# Performance evaluation and visualization of scientific applications using PMlib

1st Kazunori Mikami
*Flagship 2020 Project*
*Riken R-CCS*
Kobe, Japan
kazunori.mikami@riken.jp

2nd Kenji Ono
*RIIT, Kyushu University*
*Riken R-CCS*
Fukuoka, Japan
keno@{cc.kyushu-u.ac, riken}.jp

3rd Jorji Nonaka
*HUD unit*
*Riken R-CCS*
Kobe, Japan
jorji@riken.jp

***Abstract*—The computational performance of scientific applications on HPC systems is often much lower than user expectation based on the system's maximum performance specifications. To understand the basis for this performance gap, a multi-perspective evaluation is important. For instance, from the user perspective, correlating the theoretical computation coded as a source program with the actual computation workload produced by the compilers is valuable. From the system perspective, evaluating the characteristics of microarchitecture elements such as processor core and memory is of significance. An open source library called PMlib was developed to address these types of synthetic evaluations. PMlib provides an avenue for reporting the arithmetic/application workload explicitly coded in the source program, as well as the actually executed system workload. It also provides detailed utilization reports of processor-specific hardware including the categorized SIMD instruction statistics, the layered cache hit/miss rate, and the effective memory bandwidth, which are captured via hardware performance counters (HWPC). Using PMlib, users can conduct a synthetic analysis of application performance, and obtain useful feedback for further optimized execution of applications.**

***Index Terms*—performance evaluation, arithmetic workload, system workload, HWPC, performance visualization, open source library, PMlib**

## I. Introduction

In the development and productive phase of scientific applications, performance evaluation is frequently conducted on target HPC systems, in order to understand the performance characteristics of the application on the target system, and to explore the possibility of applying further optimization to the application to reduce the amount of elapsed time. The sustained computational performance of applications is often much lower than what is expected based on the system specification. Preceding studies that investigated the origin of this gap between the expected performance and the actual performance addressed it primarily from the architecture perspective, e.g., utilization of parallel functional units, and locality of data residing on memory/cache hierarchy.

Generally, the performance of applications is measured in the unit of conducted workload/elapsed time. It should be noted that there are two factors that define the workload of a program in regard to performance measurement. One is the arithmetic workload defined in the applications source program, and the other is the system workload defined as the set of machine instructions executed on HPC systems. The difference in the performance based on these two workload definitions can be significant, as explained in section II, and is sometimes confusing to application users with respect to understanding their behavior.

PMlib [11], an open source library for performance evaluation has the functionality to explicitly measure the arithmetic workload counted in the source program using a manually formulated argument, as well as the functionality to automatically measure the system workload based on the statistics obtained from the hardware performance counters (HWPC), thus providing users the option to compare the difference in workloads. To access HWPC events, PMlib utilizes PAPI [10] low-level API, which supports number of processors typically used for HPC applications. PMlib has its own scheme for choosing the related HWPC event sets and sorting out event statistics according to the run-time environment variables, thus making it easier to extract the statistics of interest.

This paper first clarifies the difference between the arithmetic workload and the system workload. Next, the PMlib's useful features for performance evaluation are evaluated. Finally, some examples of the performance evaluation using PMlib are shown, that highlight the merits of multi-perspective analysis.

## II. Computational workload and Performance evaluation

### A. User perspective

The computational workload is generally perceived by scientific application developers as the total volume of arithmetic operations, expressed by the source code formulas written in Fortran/C/C++ etc. The workload at this level can be simply represented as below:

$$Workload = \sum_{i=1}^{types} W_{ops}(i) \tag{1}$$

where $W_{ops}(i)$ is the number of operations for the corresponding arithmetic type such as $i$=1:add, 2:sub, 3:mult, 4:div, 5:max, 6:min, 7:sqrt, etc., counted in the source program. The workload in the form of (1) is termed "arithmetic workload". In the development phase of scientific applications, developers

choose and deploy the numerical algorithm which requires the least amount of arithmetic workload based on scientific considerations, with the objective of reducing the elapsed time to obtain the desired simulation results.

