MeterPU: a generic measurement abstraction API: Enabling energy-tuned skeleton backend selection

Lu Li and Christoph Kessler

Noname manuscript No.
(will be inserted by the editor)

MeterPU: A Generic Measurement Abstraction API

Enabling Energy-tuned Skeleton Backend Selection

Lu Li · Christoph Kessler

Received: date / Accepted: date

Abstract We present MeterPU, an easy-to-use, generic and low-overhead abstraction API for taking measurements of various metrics (time, energy) on different hardware components (e.g. CPU, DRAM, GPU) in a heterogeneous computer system, using pluggable platform-specific measurement implementations behind a common interface in C++. We show that with MeterPU, not only legacy (time) optimization frameworks, such as autotuned skeleton back-end selection, can be easily retargeted for energy optimization, but also switching between measurement metrics or techniques for arbitrary code sections now becomes trivial. We apply MeterPU to implement the first energy-tunable skeleton programming framework, based on the SkePU skeleton programming library.

Keywords MeterPU · Measurement Abstraction API · GPU · Performance Measurement · Energy Measurement · Auto-tuning · Skeleton Programming

1 Introduction

Energy optimization can be achieved by suitably adapting an application's execution flow to the underlying hardware and the current execution context. A powerful adaptation mechanism is implementation switching depending on call context, such as problem size etc., especially in heterogeneous architectures where the energy efficiency can differ significantly among different types

Lu Li
IDA, Linköping University
Linköping, Sweden
E-mail: lu.li@liu.se

Christoph Kessler
IDA, Linköping University
Linköping, Sweden
E-mail: christoph.kessler@liu.se

of processors. Thus smart dynamic selection of implementation variants, for instance, switching to a GPU implementation for sufficiently large problem sizes, can yield remarkable reduction of energy cost.

In skeleton programming, implementation switching is convenient for the user as different platform-specific implementations (back-ends) of the skeletons are available and thus no further annotations nor platform-specific coding is required for the user. Especially in SkePU [1,2], different implementations on different platforms (CPU, OpenMP, CUDA, OpenCL) of each skeleton are automatically generated by the SkePU framework from user-defined code plug-ins in a platform-independent form. The optimization via dynamic implementation switching has been studied in previous work [3,4] with program runtime as the primary optimization goal.

However, energy has become the main bottleneck and a main concern in recent years, thus the investigation on the applicability of energy-tuned SkePU (featuring dynamic implementation selection for lower energy cost) and the reduction of energy cost by such techniques has become an urgent task.

As mentioned earlier, the optimization for shorter program runtime has been studied in SkePU previously. Furthermore, one can generalize the optimization framework by decoupling it from a specific optimization goal (metric). In other words, the autotuning (optimization) framework *optimizes and only optimizes on objective function values*, which can be time values or energy values or other possible metric values. We could thus port the autotuning framework by providing energy values instead of time values, and an autotuning framework originally used to optimize for time could now actually optimize for energy automatically. Not only for SkePU, but also for other optimization and empirical modeling frameworks based on sampling, the frameworks could be reused with trivial changes as long as the frameworks' underlying assumptions on the metrics' behavior over the parameter space (such as monotonicity or the heuristic convexity assumption [4,5] used for relative performance modeling) remain basically the same with the new metric.

The major software-technical barrier to achieve such a goal is to make the interface of energy measurement the same as that of time measurement, and if possible, be the same as that for other metrics that might appear in the future. Such an *abstraction of measurement* is not only useful for reuse of a legacy optimization framework, but also allows the platform-specific postprocessing of measurements of e.g. energy (which is required, e.g., with recent GPUs), and the easy switching among different optimization goals. For this purpose, we design a software prototype *MeterPU*[1], a C++ template-based library to provide a unified programming API for measurement of various metrics on different types of processors. With such a library at hand, we only need to modify the code of the legacy tuning frameworks, e.g. in SkePU, at a few places, namely where the measurements are taken for empirical model constructions. For the evaluation of the reduction of energy cost, we can measure for SkePU skeleton calls at different call contexts, e.g. problem sizes, whether

[1] MeterPU is publically available at `http://www.ida.liu.se/labs/pelab/meterpu/`.

the selection is smart enough, and how much the reduction of energy cost can be achieved. Of course, MeterPU can also be used stand-alone in arbitrary C++ applications for generic portable measuring.

