JetsonLeap: a Framework to Measure Energy-Aware Code Optimizations in Embedded and Heterogeneous Systems

Tarsila Bessa[1], Pedro Quintão[1], Michael Frank[2] e Fernando Pereira[1]

[1] UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte, MG, Brazil
{tarsila.bessa,fernando}@dcc.ufmg.br,pedrohquintao@gmail.com
[2] LG Mobile Research, San Jose Lab, 2540 North 1st Str., San Jose, CA 95131
michael.frank@lge.com

Abstract. Energy-aware techniques are becoming a staple feature among compiler analyses and optimizations. However, the programming languages community still does not have access to cheap and precise technology to measure the power dissipated by a given program. This paper describes a solution to this problem. To this end, we introduce JetsonLeap, a framework that enables the design and test of energy-aware code transformations. JetsonLeap consists of an embedded hardware, in our case, the Nvidia Tegra TK1 System on a Chip Device, a circuit to control the flow of energy, of our own design, plus a library to instrument program parts. We can measure reliably the energy spent by 400.000 instructions, about half a millisecond of program execution. Our entire infra-structure – board, power meter and circuit – can be reproduced with about $500.00. To demonstrate the efficacy of our framework, we have used it to measure energy consumption of programs running on ARM cores, on the GPU, and on a remote server. Furthermore, we have studied the impact of OpenACC directives on the energy efficiency of high-performance applications.

1 Introduction

Compiler optimizations improve programs along three different directions: speed, size or energy consumption. Presently, advances in hardware technology, coupled with new social trends, are bestowing increasing importance on the latter [15]. This importance is mostly due to two facts: first, large scale computing - at the data center level - has led to the creation of clusters that include hundreds, if not thousands, of machines. Such clusters demand a tremendous amount of power, and ask for new ways to manage the tradeoff between energy consumption and computing power [1]. Second, the growing popularity of smartphones has brought in the necessity to lengthen the battery life of portable devices. And yet, despite this clear importance, researchers still lack precise, simple and affordable technology to measure power consumption in computing devices. This deficiency provides room for inaccuracies and misinformation related to energy-aware programming techniques [14, 19, 22].

Among the sources of inaccuracies, lies the ever-present question: how to measure energy consumption in computers? Given that the answer to such question does not meet consensus among researchers, conclusions drawn based on potential answers naturally lack unanimity. For instance, Vetro *at al.* [20] have described a series of patterns for the development of energy-friendly software. However, our attempt to reproduce these patterns seem to indicate that they are rather techniques to speedup programs; hence, the energy savings they provide are a consequence of a faster runtime. This strong correlation between energy consumption and execution time has already been observed previously [22]. As another anecdotal case, Leal *et al* [9, 10] have used a system of image acquisition to take pictures each one second of a energy display, in order to probe energy consumption on a smartphone. Such creativity and perseverance would not be necessary, had they access to more straightforward technology. In our opinion, such divergences happen because the programming languages community still lacks low-cost tools to measure energy reliably in computing devices.

The goal of this paper is to fill up this omission. To this end, we provide an infra-structure to measure energy in a particular embedded environment, which can be reproduced with affordable material and straightforward programming work. This infra-structure – henceforth called *JetsonLeap*[3] – consists of an NVIDIA Tegra TK1 board, a power meter, a simple electronic circuit, plus a code instrumentation library. This library can be called directly within C/C++ programs, or indirectly via native calls in programs written in different languages. We claim that our framework has three virtues. First, we measure actual – physical – consumption, at the device's power supply. Second, we can measure energy with great precision at the granularity of about 400,000 instructions, e.g., 100 microseconds of execution. Contrary to other approaches, such as the *AtomLeap* [11], this granularity does not require synchronized clocks between computing processor and measurement device. Finally, even though our infra-structure has been developed and demonstrated on top of a specific device, the NVIDIA Jetson board, it can be reused with other gadgets that provide general Input/Output (GPIO) ports. This family of devices include FPGAs, audio codecs, video cards, and embedded system such as Arduino, BeagleBone, Raspberry Pi, etc.

