core micro-architecture with 16 3-wide, out-of-order cores. Each core has a 32 K L1-D cache, a 32 K L1-I cache, and a vector unit capable of executing AVX512 instructions. Hardware prefetchers are enabled. Each pair of cores share an 1 MB L2. Sixteen cores can access up to 512 MB of in-package memory through a single memory controller, with a bandwidth limit of 48 GB/s. Note that this configuration models only a fraction (e.g., 1/4) of a full many-core die. Although this model does not capture any sharing or contention effects that would exist in an application that requires shared-memory communication between the fractions of the die, it is reasonable for modeling a single rank of an MPI application that uses OpenMP threads to populate 16 cores.

Finally, although we believe our model has many of the salient features of a many-core architecture and is useful for some relative comparisons between workloads, it is important to note that its architectural parameters **intentionally do not match any known product** on the Intel Xeon Phi roadmap. Thus, our model or results **should not be used as estimate of absolute performance on Knights Landing or any other Intel silicon**.

4 Output from the Data Analyzer

In this section, we discuss each QMSprof output for the case study described in Sect. 3, explaining how each analysis can be used to determine the relative suitability of a workload for a many-core architecture.

4.1 Vector Instruction Mix

The first analysis one would typically run is an instruction mix, which shows the vector instructions executed in a workload, expressed as a fraction of the total instructions executed. To get maximal benefit from a many-core architecture like Xeon Phi, we generally want the fraction of vector instructions to be as close to 100 % as possible. Moreover, we ideally prefer to have full-length vector instructions, i.e., AVX512 packed SIMD instructions, rather than shorter-vector or SIMD scalar instructions. We also prefer fewer masked vector instructions, since 0-valued mask bits imply unused vector lanes. If the fraction of vector instructions is low, this is an alert that some vectorization enabling may be required for compiler-generated code, or that vector-enabled libraries may not be in use. If the ratio of scalar SIMD to packed SIMD is high, significant time may being spent in unvectorized outer loops, suggesting a possible need for placing vectorization directives on outer loops. Excessive masking may be mitigated by eliminating conditionals, which is sometimes possible by inlining functions to enable constant propagation by the compiler.

QMSprof automatically generates plots from SDE that reveal packed vs. scalar SIMD, and non-scalar AVX type, as shown in Fig. 1. This analysis is run first because it runs faster on the SDE emulator than on the Sniper simulator. In our case study, we see that only BlackScholes and PeakFLOPS are getting close to having 100 % of vector instructions. Other workloads have a noticeable fraction of shorter vectors (e.g., LULESH, at 70 % short vectors), or non-vector instructions, even though we compiled the workloads targeting AVX512. Finally, there appears to be little use of masking (reported separately from the plot shown) in these workloads, with the highest fraction of 14 % of all instructions for WSM5. Reporting of data types, e.g. single-precision, double-precision and bit-wise SIMD, has also been demonstrated, but is not shown here. This instruction mix analysis is useful in correlating performance differences with and without vectorization (e.g., for the Intel compiler, code compiled

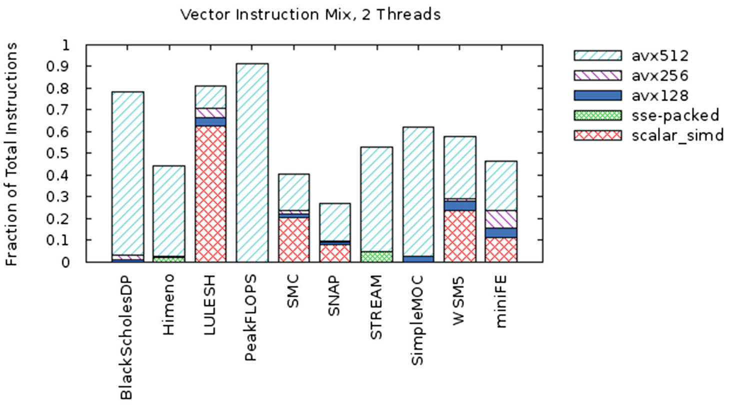

Fig. 1. Instruction mix of workloads, classifying percentages of vector instructions executed. This data was collected using SDE for runs with two OpenMP worker threads.

with the -no-simd -no-vec -no-openmp-simd flags). Since the direct evaluation of that silicon performance is not part of QMSprof, we do not show those results.