To sum up our contributions, MeterPU provides a very easy-to-use measurement abstraction API with negligible overhead, while still keeping the generality. SkePU, when integrated with MeterPU, provides the first energy-tuned reusable skeletons on heterogeneous systems, to the best of our knowledge.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the idea and design of MeterPU. How it helps to make SkePU energy-tuned is shown in Section 4. Section 5 lists our experimental results, and Section 6 concludes.

2 Related Work

2.1 Measurement Abstraction

Several software libraries are proposed to act as an abstraction of (energy) measurement on different hardware components. EML [6] is a C library that implements a software abstraction of measurement with low overhead (1.54%), supporting dynamic discovery of measurement devices. REPARA's performance and energy monitoring library [7] provides a unified interface to support both counter-based and hardware-based measurement methods, which can be discovered by their hardware description language HPP-DL. Comparing to those libraries, MeterPU provides the simplest interface while maintaining generality, which facilitates retargeting legacy tuning frameworks, and keeps the overhead minimal (only one extra function call).

Monitoring frameworks, e.g. Nagios [8], and GroundWork [9], take measurements in either intrusive or non-intrusive way while applications are running, and store performance data to file systems or databases. However, from the optimization's point of view, retrieving performance data from file systems or databases to calculate an aggregate metric value at runtime in a feedback loop adds additional overhead compared with a light-weight measurement abstraction such as MeterPU. Another monitoring framework PowerAPI [10], is implemented by scala language with the goal of the capabilities for monitoring programs in different languages etc. MeterPU is implemented in C++, with potentially better integration in main-frame heterogeneous programming environments such as OpenMP, CUDA, etc.

2.2 Skeleton Programming

Besides SkePU, there are other skeleton programming frameworks targeting heterogeneous systems. SkelCL [11] provides OpenCL-based high-level skeletons and data types to ease the development of programming on heterogeneous systems and competitive performance results. MueSLi [12] is a C++ template

library with various data and task parallel skeletons where the users may specify where the skeletons are executed (CPUs or GPUs). Fastflow [13] provides layered abstraction on cache-coherent shared memory multicores, with typical streaming patterns. Marrow [14] is a skeleton framework for OpenCL computations, enabling combinations, nesting of skeletons and overlapping of communication and computation. Comparing to these frameworks, SkePU provides automated implementation (backend) selection for both time and energy optimization.

3 MeterPU

MeterPU is a C++ template-based library prototype, which aims to provide an abstraction layer for measurement of various metrics on different platforms. Currently it supports measuring time and energy on CPU (energy on DRAM) and (single and multiple) GPU(s). It can also measure energy by external measurement devices (e.g. Wattsup power meter). The library can be extended easily for new metrics, such as FLOPS, cache misses etc.

3.1 Library API and Example Applications

MeterPU exposes to programmers a unified API no matter which type of supported metric is used.

Listing 1 shows the API that MeterPU exposes to programmers. It consists of a class called `Meter` with a template parameter to specify the Meter's type. The `Type` parameter specifies which metric is used, such as time or energy, and which hardware component to measure, such as CPU or GPU. The generic type also has a C++ trait [15] defined with it, where it can fetch all metric-related types. A C++ trait class is a class that encapsulates all related types together. For instance, if CPU time is used as metric (template parameter `Type` is `CPU_Time`), then `typename Meter_Traits<Type>::ResultType` will give the time unit, such as microsecond. The four main member functions of `Meter` are `start()`, `stop()`, `calc()`, and `get_value()`, which are used as follows:

- start(): mark the start of a measurement phase/period.
- stop(): mark the end of a measurement phase/period.
- calc(): calculate the metric value based on the measurements taken between start() and stop(). It is possible to merge calc() with stop(), but here we expose the two functions to programmers[2], and class `Meter` can be inherited if the merge is preferred.
- get_value(): return the value of the measurement between the start and stop calls.

[2] Actually this separation can lead to reduced overhead if more than one meter is measured over the same time interval, as the possibly time-consuming calc() calls can be done outside the timing interval for each metric.

```
template<class Type>
class Meter
{
public:
 void start();
 void stop();
 void calc();
 typename Meter_Traits<Type>::ResultType const &get_value() const;
private:
  ...
}
```

Listing 1 Main MeterPU API

Listing 2 shows an example application that uses MeterPU. First a meter is initialized with the type `CPU_Time`, showing that it is a time meter on CPU. Then `start()` and `stop()` are used to mark the start and end of a time measurement by the meter, and finally `calc()` and `get_value()` are called to calculate the difference in metric value between the two program points, and to print the meter reading on a terminal.

```
#include <MeterPU.h>
int main()
{
    using namespace MeterPU;
    Meter<CPU_Time> meter;
    meter.start();
    //Do sth here
    sleep(2);
    meter.stop();
    meter.calc();
    print( meter.get_value() );
}
```