To validate our apparatus, we have used it to carry out experiments which, by themselves, already offer interesting insights about energy-aware programming techniques. For instance, in Section 4 we compared the energy consumption of a linear algebra library executing on the ARM CPUs, on the Tegra GPU, or remotely, in the cloud. We have identified clear phases on programs that perform different tasks, such as I/O, intensive computing or multi-threaded programming. Additionally, we have analyzed the behavior of sequential programs, written in C, after been ported to the GPU by means of OpenACC directives. We could, during these experiments, observe situations in which the faster GPU code was not more energy-friendly than its slower CPU version. The recipe to

[3] LEAP (Low-Power Energy Aware Processing) is a name borrowed from McIntire [6].

reproduce these experiments is, in our opinion, one of the core contributions of this work.

2 Overview

Power, Energy and Runtime Computer programs consume energy when they execute. Energy – in our case electric power dissipated on a period of time – is measured in watts (W). The instantaneous power consumed by any electric device is given by the formula:

$$P = V \times I \tag{1}$$

Where V measures the electric potential, in volts, and I measures the electrical current passing through a well-known resistance. Therefore, the energy consumed by the electrical device on a given period of time $T = e - b$ is the integral of its instantaneous consumption on T, e.g.:

$$E = \int_b^e V_f I(t)dt = V_f \int_b^e I(t)dt = V_f \int_b^e \frac{V_s(t)}{R_s}dt = \frac{V_f}{R_s} \int_b^e V_s(t)dt \tag{2}$$

Above, V_f is the source voltage, which is constant at the power source. To obtain I we utilize a shunt resistor of resistance R_s. Thus, by measuring V_s at the resistor, we get, from Ohm's Law, the value of $I = V_s/R_s$. One of the contributions of this work is a simple circuit of well-known R_s, plus an apparatus to measure V_s with high precision in very short intervals of time. This circuit can be combined with different hardware. In this paper, we have coupled it with the NVIDIA TK1 Board, which we shall describe next.

The NVIDIA TK1 Board All the measurements that we shall report on this paper have been obtained on top of an NVIDIA "Jetson TK1" board, which contains a Tegra K1 system on a chip device, and runs Linux Ubuntu. Tegra has been designed to support devices such as smartphones, personal digital assistants, and mobile Internet devices. Moreover, since its debut, this hardware has seen service in cars (Audi, Tesla Motors), video games and high-tech domestic appliances. We chose the Tegra as the core pillar of our energy measurement system due to two factors: first, it has been designed with the clear goal of being energy efficient [18]; second, this board gives us a heterogeneous architecture, which contains:

– four 32-bit quad-core ARM Cortex-A15 CPUs running at up to 2.3GHz.
– a Kepler GPU with 192 ALUs running at up to 852MHz.

Thus, from a research standpoint, this board lets us experiment with several different techniques to carry out energy efficient compiler optimizations. For instance, it lets us offload code to the local GPU or to a remote server; it lets us scale frequency up and down, according to the different phases of the program execution, and it gives ways to send signals to the energy measurement apparatus, as we shall explain in Section 3.

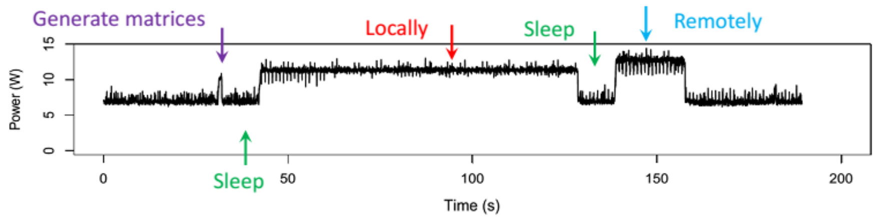


Fig. 1: Example showing the energy consumed at different phases of a matrix multiplication program.

JetsonLEAP in one Example. Before we move on to explain how our energy-measurement platform works, we shall use Figure 1 to illustrate which kind of information we can produce with it. Further examples shall be discussed in Section 4. That figure shows a chart that we have produced with JetsonLeap, for a program that performs three different tasks: (i) it initializes two 3000×3000 matrices; (ii) it multiplies these matrices locally; and (iii) it sends these matrices to a remove server, and reads back the product matrix, which was constructed remotely. Notice that phases (ii) and (iii) represent the same operation, except that in the former case the multiplication happens locally, and in the latter it happens remotely.