4.2 Thread Scalability

Thread scalability analysis can provide guidance on how to balance parallelism between MPI ranks and threads: more MPI ranks may be better when OpenMP thread scaling trails off. Thread serialization may not be noticed, but it tends to kill scalability. The thread scaling efficiency can be inferred from QMSprof's plot of Sniper-based parallel speedups as a function of **thread count** P.

Figure 2 shows a parallel speedup plot produced by QMSprof for our case study. For our workloads, we see that SimpleMOC, SNAP, BlackScholes, and WSM5 are the 4 workloads (ignoring PeakFLOPS) that scale reasonably well by adding more threads, while the others appear to have other limits to scalability. QMSprof can be configured to test scalability on a few different Sniper models with different parameters (e.g., different cache size, bandwidth, etc.). It also produces additional summary plots which help users understand other scalability limiters in greater detail.

Memory Bandwidth and Cache Behavior. Memory bandwidth and cache misses can become bottlenecks that limit thread scaling, and QMSprof produces summary plots for each, as shown in Fig. 3. The lack of thread scaling for Himeno, STREAM and miniFE correlate with the high memory bandwidth that does not scale with threads. High bandwidth is not necessarily an indication of high cache miss rates. BlackScholes has a low cache miss rate, but bandwidth that scales with threads; this is an indication of effective prefetching. We focus on last-level

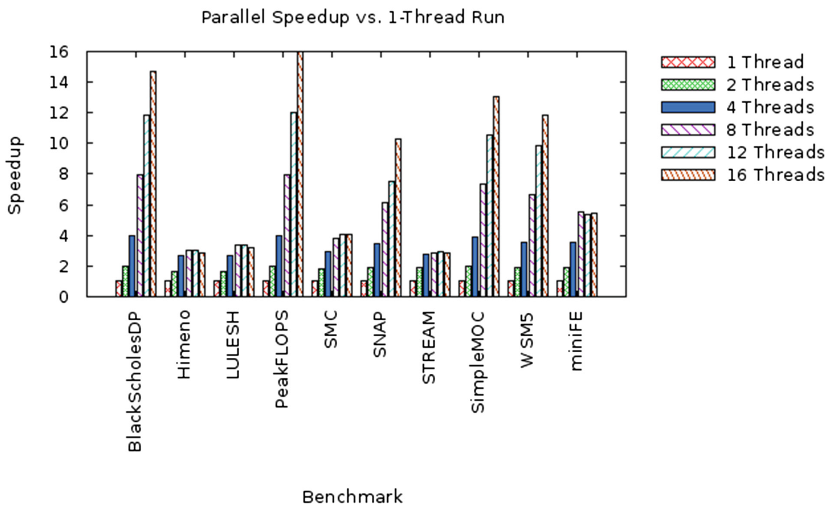

Fig. 2. Parallel speedup of workloads. This plot shows speedup for various numbers of threads from P=1 to P=16, on a simulated 16-core architecture with each pair of cores sharing an L2.