Listing 2 An example application that uses MeterPU library

3.2 Implementation

The actual measurement plug-in code used with MeterPU is implemented on top of native measurement libraries. For the CPU time measurement, it may use the native `clock_gettime()` function. For the CPU energy measurement, it may use the Intel PCM library. For the GPU energy measurement, it may use the Nvidia NVML library (nvmlDeviceGetPowerUsage(), reported sampling error within 5% [16]) to get power samples, and apply numerical integration to calculate energy values and compensate the Nvidia NVML's capacitor-like effect[3] with K20c. MeterPU is not coupled with a specific measurement method, and it can be hooked with more accurate methods or devices. So far we are not

[3] The approach by Burtscher et al. [17] to correct power values from Nvidia NVML library might give better accuracy on GPU power measurement, see Sect. 3.4. On the other hand, MeterPU is not coupled with a specific compensation method.

aware of any library that offers power samples on the PCI Express bus on the motherboard side, thus the motherboard-side energy of GPU data transfers is not supported yet in MeterPU, and not included in the experimental results described in Section 5. In the future one can consider to use an external power meter between the correct power rails of a motherboard for the power samples of PCIe, in this case a mother computer can act as a host for the power meters, and one can develop a library for polling samples of PCIe power from the host, then it will integrate with MeterPU easily.

MeterPU uses C++ traits to encapsulate all type information related to a meter type, e.g. for CPU time, `Meter_Traits<CPU_Time>::ResultType` gives the unit type for the final result of a CPU time measurement. The class `Meter` acts as a facade, all types related to its meter type will change as the template expands statically.

MeterPU uses the `Meter` class as the interface exposed to programmers. A class has the advantage that it can hide arbitrarily complex data structures and logic underneath, and keeps the exposed part simple and unified, thus MeterPU has the potential and possibility to unify the API design for all measurements of metrics of interest, and acts as an important step towards this goal. MeterPU is designed not only to make a legacy optimization framework based on empirical sampling and modeling easy to migrate to other optimization goals, but also targets at proposing a general interface standard, thus different vendors could hook their own abstracted measurement implementation if necessary. MeterPU is also designed to be extensible, thus it can adapt to metrics that will appear in the future easily.

3.3 More Examples

Listing 3 (see code snippets for `cpu_energy_meter` and `gpu_energy_meter`) show how MeterPU can be used to measure CPU and Nvidia GPU energy with a unified API; only the template parameter to initialize a meter differs in these scenarios (other code to use these meters are the same as in Listing 2). For GPU energy measurement, there is one extra template needed, the device id of a GPU. For example, to initialize a meter associated with a GPU of device id 3, one can write: `Meter< NVML_Energy<3> > meter;` and `Meter< NVML_Energy<> > meter;` for the GPU with device id 0 as default, which is usually the case on machines with only one Nvidia GPU.

```
Meter<PCM_Energy> cpu_energy_meter;
Meter< NVML_Energy<> > gpu_energy_meter;
Meter<CUDA_Time> nvidia_gpu_time_meter;
Meter<CUDA_Multiple_Time<0,1> > gpu_cuda_time_meters;
Meter< System_Energy<GPU_0> > system_energy_meter;
Meter<Wattsup_Energy> wattsup_meter;
```

Listing 3 Code snippets to initialize different kinds of meters

MeterPU can initialize meters not only for individual hardware components, but also for combinations of those components, while keeping the usage of those meters the same as individual ones. For homogeneous hardware components, MeterPU can easily build a meter associated with them, e.g. `gpu_cuda_time_meters` in Listing 3, which can measure kernels that run on multiple GPUs, and only requires the set of GPU ids specified. This feature allows a small constant number of lines of code (LOC) to use MeterPU when increasing the number of homogeneous hardware components to be measured. For heterogeneous components, we can, for example, use combinations of energy meters for CPUs, DRAM and a GPU by `system_energy_meter`[4] in Listing 3.

Allowing combinations of meters brings several benefits: first, it is easier to use, as optimizing energy on a system or a combination of important hardware components is usually preferred. Second, MeterPU will enforce the sequence in which different meters starts, thus possible overhead (like energy overhead for spawning new thread of continuous sampling) is factored out. Third, by metaprogramming for building meters, some runtime overhead like a for loop over many homogeneous meters is removed, which will yeild more accurate measurement values.