We have forced the main program thread to sleep for 10 seconds in between each task. In this way, we have made it visually noticeable the beginning and the ending of each phase of the program. These marks, e.g., a 10 seconds low on the energy chart, lets us already draw one important conclusion about this program: it is better, from an energy perspective, to offload matrix multiplication, instead of performing it locally. However, this modus operand is far from being ideal. Its main shortcoming is the fact that it makes it virtually impossible to measure the energy consumed by program events of very small duration. Additionally, this modus operandi bestows too much importance on visual inspection. We could, in principle, apply some border detection algorithm to detect changes in the energy pattern of the program. However, our own experience has shown that at a very low scale, border detection becomes extremely imprecise. One of the main contributions of this paper is to demonstrate that it is possible to mark – in an unambiguous way – different moments in the execution of a program.

3 The Infra-Structure of Energy Measurement

The infra-structure of energy measurement that we provide consists of two parts: on the hardware side, we have an electric circuit that enables or disables the measurement of energy, according to program signals; on the software side, we have a library that gives developers the means to toggle energy acquisition; plus

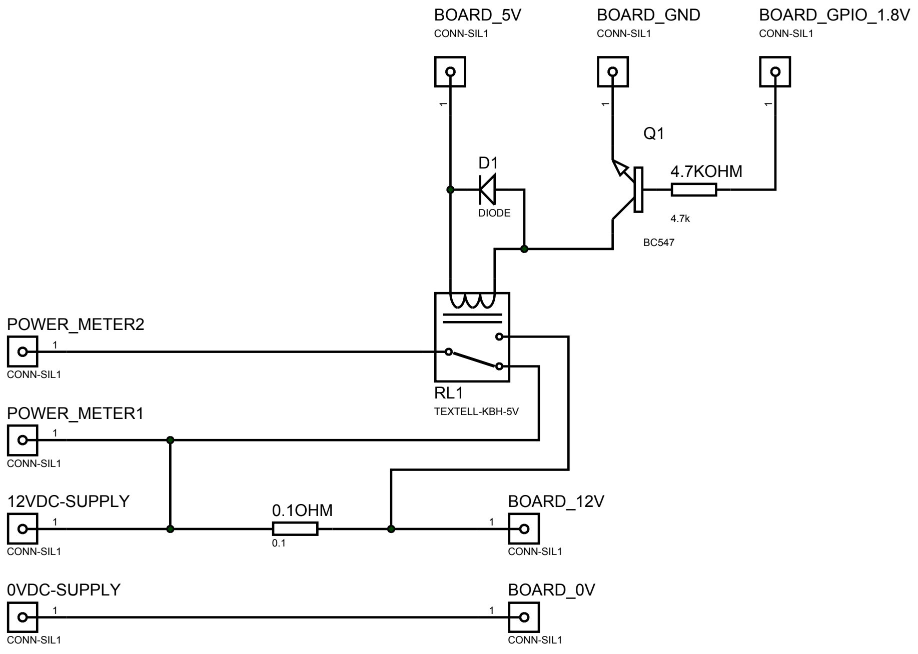


Fig. 2: Schematic view of the circuit that we use to measure energy in the Jetson board.

a program that reads the output of the power meter, and produces a report to the user. In this section we describe each one of these elements.

Hardware. Figure 2 shows the electric circuit that we use to control the measurement of energy. This circuit enables or disables the power measurement, once it links or not the power meter's probes to the shunt resistor edges. This measurement is controlled by signals which are issued from the target program, in such a way that only regions of interest within the code are probed. The circuit is formed by the following components: 1 relay of 5V [4], a resistance of 0.1Ω and 5W, a resistance of 4.7 $K\Omega$ and 0,25 W, a transistor BC547, 1 flyback diode, 10 mini electric cables, 2 connectors with sockets to feed the board, and a protoboard. All in all, these components can be acquired with less than \$ 20.00. Figure 3 (Left) shows how the circuit looks like in practice.

The measurement of power spent by the circuit is controlled by the General Purpose I/O (GPIO) pin of the Jetson board. The GPIO port can be activated from any software that runs on the board. Each hardware defines GPIO ports in different ways. In our particular case, the Jetson has eight such ports, which we have highlighted in Figure 3 (Right). Besides, the 5V supply and the Ground