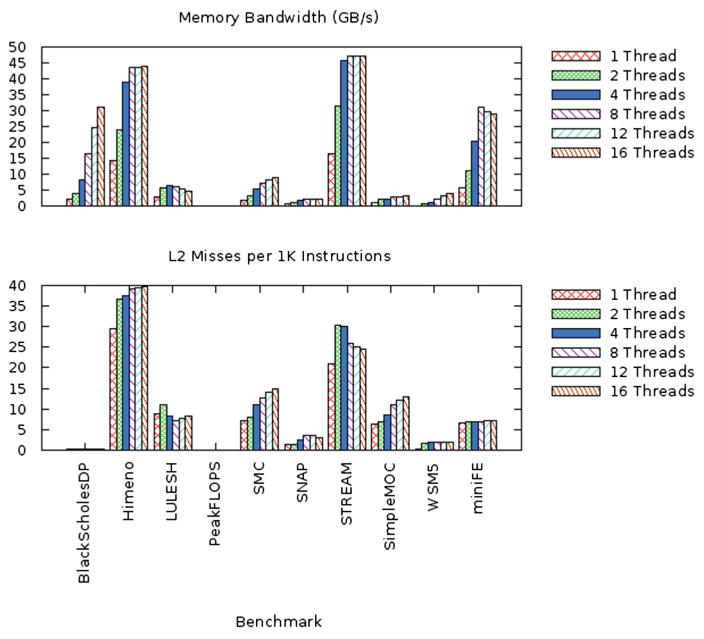

Fig. 3. Average memory bandwidth (top) and demand L2 misses per 1K instructions (bottom) for various thread counts, aligned to the same set of workloads.

cache misses per 1 K instructions executed as a metric instead of absolute miss rates, since it tends to be a better indicator of performance impact.

OpenMP Runtime Overheads. Another thread scaling inhibitor is overheads in both serial and parallel sections in a program. Serial sections, which can come from both an application or a parallel runtime such as OpenMP, do not speed up as more threads are added, and thus naturally limit scalability. Similarly, other OpenMP overheads in a parallel region will not be amortized away if there is too little work within a parallel region and the number of threads is large. QMSprof aggregates profiling data from Sniper to help users estimate such overheads.

Figure 4 shows that LULESH, miniFE, and SimpleMOC are the only workloads which spend more than 5 % of their total execution time in the OpenMP runtime. The implementations of LULESH and miniFE we used are known to have many fine-grained parallel regions, which contribute to fork-join overheads. SimpleMOC had many dynamically-scheduled loops, which contributes to the runtime overhead. In particular, for SimpleMOC, Sniper's profile output indicated that most of the time spent in the OpenMP runtime was in methods for lock acquisition, even for the single-thread runs. The current QMSprof prototype does not distinguish between overheads in serial and parallel sections, although we observed through other analysis that serial sections in the chosen ROIs of these workloads were generally negligible.

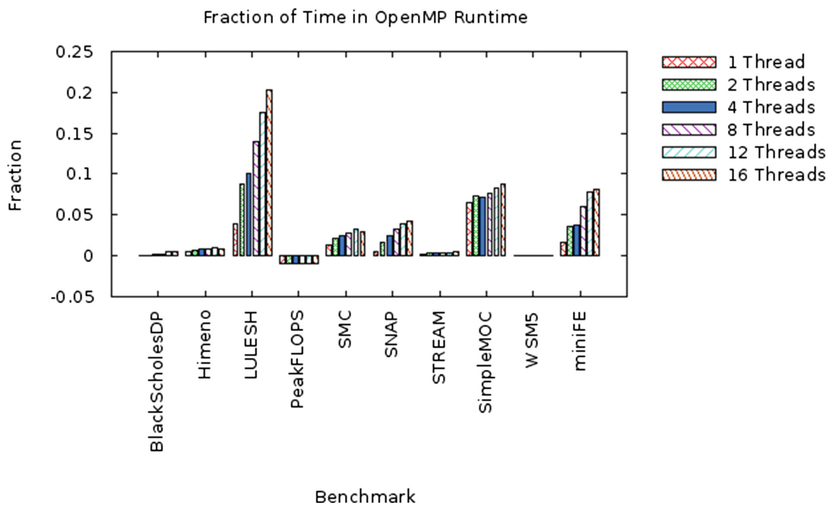

Fig. 4. Fraction of execution time that is overhead in the OpenMP runtime. A negative fraction (e.g., for PeakFLOPS) indicates that OpenMP was not detected at all, which usually means the workload was parallelized using some other approach.