The MeterPU API can also be applied for external measurement devices. We developed a plugin for the Wattsup .Net power meter. When Wattsup is connected to a computer that runs MeterPU, we can use `wattsup_meter` in Listing 3 to control the measurement, and get the calculated energy value from measured power values by numerical integration. MeterPU can also be synchronized for device startup time and neglect error samplings (due to Wattsup device hardware error) automatically.

One limitation of the current implementation for system meters for energy measurement is the measurement overhead in terms of energy. When measuring energy of an Nvidia GPU, a CPU thread is spawned to periodically poll power samples from the GPU. Although the energy overhead of thread creation and destruction is not included, the energy spent on the CPU thread to periodically poll GPU power samples is included in the system meter measurement, and leads to an overestimation of energy values consumed by the CPU and the GPU. We however expect that this part of energy is linear in the time of the polling period and can be eliminated a-posteriori in a future version of MeterPU.

3.4 MeterPU Support for Visualization

MeterPU can visualize the measurement samples obtained by different plugins for analysis and debugging purpose. Figure 1 shows a power visualization for a GPU kernel execution. The red curve shows the original samples by

[4] We call this combination system_meter. So far we do not support separate measurement for different CPUs; if necessary we can extend MeterPU in the future

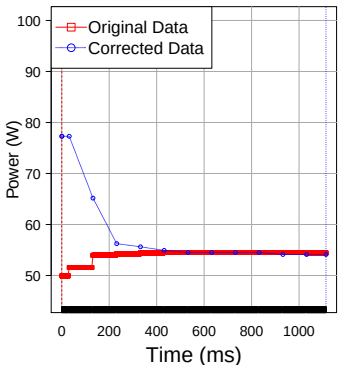


Fig. 1 GPU Power Visualization by MeterPU. The vertical dashed lines denote the time when the kernel under measurement starts and ends.

MeterPU's `NVML_Energy` plugin (internally use Nvidia NVML library), and the blue curve shows the corrected power samples by Burtscher's approach [17], with redundant power samples removed. Based on those corrected values, the final energy value is calculated by numerical integration in MeterPU.

3.5 Discussion

Energy measurement can both be performed by hardware-based measurements (e.g. shunts and A/D converters) and software methods using models and hardware counters. Hardware-based measurements can offer power samples at higher precision and sampling rate than software methods. However, from the optimization's point of view, using energy feedback by software methods is more practical and may dominate in the tuning framework use cases, because usually the energy cost is hardware- and software-dependent, and hard to predict analytically by a single general model (also hardware evolves fast). Thus hardware-based measurement at deployment time is usually necessary, but deploying hardware instrumentations on every machine (where due to heterogeneity, different machines' configurations usually differ) is not very possible, and hosting those hardware instrumentations (e.g. A/D converter) for an energy-tuning framework with on-line training often requires a separate host computer, which may cost much more energy than the savings by the tuning framework.

4 Modification of SkePU for Energy Tuning

SkePU is an open-source C++ based skeleton programming library for GPU-based systems. SkePU supports multi-platform code generation (C, OpenMP, CUDA, OpenCL, StarPU) including support for multi-GPU execution and hybrid execution, from the same high-level source code.

SkePU's adaptive off-line tuning [4] is a full-phase implementation selection methodology consisting of an efficient (time) sampling strategy, deployment-time offline training, and generation of a compact dispatch data structure used in dynamic selection of the expected best implementation variant for a call to a multi-variant skeleton instance.