The computational performance is defined based on the workload and the elapsed time as represented in the following expression.

$$Performance = Workload/Time \tag{2}$$

The performance based on the arithmetic workload (1) is most meaningful from user perspective, and is termed "arithmetic performance". In this paper, "user workload" refers to arithmetic workload, and "user performance" refers to arithmetic performance hereafter. For example, Linpack (HPCC) reports the user performance using the addition and multiplication terms.

$$Workload = W_{ops}(add) + W_{ops}(mult) \tag{3}$$
$$W_{ops}(add) = W_{ops}(mult) = 1/3N^3 + 3/4N^2 \tag{4}$$

where N is the size of the matrix to be solved. Linpack performance value flops is reported using this workload.

$$Gflops = N^2 \times (2/3 * N + 3/2) \times 1.0^{-9}/Time \tag{5}$$

Above workload and performance concept using the arithmetic operations can be extended to model the actually executed workload and performance. Compilers translate the arithmetic statements into the sequence of assembly instructions. The simplest approach to map the relationship between the source level arithmetic and the corresponding assembly instructions would be to apply weight factors to (1). Representing the combined effect of the latency and throughput of the instructions needed for $i$-th arithmetic type by weight factor $c(i)$, the workload is expressed as:

$$Workload = \sum_{i=1}^{types} (c(i) \times W_{ops}(i)) \tag{6}$$

The workload in the form of (6) is termed "application workload" in this paper. It approximates the actually executed system workload in a simple form and "application performance" can be defined in the same manner. The application workload defined as (6) can provide the base for analytical evaluations. However, its usefulness is limited to certain qualitative comparison for the following reasons. The mapping of mathematical functions to native instructions depends on not only the system architecture but also the compilers. There are multiple choices of available instructions to accomplish an arithmetic operation on modern processors, and their associated latency and throughput vary significantly. The compiler may choose scalar/vector context based on the its optimization strategy, as well as the degree of parallelism according to the loop length and the operation density. Therefore, the mapping needs to be examined case by case in quantitative evaluations. With this reason, the actually executed system workload is better evaluated from the system perspective as discussed next.

### B. System perspective

It this section, we will focus on workload and performance from the system perspective. The micro-architecture elements including the number of computation cores in a processor, the degree of parallelism inside each core, the depth of cache/memory hierarchy and the data move rate at each hierarchy, all have relational impact on how machine instructions are executed, thus achieving the sustained computing performance. The fact that the actual numerical computation is performed by a set of hardware components makes it rational to measure the system workload using the information obtained from hardware.

To date, measuring the system workload using HWPC has been recognized as valid practice. HWPC records the processor specific hardware utilization statistics including categorized SIMD instructions, layered cache hit/miss counts, and memory load/store instuctions. HWPC statistics can be accessed through appropriate APIs. For example, floating point operation related events on Intel Xeon Skylake processor which is used in section IV-D can be accessed using PAPI API as the following events.

```
fpsp1  : "FP_ARITH:SCALAR_SINGLE";
fpsp4  : "FP_ARITH:128B_PACKED_SINGLE";
fpsp8  : "FP_ARITH:256B_PACKED_SINGLE";
fpsp16 : "FP_ARITH:512B_PACKED_SINGLE";
fpdp1  : "FP_ARITH:SCALAR_DOUBLE";
fpdp2  : "FP_ARITH:128B_PACKED_DOUBLE";
fpdp4  : "FP_ARITH:256B_PACKED_DOUBLE";
fpdp8  : "FP_ARITH:512B_PACKED_DOUBLE";
```

The floating point operation workload is obtained from these event counts multiplied by the corresponding width of the instructions. The total number of single precision (32-bit) and double precision (64-bit) floating point operations is:

$$\begin{aligned} \mathrm{W}_{hwpc} = {} & \mathrm{fpsp1} + \mathrm{fpdp1} \\ & + 4.0 * \mathrm{fpsp4} + 8.0 * \mathrm{fpsp8} + 16.0 * \mathrm{fpsp16} \\ & + 2.0 * \mathrm{fpdp2} + 4.0 * \mathrm{fpdp4} + 8.0 * \mathrm{fpdp8} \end{aligned} \tag{7}$$

The workload and the corresponding performance using this approach are termed "HWPC workload" and "HWPC performance", and they are refered to as system workload and system performance hereafter.

### C. Difference in workloads

In this section, an example is presented that illustrates the significant difference between the user perspective arithmetic workload and the system perspective HWPC workload.