[4] `http://voron.ua/files/pdf/relay/JQC-3F(T73).pdf`

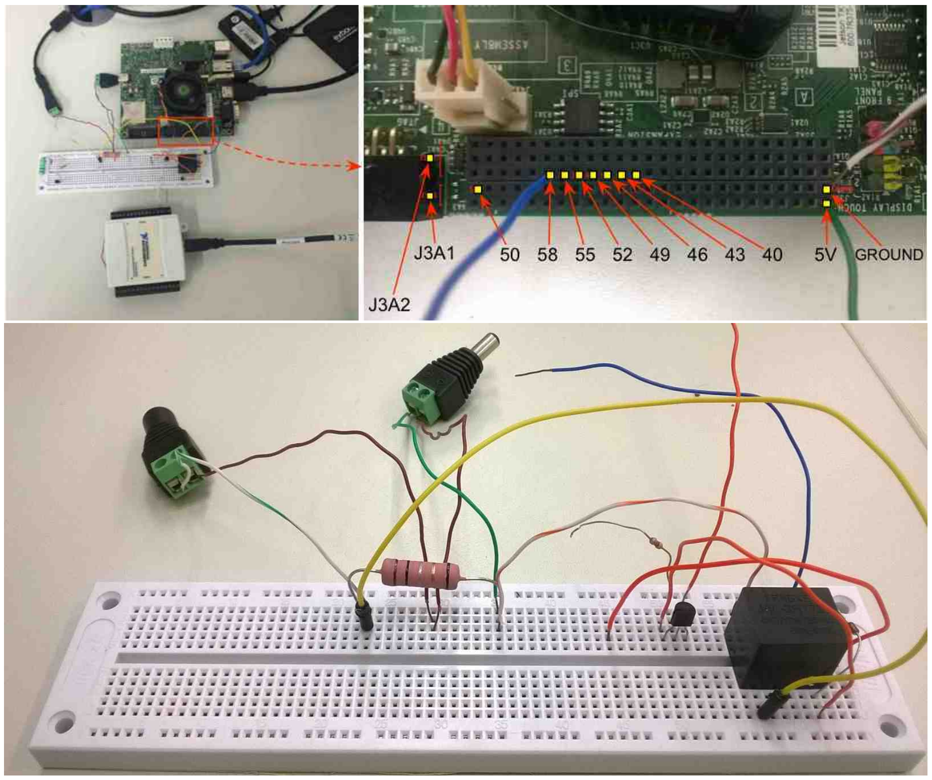


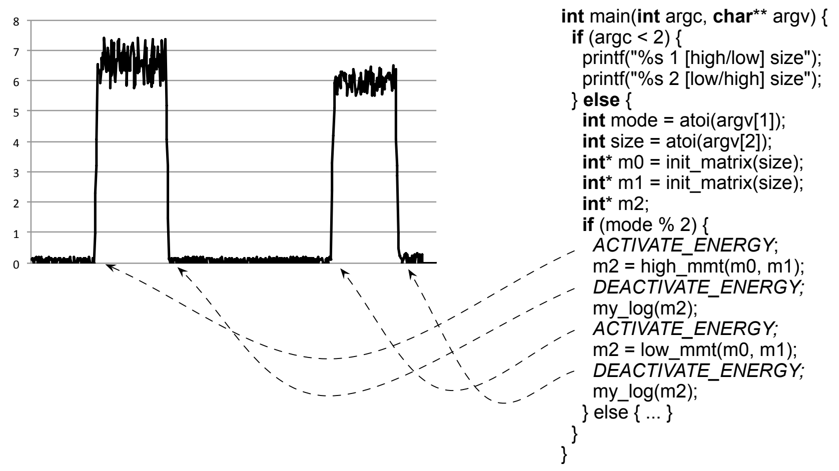
Fig. 3: A picture of our apparatus. (Left) The overal setup. (Right) The ports on the Jetson board. (Down) Detailed view of the circuit.

pins, can be found in the same figure. According to the Jetson's programming sheet, these ports are installed on the pins: 40, 43, 46, 49, 52, 55 and 58, in J3A2, and 50, in J3A1 [5]. Each port can be signalled independently.

Figure 2 shows that in the absence of positive signals in the GPIO port, the two cables of the power meter perform readings at the same logical region, which gives us a voltage of zero. Hence, energy will be zero as well. On the other hand, in face of a positive signal, the transistor lets energy flow until the relay, powering up its coil. In this way, the cables of the power meter become linked with the shunt resistor, enabling the start of the power measurement. From Equation 2, the difference in voltage lets us probe the current at the shunt, which, in turn, gives us a way to know the current that flows into the Jetson board.