4.3 Working Set Analysis

Thread scaling, analyzed in Sect. 4.2, is impacted by whether there is constructive interference across threads, and what cache capacity is needed per thread to keep cache miss rates low. Decisions about the number of threads to use per core, or per L2, or per set of L2s, or per in-package memory, may be based on the working set per thread and the kind of interference there is across threads. We use Sniper's 'Cheetah' functionality [15] to estimate the working set sizes for the applications, providing a more direct measure of the effect of changing cache sizes than the analysis in Sect. 4.2. Sniper allows us to estimate the cache miss rate for the outer-level cache as a function of cache size, at power-of-2 cache sizes, using a single simulation run. We extracted the minimum cache sizes needed to achieve a given cache miss rate (ranging from 1 % to 20 %), as shown in Fig. 5. This kind of exploratory analysis is not generally available with hardware-based profiling.

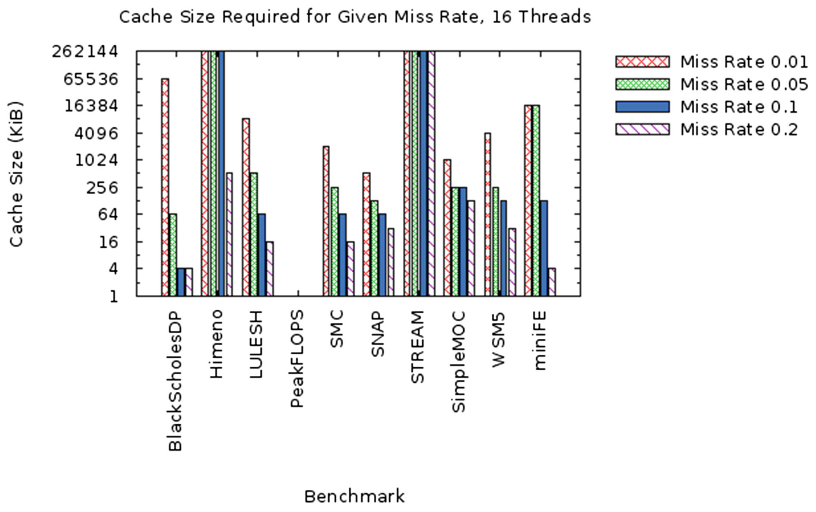

Fig. 5. Minimum cache size required to achieve a given cache miss rate (or less). This plot is extracted from Sniper's Cheetah data.

4.4 Memory Footprint Characterization

An analysis of the memory footprint may help determine how many MPI ranks can share resources like in-package memory such as MCDRAM or high-bandwidth memory (HBM). Large memory footprints cause a capacity issue, which can turn into a performance issue. We use the footprint tool from SDE to count the *total* number of pages accessed at a 2 MB page size in Fig. 6, regardless of caching effects. From this data, we observe that most of the workloads in our study access a total number of pages which is unlikely to fit into a typical L2 cache, and some would fit in a modest amount of in-package memory.

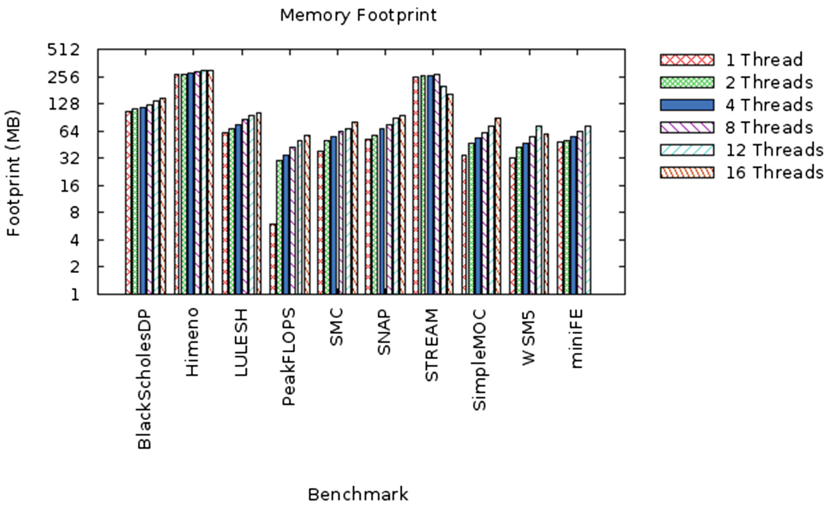

Fig. 6. Memory footprint of workloads, as measured by total memory covered (MB) and estimated using SDE, for accesses at the granularity of 2 MB pages.