Fig. 1 shows the user performance and Fig. 2 shows the system performance of the same kernels selected from section IV-A on IVY server listed in II. PMlib user mode report and HWPC mode report are used to obtain these results. The vertical axis shows the performance per core defined by (2), corresponding to their workloads. Fig. 3 shows the difference between these two parameters as a ratio. The execution elapsed time of Fig. 1 and Fig. 2 are basically identical, and the difference observed in Fig. 3 actually represents the difference in the workload definition. The ratio ranges from 1.0 to 2.6

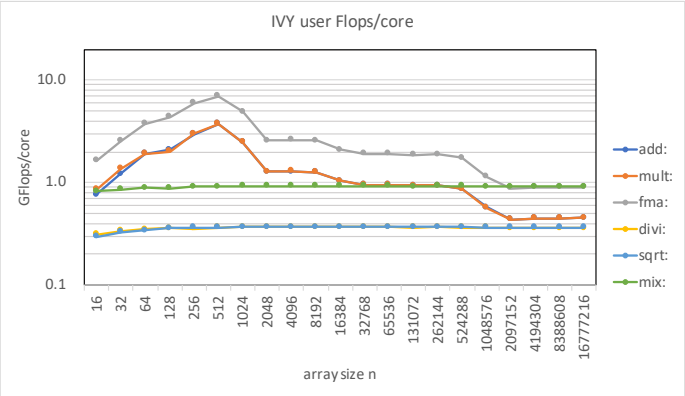


Fig. 1. User perspective workload and performance

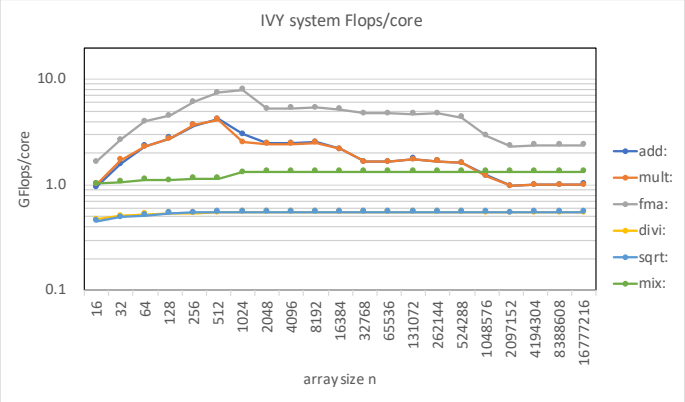


Fig. 2. System perspective workload and performance

depending on the type of arithmetic operation, and it also varies depending on the loop length.

This simple example clearly indicates that the performance evaluation of scientific applications on HPC systems must be associated with the clear definition of workload, and suggests that the evaluation from both the user and the system perspectives should be conducted. A detailed analysis of a similar example is given in section IV-A.

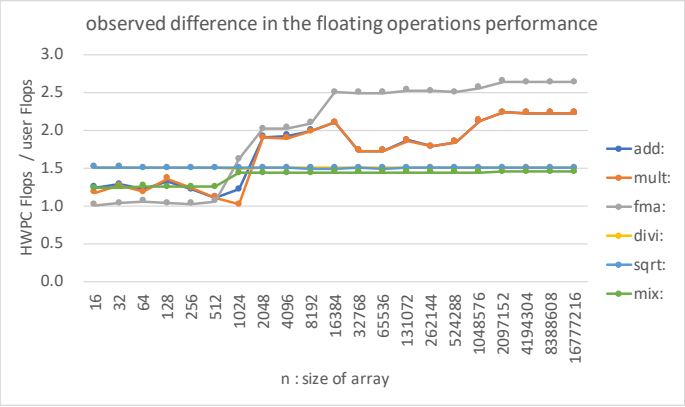


Fig. 3. Difference between the workloads

TABLE I
LIST OF BASIC APIS PROVIDED BY PMLIB

| C++ | function | arguments |
|---|---|---|
| initialize | initial setup | number of sections |
| setProperties | sections property | label, type, exclusiveness |
| start | start of section | label |
| stop | end of section | label, arithmetic workload |
| print | basic report | filename, comment, sorting |
| printDetail | per process report | filename, legend, sorting |
| printThreads | per thread report | filename, rankID, sorting |

## III. PERFORMANCE MONITORING LIBRARY

### A. Overview of PMlib

PMlib is an open source library for monitoring the performance of scientific applications and is able to measure the arithmetic, application and HWPC workload discussed in section II-A and II-B. PMlib defines a small number of properties such as name, type of operation, exclusiveness, and workload values to the specified block within a source program called section. Actual location of the section is decided by PMlib start and stop APIs, and there can be multiple locations to form a single section. The report from PMlib is sorted by sections, processes and threads, depending on the controlling environment variable.

### B. PMlib API

PMlib provides APIs for C++ and Fortran programs. There are only a small number of APIs which must be called from applications. C++ APIs are shown in Table I.

