
Lu Li, Christoph Kessler
lu.li@liu.se, christoph.kessler@liu.se
Agenda

- Motivation
- MeterPU
- SkePU and Integration with MeterPU
- Experiments
- Related Work
- Conclusions and Future Work
Acknowledgment
EXCESS Project (2013-2016)

- EU FP7 project
- Holistic energy optimization
- Embedded and HPC systems.
- More info: http://excess-project.eu/
Motivation

- **Parallel programming is challenging**
  - Decomposition, communication, synchronization, load-balancing ...

- **Parallel programming abstraction needed**
  - SkePU: state-of-art skeleton programming framework.
  - Parallel map, reduce etc, on multiple CPUs and GPUs.
  - Automated selection for running skeletons on CPU or GPU for time optimization.

- **Energy** becomes the **main bottleneck** of continuous performance improvement in recent years.

- **How to make a legacy empirical autotuning framework such as SkePU energy-tuned?**
  - Other empirical autotuning framework: PetaBricks [1]
Main idea

- Autotuning software based on empirical modeling optimize on value, not metric
  - Measurement facility give time value $\Rightarrow$ time model built $\Rightarrow$ time opt.
  - The same for energy values.
  - Autotuning logic can be reused.

- A natural solution: unification of measurement interface allow reuse of legacy empirical autotuning frameworks.

- More reasons for such a unification
  - Energy measurement is tricky, especially for GPUs.

- MeterPU is developed in our group.
Figure 1: Unification allows empirical autotuning framework to switch to multiple meter type on different hardware components.
Why is GPU Energy Measurement Tricky?

Figure 2: Data visualization [8] for a program run. Illustrating capacitor effect, and correction methods [2] for true instant power. 

Blue dashed line: program start;  
Red dashed line: program ends;  
green dashed line: power drops again to the static power level.
A software **multimeter**

A **generic** measurement **abstraction**, hiding metric-specific details.

**Four simple functions** to unify measurement interfaces on various metrics on different hardware components.

- Time, Energy on CPU, GPU
- Easy to extend: FLOPS, cache misses etc.

**On top of native measurement libraries.**

- CPU time: `clock_gettime()`
- GPU time: `cudaEventRecord()`;
- CPU and DRAM energy: intel PCM library.
- GPU energy: Nvidia NVML library.
```cpp
template<class Type> // analogous to switch on a real multimeter
class Meter
{
    public:
        void start(); // start a measurement
        void stop();  // stop a measurement
        void calc();  // calculate a metric value of a code region

        typename Meter_Traits<Type>::ResultType const &
            get_value() const;

        // get the calculated metric value

    private:
        METRIC_SPECIFIC_LOGIC_HIDED . . .

};
```

Listing 1 : Main MeterPU API
```cpp
#include <MeterPU.h>

int main()
{
    using namespace MeterPU;
    Meter<CPU_Time> meter;

    meter.start();   // Measurement Start !!
    cpu_func();      // Do sth here
    meter.stop();    // Measurement Stop !!

    meter.calc();
    BUILD_CPU_TIME_MODEL( meter.get_value() );
}
```

Listing 2: An example application that measures CPU Time
An MeterPU Application: Measure GPU Energy

```cpp
#include <MeterPU.h>

int main()
{
    using namespace MeterPU;
    // Only one line differs !!!!
    Meter<NVML_Energy> meter;

    meter.start(); // Measurement Start !!

    cuda_func<<<... , ... >>>(...);
    cudaDeviceSynchronize();

    meter.stop(); // Measurement Stop !!

    meter.calc();
    BUILD_GPU_ENERGY_MODEL(meter.getValue());
}
```

Listing 3: An example application that measures GPU energy
# include <MeterPU.h>

int main()
{
    using namespace MeterPU;
    // Only one line differs !!!!
    Meter< System_Energy<0> > meter;

    meter.start(); // Measurement Start !!

    cpu_func();
    cuda_func<<< ... , ... >>>(...);
    wait_for_cpu_func_to_finish();
    cudaDeviceSynchronize();

    meter.stop(); // Measurement Stop !!

    meter.calc();
    BUILD_SYSTEM_ENERGY_MODEL( meter.get_value() );
}

Listing 4 : An example application that measures system energy
SkePU introduction

- State-of-art skeleton programming framework.
- Supported skeletons: map, reduce, scan, maparray etc.
- Multiple back-ends and multi-GPU support.
- Automated context-aware implementation selection.
- Smart containers.
- ...
Some of SkePU Skeletons

(a) Map

(b) Reduce

(c) MapReduce

(d) MapOverlap

(e) MapArray

(f) Scan
Trivial changes on SkePU code

- In SkePU’s empirical modeling part of code.
- Declare a MeterPU System Energy Meter
- Change the time measurement code
  (previous implemented by native library call: `clock_gettime()`) with MeterPU similar APIs.
  (6 lines of code)
- Done!

Remarks

- ☀️ Change opt. goal:
  change only 1 line of code (meter initialization).
- ☹️ Data transfer is not considered for now.
Integrate SkePU with MeterPU System Meters, enabling tune for time and energy easily.