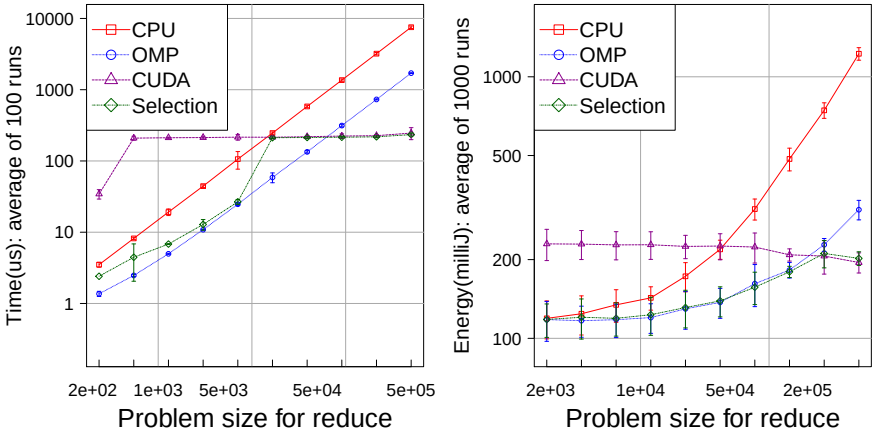
More specifically, SkePU samples the time consumption of each implementation variant within the boundaries of user-specified bounds for run-time contexts such as problem sizes iteratively and adaptively. At the first iteration, it samples the boundary points. If the same winner on different boundaries by sampling are found on these boundaries, then we consider the space represented by those boundaries *closed*, and stop the further sampling in this space. If different winners are found, there exists at least one decision boundary in this subspace where the winner shifts. Then we split the space by equal half in each dimension and sample the vertices of the generated subspaces by splitting. We check the homogeneity of winners following the same procedure until all spaces are closed or some user-specified condition is met. More deeply the sampling process goes, the more accurate the decision boundaries are located with better run-time context-aware dispatch table for implementation variants generated and increased sampling cost. Usually the performance difference is small for the space that is close to the decision boundary, thus stopping at a shallow depth may not hurt the performance significantly, and the dispatch table servers as a reasonable approximation for the real decision boundary. More details of this technique can be found in [18, 5].

With MeterPU integrated, automatic back-end selection in SkePU can migrate from time optimization to energy optimization easily. In SkePU, the time sampling for different skeleton calls with user-defined code snippets is encapsulated in separate wrapper functions in back-ends for different platforms (CPU, OpenMP, CUDA etc). Usually we optimize for the total system energy, not the energy of a specific hardware component. Even if a CPU implementation is invoked, the GPU on the target machine is not turned off and we can not only measure CPU energy when only a CPU implementation is invoked. Thus we use a system meter as described in Section 3.3 for each of the wrappers. Then we replace the time measurement region marker code in SkePU's off-line sampling module [4] with MeterPU `start()` and `stop()` calls, calculate the energy value by `calc()`, and finally pass the value by `get_value()` to the tuning framework. The cost model for communication between CPU and GPU needs to be explicitly modified from time to energy[5]. Now SkePU is ready to tune for reducing energy cost. Furthermore, the code for the initialization of different types of meters can co-exist and be guarded by macros, which makes the switching between different optimization goals as easy as passing a different compiler flag.

[5] Such parameters and the necessary microbenchmarking code and setup will in the future be provided from the XPDL specification of the target system, see [19].

Skeleton type	Description	User function
Map	$b_i = f(a_i), i = 1, ..., n$	return a*a;
	$c_i = f(a_i, b_i), i = 1, ..., n$	return a*b;
Reduce	$d = f(a_1, a_2, ..., a_n), i = 1, ..., n$	return a+b;
...	...	...

Table 1 Setup for different SkePU skeletons. (a, b, c: vector. d: scalar. t: positive integer constant.) More skeleton types can be found in [20]



(a) Time tuning for reduce skeleton.

(b) Energy tuning for reduce skeleton.

Fig. 2 Tuning SkePU skeletons with MeterPU

5 Experimental Results and Discussion

5.1 Experimental Setup

In order to evaluate energy-tuned SkePU with MeterPU, we use SkePU v1.1.1 on an Intel Xeon (E5-2630L v2) server with a Nvidia K20c GPU; the detailed configuration of the machine can be found in [21]. During all experiments, the same configuration for the machine are used with all cores and GPU switched on. We start with the training of SkePU skeletons (the skeletons used and their user functions are listed in Table 1), and the tuning framework will sample training runs by MeterPU to build empirical models for guidance of dynamic implementation switching. Then we generate test cases for different call contexts (problem sizes), and measure the time and energy speedups. The dynamic selection is performed always on CPU, even if the GPU implementation is selected, which will cause some energy cost for dynamic implementation selection spent on CPU, thus we use a system meter as described in Section 3.3 to measure CPU and GPU energy at the same time. Meanwhile we also avoid the problem of having to initialize the correct meter statically for dynamically-known choice of component invocations.

Regarding the energy cost of CPU-GPU communications, as described in Section 3.2, CPU side PCIe communication cost is not measured by MeterPU's currently implemented meters, only the GPU side PCIe communication cost is included. In our experiments, we ignore communication cost. By the usage of smart containers [2], data transfer only happens at the first execution of the same skeleton with the same operands, and we discard the measurement for the first execution. Thus in the subsequent skeleton calls, no data transfers are performed, and the energy changes by software components are well captured by the MeterPU system meters.