Fortran APIs follow the similar naming convention, i.e. f_pm_{*C++API_name*}, The following is an example Fortran program calling PMlib APIs.

```
1  program main
2      call  f_pm_initialize   (Nsections)
3      call  f_pm_setproperties  ("section1 ", icalc , iexcl )
4      call  f_pm_start  ("section1 ")
5      call  some_computation (fops)
6      call  f_pm_stop ("section1 ", fops,0)
7      call  f_pm_print  ("",0)
8      call  f_pm_printdetail   ("",0,0)
9  end
```

### C. Choosing workload type

The choice of the workload is made by a run-time environment variable HWPC_CHOOSER which can take one of the following values.

FLOPS|BANDWIDTH|VECTOR|CACHE|CYCLE|user

If the arithmetic workload is measured, The number of arithmetic operations or the formula defining the workload must be provided to PMlib API as user argument. PMlib accumulates the workload per section sandwiched by start and stop APIs. In practice, counting the number of arithmetic operations in the application source code can be a non-trivial task, and automating such a task is desirable. To automate the counting task for Fortran applications, we can utilize the

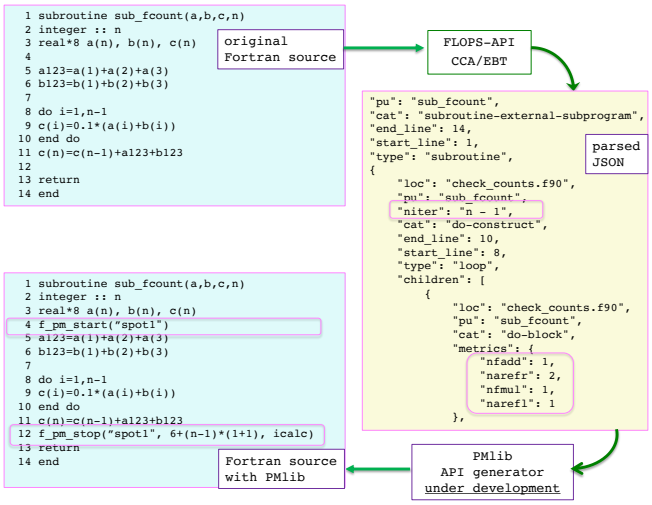


Fig. 4. Counting the operation with ccaebt parser program

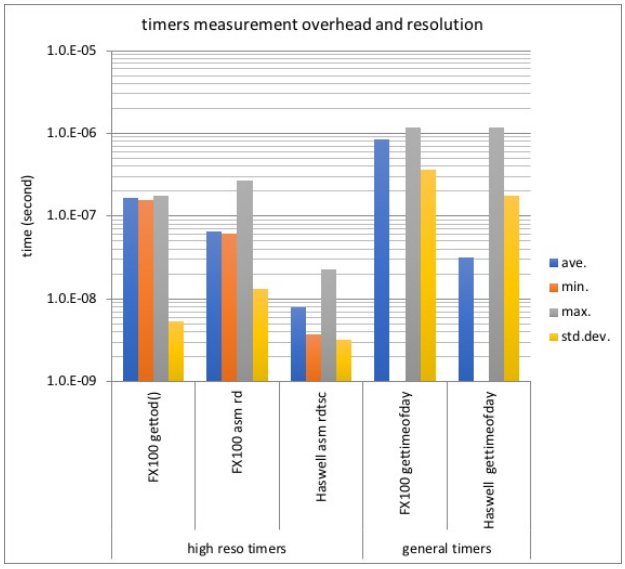


Fig. 5. general timer and precise timer

related research [7], [8] to let the parser program analyze the Fortran source code and produce JSON intermediate reports showing the arithmetic operations counts. Then, a second filter can be used to produce the partial Fortran source program including call statements for PMlib API with provisioned arguments. Fig. 4 shows the schematic view to accomplish this flow.

If the HWPC workload is measured, PMlib internally utilizes PAPI low-level APIs to access HWPC event statistics. Each type of processor defines many HWPC events. For instance, the standard output from *papi_native_avail* command on a Skylake processor produces 5000+ lines. PMlib detects the type of processor, chooses the appropriate native PAPI event sets and translates the raw event statistics into the desired performance category according to HWPC_CHOOSER. The application developpers do not have to worry about the different PAPI event mnemonics on different processors. In HWPC mode, user given operation count arguments are discarded.

### D. High-resolution timer available to PMlib

PMlib utilizes the Linux standard timer gettimeofday() by default. It also has the installation option to utilize a system specific high-resolution low overhead timer if such timer is available on the platform. Fig. 5 shows a comparison of a standard timer and the high-resolution timer on the two HPC systems highlighted in the previous section. PMlib has a provision to use them as an installation option.