Although we observed some correlation between working sets size and memory footprint for the workloads in this study, in general a large memory footprint, as measured above by SDE, does not necessarily indicate a large working set for the workload, since memory that is only accessed once still contributes to the number of new pages. This analysis does not give any good measure of data sharing between threads, since at any fixed value of memory footprint, the partitioning of data between the threads could be arbitrary. For example, we could have complete sharing, with all threads accessing all the data, or complete partitioning, with each thread accessing its own private data.

5 Conclusions

Harvesting thread and vector parallelism are critical to reaching peak performance on extreme-scale execution targets. Yet many applications do not approach peak performance on many-core targets because they lack effective parallelization for threaded and vector architectures, and they lack effective memory tuning. The QMSprof infrastructure provides an initial but broad-ranging analysis that give an indication of how well suited an application is for making good use of a many-core target, how well it can be scaled within that target with respect to memory capacity, MPI rank vs. OpenMP threading trade-offs, threads per core and per L2, whether additional memory blocking may be required, and how well vectorized it is.

While QMSprof does not offer all of the characterization that one could ever ask for, we've demonstrated that it provides a useful beginning. Based on feedback on the perceived utility of this tool and on what additional characterizations might provide the greatest benefit, more features may be added to QMSprof as they are brought to a sufficient degree of automation. Since the primary motivation for QMSprof is for quick initial estimates rather than comprehensive studies, however, some selectivity is desirable to avoid inundating users with too many statistics.

This analysis of suitability could either be used to prioritize codes that are already more suitable over others that are not (yet) suitable, and/or they can be used to focus optimization efforts on making applications more suitable for highly-parallel targets. While the absolute results presented here are not specific to a given target product, and are not intended to be representative of Knights Landing or other such products, the trends and "shape of the curves" is expected to provide some actionable insights to those assessing suitability and doing optimization.

6 Future Work

One characterization that could form an interesting complement to what has been done here is to analyze the effectiveness in trading off number of threads for number of MPI ranks, especially with respect to its impact on the memory system. Another, which is of particular interest for offload, is an analysis of the volume of data that must be communicated between the sequential parts of the program that might be executed on an Intel Xeon with low latency, and the highly-parallel parts that might be executed on a high-throughput Intel Xeon Phi Processor.

Workload statistics, like execution time and cache misses, could be reported for each serial section and OpenMP parallel region by compiling some additional instrumentation markers into the OpenMP runtime. This could help to highlight the significant serial sections and their characteristics.

One might also add silicon-based data collection, and possibly automate the correlation of simulation results with real silicon measurements. Runs on real silicon may enable us to analyze larger problem sizes. One challenge however, is that for the relatively small problems we typically run through simulation using QMSprof, real silicon measurements may be much more noisy. One benefit of using Sniper and SDE to collect raw data is that these tools have relatively little to no sensitivity to the runtime environment, which tends to make their results much more repeatable.

A Appendix: Interface for QMSprof

This appendix demonstrates the interface for QMSprof. We first present an example of configuring the collector to run simulations, and then describe how to run the analyzer to extract summary statistics and generate plots.

A.1 Collector Interface

The collector interface for QMSprof is divided into four major parts, namely configuration for (1) workload binaries and run arguments, (2) Sniper models, (3) environment setup, and (4) experiment script.