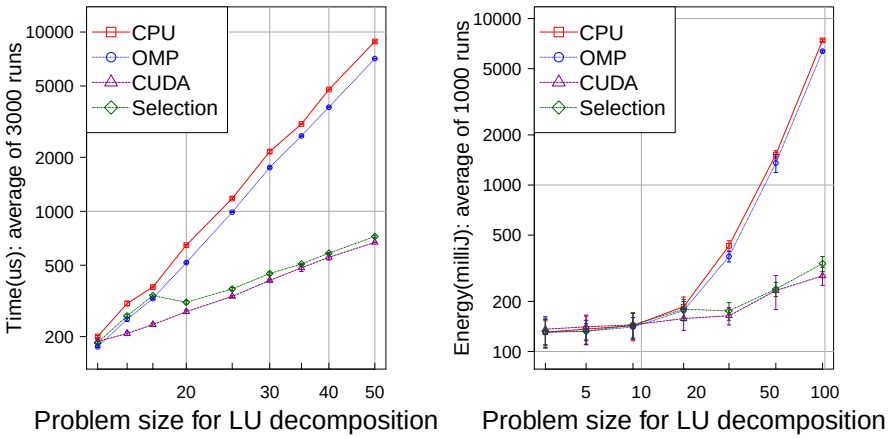
5.2 Tuning for Individual Skeletons

Figure 2 shows the results obtained by applying the user function of Table 1 on a reduce skeleton. More experimental results for other skeletons can be found in [20]. The left hand side shows the time for CPU (sequential), OpenMP, CUDA and time-optimizing selection, and the right hand side shows the energy of these scenarios with energy-optimizing selection. We can see that in most cases the time and energy cost of the smart selection switches to the least-cost software component as the problem size (i.e., operand size) changes. The selection is not optimal only in few test cases, which is an artifact of a too shallow maximum depth of recursive subdivision in adaptive sampling, and better accuracy can be achieved by increasing the training depth [4, 5]. The time and energy savings for large problem sizes can be remarkable, e.g. for a map skeleton, time speedup can reach about 647× (calculated by the time cost of CPU implementation and smart selection at the largest problem size experimented) if a more time-efficient processor type is chosen, and for energy the savings can be about 10×; furthermore, the time speedup and energy reduction factor will continue to increase as problem size grows even larger. The overhead of smart selections for both time and energy is negligible.

We also observe that the transition points usually differ where the most time-efficient, respectively the most energy-efficient, implementation for the same skeleton switches. This makes it necessary for separate training and sampling when switching the optimization goal, and an abstraction of sampling and measurement such as MeterPU is necessary to facilitate the construction of different prediction models. Here the general behavior of time and energy for the computations is not too dissimilar, especially on our K20c GPU where the static (idle) power is already about two thirds of the full-load power, due to power capping, and hence the majority of GPU energy is linear in GPU time.

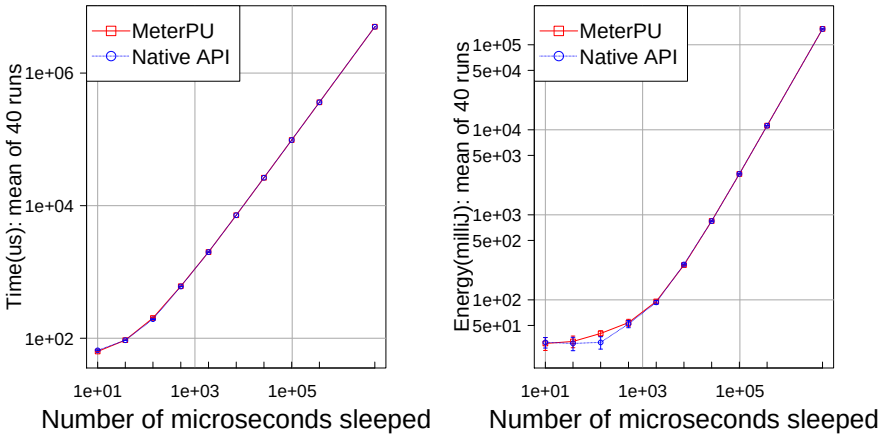
5.3 Tuning for LU Decomposition

In order to show that SkePU can be applied in the area of linear algebra and linear system solving for the reduction of execution time and energy, an experiment is performed on a frequently used linear system solving computation, LU



(a) Time tuning (b) Energy tuning

Fig. 3 Time and energy tuning of LU decomposition by SkePU and MeterPU



(a) MeterPU time overhead (b) MeterPU energy overhead

Fig. 4 MeterPU overhead, including 95% confidence interval on mean value.

decomposition, where the MapArray skeleton is heavily used. Figure 3 shows that in almost all cases SkePU chooses the implementation that has the highest available efficiency in time or energy. At maximum, SkePU provides about 12× in time speedup and about 21× in energy reduction, and the savings will continue to increase as problem size increases above the maximum used in our experiments.