### E. Output information

The default output information from PMlib is a blocked text report based on the time-averaged performance statistics. PMlib is also able to produce the Open Trace Format (OTF) [9], [5] tracing file that enables the detail time-history visualization of the shorter interval statistics.

### F. Other performance evaluation tools and related research

Numerous tools have been developed to evaluate the performance of HPC systems. In general, open source tools are designed to be portable. Their functionalities are distinct and in many cases, multiple tools are used in sequence to obtain the desired performance information. These tools are classified as either sampling or tracing tools and they each have different overheads. Commercial vendor tools are used for many HPC systems. They are best used for specific processor types. HPC systems vendors also supply performance evaluation tools for their systems. Those vendor tools are integrated into systems and are easy to use in general, while the availability is limited to specific systems.

Some of the commonly used tools are listed below. In the open source category:

- Scalasca [2], [6] : trace generation, Score-P infrastructure
- Extrae [1] : trace generation
- PAPI [10] : API to access HWPC
- Linux perf tools : API to access HWPC

In the commercial vendor category:

- Intel VTune [3] : Intel processors
- PGI Profiler [4] : X86 and GPU

These existing tools all utilize the processor performance information based on HWPC workload, i.e. system workload. PMlib appears to be the only tool that enables arithmetic and/or application workload evaluation.

## IV. Performance Evaluation using PMlib

### A. Basic kernels and test servers

Some examples of performance evaluation utilizing PMlib are shown in this section. The tested programs are the basic kernels composed of four basic arithmetic operations, square root and reversed distance. The kernels repeat the same operations with loop length hidden to the compiler. The kernels

TABLE II
SERVER CONFIGURATION AND HARDWARE SPECIFICATION

| Symbol | FX100 | SKY | IVY |
|---|---|---|---|
| Platform | FX100 | Skylake | Ivybridge |
| CPU | SPARC64 XIfx | Gold 6148 | E5-4620v2 |
| core GHz | 1.975 | 2.4 | 2.6 |
| core Gflops | 31.6 | 30 | |
| #cores/cpu* | 16 | 20 | 8 |
| L1$ size(D,I) | 64KB, 64KB | 32KB, 32KB | |
| L1D$ BW GB/s | 140/R+70/W | 154/R + 77/W | |
| $ Linesize | 256B | 64B | 64B |
| L2$ size | - | 1MB | 256KB |
| L2$ BW GB/s/c | - | 154 ( 70) | |
| LL$ size | 12MB | 28MB | 20 MB |
| LL$ BW GB/s/c | 70/R+35/W | 77 ( 43) | |
| Memory | HMC(8x16Ls) | DDR4-2666 | DDR3-1600 |
| Mem GB/s/cpu* | 120/R+120/W | 128 | |

remark. cpu* indicates processor or CMG

are expected to be executed using pipelined SIMD instructions on the measuring HPC systems. The Fortran source of these kernels is shown below.

```
1  subroutine  add(a, b, c, n)
2      do i=1,n
3      c(i)=a(i)+b(i)
4      end do
5  subroutine  mult(a, b, c, n)
6      do i=1,n
7      c(i)=a(i)*b(i)
8      end do
9  subroutine  fma(a, b, c, d, n)
10     do i=1,n
11     c(i)=a(i)+b(i)*d
12     end do
13 subroutine  divide (a, b, c, n)
14     do i=1,n
15     c(i)=b(i)/a(i)
16     end do
17 subroutine  sqrt (a, b, c, n)
18     do i=1,n
19     c(i)=sqrt(a(i))
20     end do
21 subroutine  mix(a, b, c, n)
22     do i=1,n
23     c(i)=1.0/sqrt (a(i)**2+b(i)**2)
24     end do
```

The choice of data precision, either double or single, is made at compile time, and the default optimization is done by the compilers. The copies of the same kernel are run on all cores of the processor. The kernels are called many times. After the calling overhead is subtracted, the average elapsed time including the do loop overhead cost is obtained.

The servers used for the tests are:

- SGI Intel Ivybridge server
- SGI Intel Skylake server
- Fujitsu prime HPC FX100

Their processor specifications and the major performance specifications are listed in Table II.