For part (1), binaries and run arguments are configured by specifying a `Benchmarks` dictionary, which maps a key for each benchmark to a per-benchmark dictionary with additional information. When running an experiment, the collector uses information from a per-benchmark dictionary to *stage* each simulation run, i.e., creating a separate run folder for each simulation run in a staging area, and copying and/or renaming any necessary binaries and input files into that folder. This staging step is needed because workloads are not always built to support concurrent executions from the same run folder.

The per-benchmark dictionary is built to support benchmark *variants*, i.e., different versions of the same workload, with possibly different binaries or run arguments. This dictionary has several expected fields:

1. `bindir`: This string lists the subdirectory of the root benchmark directory containing the files for this benchmark. The root benchmark directory is a global variable specified separately in the top-level of the configuration file.
2. `files`: This dictionary maps a variant of the benchmark to a list of files needed to run each variant. Each file is itself a pair, with the first value being the name of the file in the source binary directory, and the second value being the name of the target file in the staging area. This pair allows an input file to be renamed in the staging area before it is run.
3. `runargs`: This dictionary maps a variant of a benchmark to the command-line arguments needed to run the variant.
4. `gen_inputs`: This dictionary maps a variant to a list of shell commands to execute in `bindir` to generate any input files that are needed for a run.
5. `requires_MPI`: This flag is set if this particular binary requires the use of an MPI library to execute. Our current prototype assumes one MPI rank per program, but in principle this assumption could be relaxed.

For the `files`, `runargs`, and `gen_inputs` dictionaries, if no exact match to a particular variant name is found in the dictionary, QMSprof will map to the key that matches the longest prefix.

As an example, Fig. 7 shows part of a configuration file specifying binaries and arguments for two benchmarks: LULESH and SNAP. In this configuration, running the `sim_vec` variant of the `LULESH`, uses the source file `lulesh2.0_vec` from the subdirectory `LULESH/binaries`. The staging process will copy and rename (or link) to a file named `LULESH_sim_vec` in a each simulation run directory. This staging allows QMSprof to use a consistent naming convention for its implementation, without requiring users to duplicate or change input file names in the source directory. To run the sim_vec variant of LULESH, QMSprof will use the sim argument of -s 27 -i 6 -p, since sim is the longest matching prefix of sim_vec.

SNAP has a slightly more complicated description, with a script command list in its gen_inputs parameter. This command list indicates that before staging any files in the files list, the collector should run the script genFile from the SNAP/binaries directory to generate extra input files. The strings <P> is a special pattern in arguments and commands that the collector replaces with the thread count for a particular run. Similarly, <RunDir> is a special pattern that gets represents the run directory used to store and run the binary.

Configuration of parts (2) and (3) are relatively straightforward, as illustrated in Fig. 8. The SniperConfigs dictionary describes the Sniper models that can be used in an experiment. The key 16C_2wide is a descriptive (usually short) string

```
Benchmarks = {
    "LULESH" : {
        "bindir"     :  "LULESH/binaries",
        "files" : {
            "test_vec"   : [ ("lulesh2.0_vec",   "LULESH_test_vec")  ],
            "test_novec" : [ ("lulesh2.0_novec", "LULESH_test_novec")],
            "sim_vec"    : [ ("lulesh2.0_vec",   "LULESH_sim_vec")   ],
            "sim_novec"  : [ ("lulesh2.0_novec", "LULESH_sim_novec") ],
        },
        "runargs" : {
            "test"       : "-s 4 -i 1",
            "sim"        : "-s 27 -i 6 -p",
            "full"       : "-s 36 -i 4 -p",
        }
    },
    "SNAP" : {
        "bindir"       : "SNAP/binaries",
        "gen_inputs" : {
            "test" :  [ "genFile 1 1 <P> 4 4 4 8 8  1 <RunDir> fin_test_P<P>" ],
            "sim"  :  [ "genFile 1 1 <P> 4 4 4 8 64 3 <RunDir> fin_sim_P<P>"  ],
        },
        "files" : {
            "test_vec"   : [ ("snap_test_vec",   "SNAP_test_vec")   ],
            "sim_vec"    : [ ("snap_sim_vec",    "SNAP_sim_vec")    ],
        },
        "runargs" : {
            "test" : "fin_test_P<P> fout_test_P<P>",
            "sim"  : "fin_sim_P<P> fout_sim_P<P>",
        },
        requires_MPI = True
}
```