5.4 MeterPU Overhead

MeterPU provides an abstraction of measurement of various metrics on different hardware components. Usually an abstraction comes at a cost, in this case the cost may distort the measurement values. In the MeterPU implementation, when `meter.start()` is called, it delegates the call to the native API, such as `clock_gettime()` etc. Thus the overhead from MeterPU is just one function call, which can even be inlined for some meter types (e.g. `CPU_Time` meter). Figure 4 shows the comparison of measurement values between MeterPU and native API (time by `clock_gettime()`, and energy by Intel PCM library for CPU energy, since MeterPU runs on CPU). We can see that for large problem sizes, the overhead is too small to be observed. For small problem sizes (with less than 100 μs for time and 100 mJ for energy), we can observe that the native measurement facility is not accurate, because we measure a kernel of a sleep function, thus the measured value should be strictly linear in the problem size, but in those regions of small problem sizes, the values are above the expected linear values. To sum up, firstly MeterPU is a negligible-overhead abstraction, secondly, the resolution of those native measurement facilities is limited, and below their resolution limits the accuracy decreases to some extent.

5.5 Comparison to Other Alternatives

An abstraction's main purpose is to hide uninteresting details, thus we use LOC (lines of code) as the main evaluation metric to compare MeterPU with its alternatives: EML and REPARA, which also provide an abstraction measurement layer, described in section 2.1. We measure the lines of code needed for the measurement of a code region, including the calculation of a final metric value, as shown in Figure 5. It is clear that MeterPU requires much less code (only half compared to the second best). MeterPU's advantage on LOC will be larger if several meters of the same type are required, as by variadic template a complex meter will be constructed at compile time, and MeterPU keeps its LOC constant compared to other alternatives. Less LOC facilitates performance and energy modeling software to use. It also helps code transformation tools to automatically transform legacy code to be energy-tuned or tuned towards other user-defined metrics supported or extended by MeterPU.

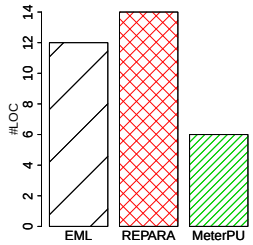


Fig. 5 LOC (lines of code) comparison

Regarding runtime overhead introduced by MeterPU and other alternatives, firstly, as previous discussed, by variadic template expansion, some overhead of meter initialization is moved to compile time. (e.g., for a large number of processors, a for loop is needed, while by variadic template, the for loop is unrolled at compile time). Secondly, MeterPU only involves minimal overhead (only a function call) for measurement. With inlining, most of the function call overhead is removed for some meter types, such as CPU time.

6 Conclusion and Future Work

We developed MeterPU as a software abstraction for measurements of various metrics on different types of processors in a heterogeneous computer system, and demonstrated that the MeterPU API can be applied for both hardware- and software-based measurement methods, and on both single hardware components and (homogeneous and heterogeneous) combinations of those components. It can facilitate both the reuse of legacy tuning frameworks and the switching among different optimization goals on those frameworks as shown by integrating it with SkePU. We thereby also extended SkePU's individual skeletons to be energy-tunable, and showed that energy-tuned SkePU can reduce energy consumption similarly as accelerating execution time e.g. for computations in the area of linear algebra, as demonstrated with LU decomposition as an example.

Future work includes extending MeterPU to support measurement of the energy of PCIe communication on the motherboard side, and testing SkePU's energy tunability on more complex applications. MeterPU may also support multi-objective optimization by for example providing measurements of different metrics for evaluation of scalarized objective function, which can be explored in the future.

Acknowledgements Research partially funded by EU FP7 project EXCESS and SeRC project OpCoReS. We thank Oleg Sysoev from Linköping University for suggestions on statistical data handling. We thank Dennis Hoppe from HLRS Stuttgart, Erik Hansson from Linköping University, Paul Renaud-Goud from Chalmers and all other EXCESS project members for comments on this work.