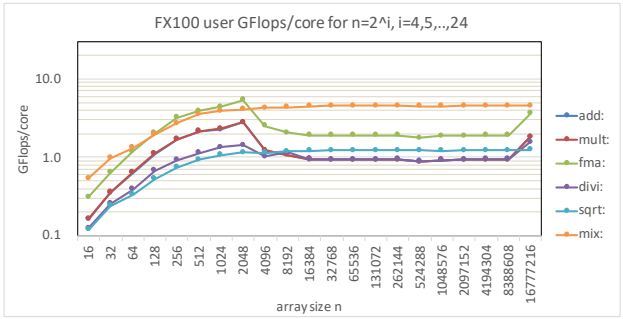


Fig. 6. user performance of FX100 for basic kernels

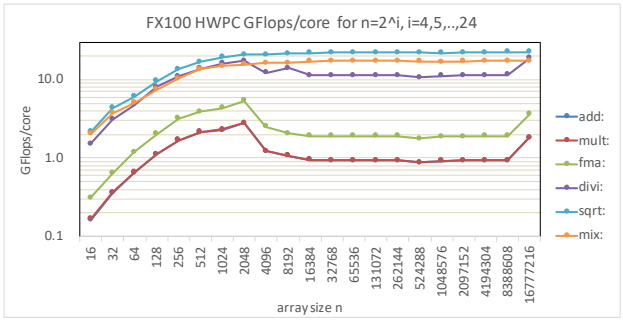


Fig. 7. system performance of FX100 for basic kernels

### B. Kernel performance on FX100 in long loops

Fig. 6 shows the user performance,i.e. arithmetic performance, of the basic kernels on FX100 system for the loop length of $n = 1, 2, 4, .., 2^{24}$ .

Fig. 7 shows the system performance based on the HWPC workload. The HWPC workload of the floating point operations on FX100 is recorded as events sorted by SIMD width as below.

```
fpdp1  : "1FLOPS_INSTRUCTIONS";
fpdp2  : "2FLOPS_INSTRUCTIONS";
fpdp4  : "4FLOPS_INSTRUCTIONS";
fpdp8  : "8FLOPS_INSTRUCTIONS";
fpdp16 : "16FLOPS_INSTRUCTIONS";
```

The total number of HWPC floating point operations is calculated as the sum of single precision and double precision operations.

$$\begin{aligned} \mathrm{W}_{hwpc} = \mathrm{fpdp1} + 2.0 * \mathrm{fpdp2} + 4.0 * \mathrm{fpdp4} \\ + 8.0 * \mathrm{fpdp8} + 16.0 * \mathrm{fpdp1} \end{aligned} \quad (8)$$

In both figures, the performance of add/mult/fma tends to improve with an increase of the loop length until the L1 cache size limit is reached, and subsequently drops to the bandwidth constrained performance level of the next level cache. This performance tendency for add/mult/fma is consistent with the reports from other related research investigations. It can be associated with the roofline [12] envelop which is shown in Fig. 8

The performance of div/sqrt/mix shows quite different tendency, and their system performance stays much higher than the user performance.

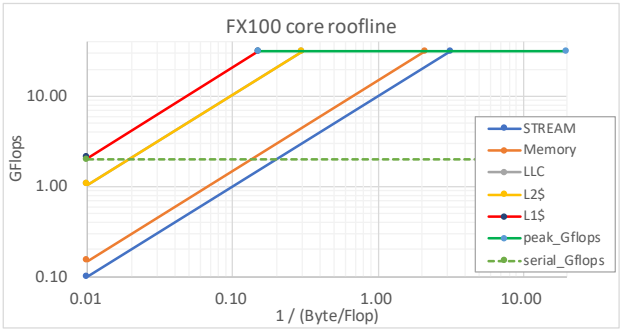


Fig. 8. roofline envelop of FX100

TABLE III
PMLIB HWPC EVENTS FOR HWPC_CHOOSER

| kernel | 1fp.ins | 2fp.ins | 4fp.ins | 8fp.ins | 16fp.ins | vector% |
|---|---|---|---|---|---|---|
| add | 1.90e+2 | 2.60e+2 | 5.12e+5 | 0.00e+0 | 0.0e+0 | 1.0e+2 |
| fma | 2.00e+2 | 2.60e+2 | 0.00e+0 | 5.12e+5 | 0.0e+0 | 1.0e+2 |
| divide | 1.90e+2 | 2.60e+2 | 1.02e+6 | 2.56e+6 | 0.0e+0 | 1.0e+2 |

Partial results extracted from PMlib HWPC report are shown in Table III. The environment variable HWPC_CHOOSER explained in III-C is set to VECTOR in this case, and the native HWPC events related with vectorization are sorted and reported.