Fig. 7. Example Benchmarks dictionary for configuring workloads in QMSprof.

```
# Sniper configurations
SniperConfigs = {
    "16C_2wide"      : "Manycore_16c_2wide.cfg",
    "16C_3wide"      : "Manycore_16c_3wide.cfg",
}

# Environment setup
EnvFiles = {
    "DefaultOpenMP"  : "ICCDefaultOpenMP.sh"
}

# Configure job manager
import collector.Netbatch
BatchJobModule = collector.NetBatch
BatchJobManager = BatchJobModule.JobManager()
```

Fig. 8. Example configuration for QMSprof for Sniper configurations and environment setup.

that the user provides for the Sniper configuration file `Manycore_16c_2wide.cfg`. Similarly, in the `EnvFiles` dictionary, `DefaultOpenMP` is a description of the environment file `ICCDefaultOpenMP.sh`. Each run script created by QMSprof sources a particular environment file before each run, passing in the thread count of the run as its argument. Thus, the user should use the environment file to setup any necessary runtime libraries or tools (e.g., compiler libraries, Sniper and SDE), and any other environment variables such as `OMP_NUM_THREADS`. Finally, Fig. 8 also specifies the job manager (e.g., Intel NetBatch) to use to run jobs in the desired compute environment.

Finally, for part (4), Fig. 9 shows an example experiment script that specifies the runs in the experiment. Each run (e.g., `Run0`) is specified as a tuple with 5 entries:

1. **Sniper configuration**: The Sniper configuration for the run, as defined by the keys in the input `SniperConfigs` dictionary. For example, `Run0` runs the `16C_2_wide` config.
2. **Thread Set**: The thread counts to run. For example, `Run0` uses the thread counts of 1, 2, 4, 8, 12, and 16.
3. **Experiment File**: The environment file for the run, as defined by the keys in the input `EnvFiles` dictionary. For example, `Run0` uses the `DefaultOpenMP` environment file.
4. **Program list**: The workload variants to run. The example in Fig. 9 executes both the `sim_vec` and `sim_novec` variants of LULESH and SNAP.
5. **Experiment Knobs**: An object that captures all the other configuration knobs for a particular run. This example uses default values for all the knobs, but additional customization is possible.

```
import collector.config    # Import collector module
import SampleConfig        # Import user's configuration file
# Get default simulation knobs
knobs = collector.config.Knobs(SampleConfig)

my_prog_list = ["LULESH_sim_vec", "LULESH_sim_novec",
               "SNAP_test_vec", "SNAP_sim_vec"]

experiment_map = {
   "Run0" : ("16C_2wide",
             [1, 2, 4, 8, 12, 16],
             "DefaultOpenMP",
             my_prog_list,
             knobs),
   "Run1" : ("16C_3wide",
             [1, 2, 4, 8, 12, 16],
             "DefaultOpenMP",
             my_prog_list,
             knobs),
}

# Name of directory to store simulation output.
output_directory = "SimOutput"
collector.experiment.GenScripts("ExperimentDescription",
                                experiment_map,
                                SampleConfig.BatchJobManager,
                                output_directory)
```

Fig. 9. Example experiment script for QMSprof.

A.2 Analyzer Interface

The analyzer for QMSprof is a separate script that takes a single input directory as its argument, scans the input directory for SDE and Sniper simulation output, parses the relevant raw statistics output files, and then generates summary plots. Our prototype for QMSprof has the specific plots demonstrated in Sect. 4 hard-coded as output, but in principle one could implement a more complex interface that would allow for some customization in the generated plots. The analyzer generates Gnuplot scripts and data files as output, which can be manually edited (e.g., to change titles, labels, or legends), and rerun manually to recreate plots.