References

1. J. Enmyren and C. W. Kessler, "SkePU: A multi-backend skeleton programming library for multi-GPU systems," in *Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, Maryland, USA*. ACM, Sep. 2010.
2. U. Dastgeer and C. Kessler, "Smart containers and skeleton programming for GPU-based systems," *International Journal of Parallel Programming*, Mar. 2015, DOI: 10.1007/s10766-015-0357-6.
3. U. Dastgeer, J. Enmyren, and C. W. Kessler, "Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems," in *Proceedings of the 4th International Workshop on Multicore Software Engineering*. New York, NY, USA: ACM, 2011, pp. 25–32. [Online]. Available: http://doi.acm.org/10.1145/1984693.1984697

4. U. Dastgeer, L. Li, and C. Kessler, "Adaptive implementation selection in a skeleton programming library," in *Proc. of the 2013 Biennial Conference on Advanced Parallel Processing Technology (APPT-2013)*, vol. LNCS 8299. Springer, Aug. 2013, pp. 170–183.
5. L. Li, U. Dastgeer, and C. Kessler, "Pruning Strategies in Adaptive Off-Line Tuning for Optimized Composition of Components on Heterogeneous Systems ," *Parallel Computing*, pp. –, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819115001179
6. A. Cabrera, F. Almeida, J. Arteaga, and V. Blanco, "Energy Measurement Library (EML) Usage and Overhead Analysis," in *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, March 2015, pp. 554–558.
7. A. Kiss, M. Danelutto, Z. Herczeg, P. Molnar, R. Sipka, M. Torquati, and L. Vidacs, "D6.4: REPARA performance and energy monitoring library," ©The REPARA Consortium, Tech. Rep., Mar. 2015.
8. D. Josephsen, *Building a Monitoring Infrastructure with Nagios*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007.
9. GroundWork Inc., "GroundWork—Unified Monitoring For Real." http://www.gwos.com/, accessed: 2015-01-21.
10. A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier, "PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level," *ERCIM News*, vol. 2013, no. 92, 2013.
11. M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL - A portable skeleton library for high-level GPU programming," in *16th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*, May 2011.
12. S. Ernsting and H. Kuchen, "Algorithmic skeletons for multi-core, multi-GPU systems and clusters," *Int. Journal of High Performance Computing and Networking*, vol. 7, pp. 129–138, 2012.
13. M. Goli and H. Gonzalez-Velez, "Heterogeneous algorithmic skeletons for FastFlow with seamless coordination over hybrid architectures," in *Euromicro PDP Int. Conf. on Par., Distrib. and Netw.-Based Processing*, 2013, pp. 148–156.
14. R. Marques, H. Paulino, F. Alexandre, and P. D. Medeiros, "Algorithmic skeleton framework for the orchestration of GPU computations," in *Euro-Par 2013 Parallel Processing*. Springer, 2013, vol. LNCS 8097, pp. 874–885.
15. U. Breymann, *Designing Components with the C++ STL*. Addison-Wesley, 1998.
16. Nvidia Corp., "NVML API reference guide," Mar. 2014. [Online]. Available: http://docs.nvidia.com/deploy/nvml-api/index.html
17. M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU power with the K20 built-in sensor," in *Proc. Workshop on General Purpose Processing Using GPUs (GPGPU-7)*. ACM, Mar. 2014.
18. L. Li, U. Dastgeer, and C. Kessler, "Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems," in *High Performance Computing for Computational Science - VECPAR 2012*. Springer Berlin Heidelberg, 2013.
19. C. Kessler, L. Li, A. Atalar, and A. Dobre, "XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization," in *Proc. 2015 Int. Workshop on Embedded Multicore Systems (ICPP-EMS'15), Beijing, Sep. 1-4, 2015, To appear in: Proc. 44th International Conference on Parallel Processing Workshops*. IEEE, 2015.
20. L. Li and C. Kessler, "MeterPU: A Generic Measurement Abstraction API Enabling Energy-Tuned Skeleton Backend Selection," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 3, Aug 2015, pp. 154–159, dOI: http://dx.doi.org/10.1109/Trustcom.2015.625.
21. C. Kessler, L. Li, U. Dastgeer, P. Tsigas, A. Gidenstam, P. Renaud-Goud, I. Walulya, A. Atalar, D. Moloney, P. H. Hoai, and V. Tran, "D1.1 Early validation of system-wide energy compositionality and affecting factors on the EXCESS platforms," Project Deliverable, EU FP7 project Execution Models for Energy-Efficient Computing Systems (EXCESS), www.excess-project.eu, Apr. 2014.