According to the report, the number of required operations for division is almost an order of magnitude more than that for addition, even though their arithmetic workload is the same. Compiler optimization transforms a division arithmetic statement into a series of reciprocal approximation and multiply-add instructions, rather than issuing a simple division instruction. They take advantage of pipelined sequence of SIMD instructions and maintain high flop/byte ratio. So the apparent system performance of this operation becomes very high, but the actual user performance is much lower, almost the same as add/mult in this case. In fact, divide kernel runs much slower than add/mult, if not pipelined at all.

The comparison of system/user performance ratio is shown in Table IV. The numbers in the table can be directly obtained from HWPC_CHOOSER=FLOPS setting. Without having to read the assembly code generated by the compiler, users can capture some idea from the statistics of PMlib HWPC report.

The reason for the system performance curves of div/sqrt/mix not having the drop near the L1 cache size limit is explained by the high computational density which hides the data supply bandwidth.

### C. Kernel performance on FX100 in short loops

We now take a closer look at short loop region where the overhead to process the loop set-up and update cannot be neglected, and the data access latency affects more compared to long loop region.

TABLE IV
RATIO OF SYSTEM/USER PERFORMANCE OF FX100 FOR BASIC KERNELS

| type | add | mult | fma | div | sqrt | mix |
|---|---|---|---|---|---|---|
| ratio | 1.0 | 1.0 | 1.0 | 12.0 | 18.0 | 38.0 |

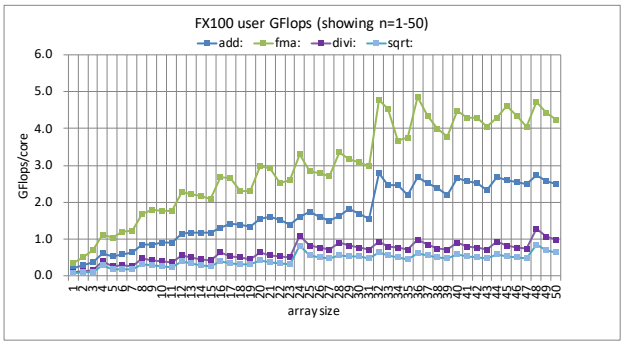


Fig. 9. FX100 user performance close up

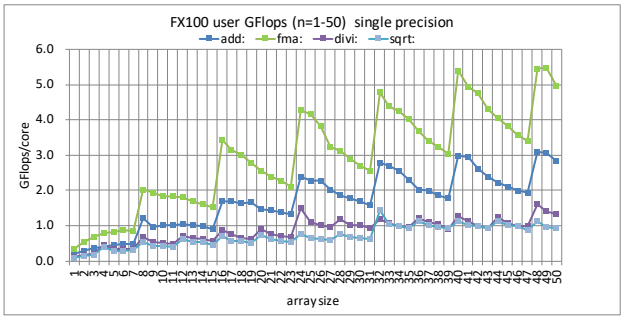


Fig. 10. FX100 user performance close up for single precision kernels

Fig. 9 shows the close-up user performance of the same basic kernels for $n = 1, 2, 3, .., 50$ . The performance curves do not show the gradual increase. Instead, they show steep stepping shapes at multiples of a constant interval. Similar stepping behavior of the performance is observed on other HPC systems with SIMD supporting processors as well, and its pattern interval corresponds to the data width of the SIMD instructions.

FX100 can utilize 256-bit wide data access and arithmetic SIMD instructions. In double precision arithmetic the resulting widest interval is $256/64 = 4$ . So the performance is expected to be efficient at $n = 4, 8, 12, \cdots$. Fig. 9 verifies such expectation. It highlights the fact that program coding practices to maintain the loop length as a multiple of the SIMD width has a significant impact on the user performance in short loop computation.

For single precision, the interval becomes $256/32 = 8$ , and the performance impact is even stronger as seen in Fig. 10 for FX100.

### D. Kernel performance on SKY in short loops

The same performance measurement is conducted on SKY. We focus on the short loop region, again. Fig. 11 shows the double precision user performance of the same basic kernels on SKY. The overall line shapes of add and fma appear similar to those of FX100, i.e. stepping shapes at $n = 4, 8, 12, \cdots$. Since SKY supports 512-bit wide SIMD instructions called