Our prototype analyzer assumes that simulation output for each simulation run is placed in a separate folder, with the thread count appearing in the folder name. The analyzer uses the names of output folders to group different thread counts for a benchmark together in summary plots, and eliminates the common suffix across all runs to shorten legends in generated plots. These assumptions are designed for processing the output from the QMSprof collector, but one can also use the analyzer to generate summary plots from other simulation runs if the output directories follow a compatible naming convention.

References

1. Bentley, B.: Validating the Intel® Pentium® 4 microprocessor. In: Proceedings of the 38th Annual Design Automation Conference, DAC 2001, pp. 244–248. ACM, New York (2001). http://doi.acm.org/10.1145/378239.378473
2. Carlson, T.E., Heirman, W., Eeckhout, L.: Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: International Conference for High Performance Computing, Networking, Storage and Analysis (2011)
3. CORAL Collaboration: Oak Ridge, Argonne, Livermore. Benchmark codes. https://asc.llnl.gov/CORAL-benchmarks/
4. Himeno, R.: Himeno benchmark (2016). http://accc.riken.jp/en/supercom/himenobmt/
5. Hong, S.Y., Lim, J.O.J.: The WRF single-moment 6-class microphysics scheme (WSM 2006). J. Korean Meteorol. Soc. **42**(2), 129–151 (2006)
6. Intel® Advisor (2016). https://software.intel.com/en-us/intel-advisor-xe
7. Intel® Software Development Emulator (2016). https://software.intel.com/en-us/articles/intel-software-development-emulator
8. Intel® VTune^{TM} Amplifier (2016). https://software.intel.com/en-us/intel-vtune-amplifier-xe
9. Intel® Xeon Phi^{TM} Product Family (2016). http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html
10. Li, S.: Case study: computing black-scholes with Intel® advanced vector extensions (2012). https://software.intel.com/en-us/articles/case-study-computing-black-scholes-with-intel-advanced-vector-extensions
11. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Comput. Soc. Techn. Committee Comput. Archit. (TCCA) Newsl. 19–25 (1995)
12. McCalpin, J.D.: STREAM: sustainable memory bandwidth in high performance computers (2016). https://www.cs.virginia.edu/stream/
13. Shai, O., Shmueli, E., Feitelson, D.G.: Heuristics for resource matching in intel's compute farm. In: Desai, N., Cirne, W. (eds.) JSSPP 2013. LNCS, vol. 8429, pp. 116–135. Springer, Heidelberg (2014). doi:10.1007/978-3-662-43779-7_7
14. The Sniper Multi-Core Simulator (2016). http://snipersim.org
15. Sugumar, R.A., Abraham, S.G.: Efficient simulation of caches under opt replacement with applications to miss characterization. In: Proceedings of the ACM SIGMETRICS Conference (1993)
16. Tramm, J., Gunow, G.: SimpleMOC-kernel, version 2.0 (2015). https://github.com/ANL-CESAR/SimpleMOC-kernel
17. Valles, A., Zhang, W.: Optimizing for reacting Navier-Stokes equations. In: Reinders, J., Jeffers, J. (eds.) High Performance Parallelism Pearls, pp. 69–85. Morgan Kaufmann, Boston (2015). http://www.sciencedirect.com/science/article/pii/B9780128021187000042
18. Williams, T., Kelley, C.: gnuplot 4.6 (2014). http://www.gnuplot.info/docs_4.6/gnuplot.pdf
19. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: simple linux utility for resource management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003). doi:10.1007/10968987_3

20. Zhang, W.: miniSMC Benchmark (2014). https://github.com/WeiqunZhang/miniSMC
21. Zhang, Z., Phan, L.T.X., Tan, G., Jain, S., Duong, H., Loo, B.T., Lee, I.: On the feasibility of dynamic rescheduling on the Intel distributed computing platform. In: Proceedings of the 11th International Middleware Conference Industrial Track, Middleware Industrial Track 2010, pp. 4–10. ACM, New York (2010). http://doi.acm.org/10.1145/1891719.1891720