Train SkePU autotuning predictors by training examples selected by our smart sampling algorithm. [7, 9]
- We train for five skeletons: map, reduce, mapreduce, mapoverlap, and maparray.

Choose test points in exponentially growing sizes.
- Besides skeleton types, we test LU decomposition implemented by maparray skeleton.

Plot the performance of each variant and automated selection.
LIU’s GPU server

- CPU: Intel(R) Xeon(R) CPU E5-2630L v2, 2 sockets (6 cores each), max frequency: 2.4GHz. Support Intel PCM library.
- DRAM: 64GB on 2 sockets
- GPU: Nvidia Tesla Kepler K20 C-class, 13 SMs, 2496 cores. Support Nvidia NVML library.
## Experimental Setup

<table>
<thead>
<tr>
<th>Skeleton type</th>
<th>Description</th>
<th>User function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Map</strong></td>
<td>$b_i = f(a_i)$</td>
<td>return $a*a$;</td>
</tr>
<tr>
<td><strong>Reduce</strong></td>
<td>$d = f(a_1, a_2, ..., a_n)$</td>
<td>return $a+b$;</td>
</tr>
<tr>
<td><strong>Mapreduce</strong></td>
<td>$d = g(f(a_1, b_1), ..., f(a_n, b_n))$</td>
<td>return $a*b$; //for map</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return $a+b$; //for reduce</td>
</tr>
<tr>
<td><strong>Mapoverlap</strong></td>
<td>$b_i = f(a_{i-t}, ..., a_{i+t})$</td>
<td>return $(a[-2]*4 + a[-1]*2 + a[0]*1 + a[1]*2 + a[2]*4)/5$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int index = (int)b; return a[index];</td>
</tr>
<tr>
<td><strong>Maparray</strong></td>
<td>$b_i = f(a_1, ..., a_n)$</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 : Setup for different SkePU skeletons. ($a, b, c$: vector. $d$: scalor. $t$: positive constant.)
Mapreduce Skeletons

(a) Time Tuning for MapReduce.

(b) Energy Tuning for MapReduce.

- Most of time selections are smart, and speedup (max $20 \times$ in time and $10 \times$ in energy) is increasing with problem size.
Reduce Skeletons

(a) Time tuning for Reduce.

(b) Energy tuning for Reduce.

- Empirical autotuning framework for time opt. reused for energy opt.
LU decomposition

(a) Time tuning for LU decomposition

(b) Energy tuning for LU decomposition

- **Easy switching** for optimization goals.
MeterPU Overhead

(a) MeterPU time overhead

(b) MeterPU energy overhead

Figure 4: Nonobservable MeterPU overhead (only one function call)
Related Work and Comparison

- **Other measurement abstraction software**
  - EML [3]: a C++ library tries to unify measurement of time and energy, and easy to extend, helps to build analytical models.
  - REPARA’s [4] performance and energy monitoring library: support both counter-based and hardware-based measurement methods. Overhead (within 5%)
  - MeterPU: the simpler interface while keep generality, our overhead is only one extra function call. Easy creation of aggregate meters based on existing meters.

- **Monitoring frameworks:**
  - Take measurement in either intrusive or non-intrusive way, and store results in database.
  - Nagios [6], Ground-Work [5] etc.
  - MeterPU: more light-weight, lower overhead to retrieve data, tailored to be used in feedback loop for opt. purpose

- A Comparison btw SkePU and other Skeleton Frameworks is detailed in our paper.
Summary of Contributions

- **Hide complexity** for complex energy measurement, especially for GPUs.

- MeterPU enables to reuse legacy empirical autotuning frameworks, such as SkePU.

- With MeterPU, SkePU offers the first energy-tuned skeletons, as far as we know.

- **Switching optimization goal** can be easy, facilitate to build time and energy models conveniently for multi-object optimization.
Some Limitations

- MeterPU requires hardware support for the native libraries, such as Intel PCM, and NVML (sampling power support).
- The minimal kernel runtime for GPU energy measurement is not very small (at least 150ms) [2].
- Counter-based measurement method may not have the same accuracy as hardware-based method. But hardware method is difficult to deploy everywhere.
- MeterPU does not couple with any specific measurement method.
Future work

- Build infrastructure to measure data transfer. (PCIe channel)
- Planned Integration with other software:
  - TunePU: Generic Autotuning Framework, coming soon.
  - GRS: Global Composition Framework,
- Build support for clusters and Intel Xeon Phi.
- MeterPU source code download: http://www.ida.liu.se/labs/pelab/meterpu

Alberto Cabrera, Francisco Almeida, Javier Arteaga, and Vicente Blanco.
Energy measurement library (eml) usage and overhead analysis.
In 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pages 554–558, March 2015.

Marco Danuletto, Zoltan Herczeg, Akos Kiss, Peter Molnar, Robert Sipka, Massimo Torquati, and Laszlo Vidacs.
D6.4: REPARA performance and energy monitoring library.
GroundWork Inc.  
GroundWork—Unified Monitoring For Real.  

David Josephsen.  
*Building a Monitoring Infrastructure with Nagios.*  

Lu Li, Usman Dastgeer, and Christoph Kessler.  
Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems.  
Christoph Kessler Lu Li.

Christoph Kessler Lu Li, Usman Dastgeer.

to appear.