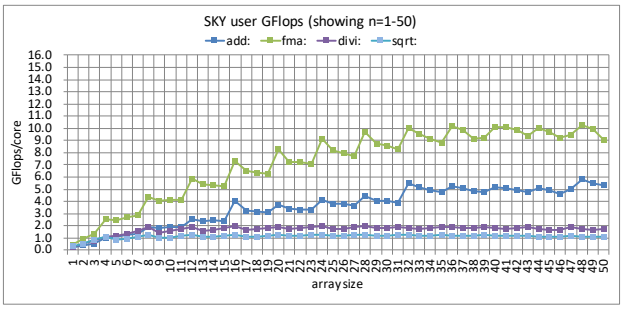


Fig. 11. SKY user performance close up

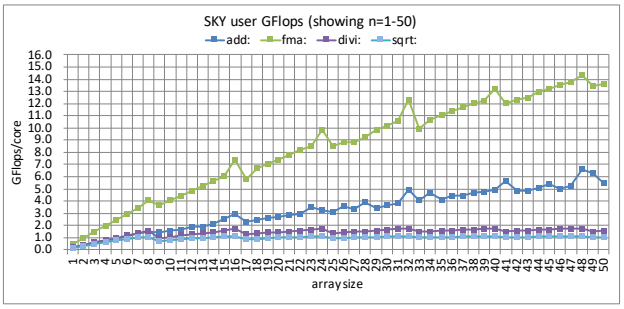


Fig. 12. SKY user performance close up forcing 512-bit SIMD

avx512, the stepping interval is expected to be $512/64 = 8$ . Checking the PMlib HWPC reports tells that the kernels are executed mostly using 256-bit wide SIMD instructions, not 512-bit. Further check into the assembly code matched this report. Knowing this, a compiler option is added to force producing 512-bit instructions. The revised test results are shown in Fig. 12. Steppings are now observed at $n = 8, 16, \cdots$, but in different shapes. This revised performance obviously reflects the effect of compiler optimization.

The overall line shapes of divide and sqrt appear low and flat compared to those of FX100, This indicates that the executed instructions were not pipelined, and their latency dominates the performance.

### E. Future development

Using PMlib, users are able to obtain various performance information to understand the behavior of applications. In order to reflect the obtained information to improve the performance of user applications, there can be many approaches going forward. One approache would be enhance the report from PMlib. If PMlib is able to report the saturated units in a processor hardware related to the user defined section in the source program, it maybe a useful index to optimize not only the application source but also the design of the future hardware. Further research is needed to harness such capability.

## V. Conclusion

In this paper, we discussed the importance of multi-perspective evaluation to understand the performance characteristics. The difference between the arithmetic workload coded as the source program and the system workload executed as machine instructions explains the gap between the user expected performance and the actually achieved performance on HPC systems. We presented an open source library called PMlib, which analyzes the behavior of these workloads. PMlib provides an avenue for reporting the arithmetic/application workload as well as the actually executed system workload. It also provides detailed utilization reports of processor-specific hardware including the SIMD instruction statistics, the layered cache hit/miss rate, and the effective memory bandwidth, which are captured via hardware performance counters through PAPI library. We could clarify the distinctive behavior by applying the PMlib to a target program on different hardware architectures, and also verified that PMlib allows users to conduct synthetic analyses of application performance, thus enabling the users to obtain useful feedback for further optimizations and better performance.

## Acknowledgment

This research was carried out using computational resources of FX100 and the K computer at the RIKEN Center for Computational Science in Kobe, Japan, and the computer resources offered under the category of a general project by the Research Institute for Information Technology, Kyushu University.

## References

[1] Barcelona Supercomputing Center. Bsc performance tools. available from https://tools.bsc.es/.

[2] Jülich Supercomputing Centre. Scalasca. available from http://www.scalasca.org/.

[3] Intel Corporation. Intel vtune ® amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[4] NVIDIA Corporation. Pgi profiler. https://www.pgroup.com/products/.

[5] Technische Universität Dresden. Open Trace Format. https://tu-dresden.de/zih/forschung/projekte/otf/.

[6] Markus; et al. Geimer. The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, Vol. 22, No. 6, pp. 702–719, Apr 2010.

[7] Masatomo Hashimoto, Masaaki Terai, Toshiyuki Maeda, and Kazuo Minami. Extracting facts from performance tuning history of scientific applications for predicting effective optimization patterns. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR 2015)*, pp. 12–23, 2015.

[8] Masatomo Hashimoto, Masaaki Terai, Toshiyuki Maeda, and Kazuo Minami. CCA/EBT : Code comprehension assistance tool for evidence-based performance tuning, January 2018. Poster presented at International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia 2018).

[9] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In *Proceedings of the 6th International Conference on Computational Science - Volume Part II*, ICCS'06, pp. 526–533, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] University of Tennessee. Papi, performance api. available from http://icl.cs.utk.edu/papi/software/index.html.

[11] Kenji Ono. PMlib public download site. http://avr-aics-riken.github.io/PMlib/.

[12] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, Vol. 52, No. 4, pp. 65–76, April 2009.