Software. The software layer of our apparatus is made of two parts. First, we provide users with a simple library that lets them send signals to the GPIO

[5] `http://elinux.org/Jetson/GPIO`



```
int main(int argc, char** argv) {
  if (argc < 2) {
    printf("%s 1 [high/low] size");
    printf("%s 2 [low/high] size");
  } else {
    int mode = atoi(argv[1]);
    int size = atoi(argv[2]);
    int* m0 = init_matrix(size);
    int* m1 = init_matrix(size);
    int* m2;
    if (mode % 2) {
      ACTIVATE_ENERGY;
      m2 = high_mmt(m0, m1);
      DEACTIVATE_ENERGY;
      my_log(m2);
      ACTIVATE_ENERGY;
      m2 = low_mmt(m0, m1);
      DEACTIVATE_ENERGY;
      my_log(m2);
    } else { ... }
  }
}
```

Fig. 4: Activation/deactivation of power data acquisition through program instrumentation. The exact behavior of the program is immaterial – it is used for illustrative purposes only.

port. Additionally, this library contains routines to record which ports are in use, and to log events already performed. Figure 4 shows a program that toggles the energy measurement circuit twice.

The second part of our software layer is an interface with the data acquisition tool. We are currently using a National Instruments 6009 DAQ. During our first toils with this device, we have been using LabView [6] to read its output. LabView is a development environment provided by National Instruments itself, and it already comes with an interface with the DAQ. However, for the sake of flexibility, and in hopes of porting our system to different acquisition devices, we have coded a new interface ourselves. Our tool, called *CMeasure*, has been implemented in C++. It lets us (i) read data from the DAQ; (ii) integrate power, to obtain energy numbers; and (iii) produce energy reports. Concerning (ii), while on its idle state, our circuit still lets pass to the DAQ some noise, which oscillate between -0.001 and +0.001 watts. The expected value of this data's integral is zero. Thus, by simply integrating the entire range of power values that we obtain through CMeasure, we expect to arrive at correct energy consumption with very high confidence.

4 Experimental Evaluation

In order to validate our energy measurement system, JetsonLeap, we ran several different experiments on the NVIDIA Tegra TK1 board. The first one concerns the precision of our apparatus. We are interested in answering the following

[6] `http://www.ni.com/labview/pt/`

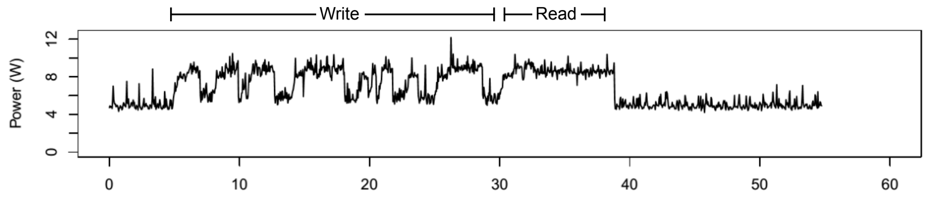


Fig. 5: Energy outline of a program that writes a sequence of records into a file, and then reads them all.

research question: what is the minimum number of instructions whose energy budget we can measure with high confidence. The second batch of experiments demonstrate the many possibilities that our platform opens up to the programming languages community. These experiments compare the energy footprint of sequential and parallel execution on the GPU, and the energy footprint of local compared with remote execution of programs. For simplicity, all the experiments using the Jetson's CPU use only one CPU, even though the board has four cores. We emphasize that these experiments, per se, are not a contribution of this paper; rather, they illustrate the benefit of our framework. Nevertheless, these experiments are original: no previous work has performed them before on the Tegra board. Before we start, we provide evidence that the power dissipated by a program is not constant along its entire execution, even if it is restricted to a single core within the available hardware.

Program Phases. Figure 5 shows the energy skyline of a program that writes a large number of records into a file, and then reads this data. The different power patterns of these two phases is clearly visible in the figure. We show this example to enforce the fact that programs do not have always a uniform behavior in terms of energy consumption. It may spend more or less energy, according to the events that it produces on the hardware. This is one of the reasons that contribute to make energy modelling a very challenging endeavour.

4.1 On the precision of the apparatus

We have used the program in Figure 6 (Left) to find out the minimum number of ARM instructions whose energy footprint we can measure. This program runs a loop that only increments a counter for a certain number of iterations. By varying the number of iterations, we can estimate the minimum number of instructions that gives us energy numbers with high confidence. When compiled with gcc 4.2.1, the program in Figure 6 (Left) yields a loop with only two instructions, a comparison plus an increment.

Figure 6 (Right) gives us the result of this experiment. For each value of `INTERVAL`, we have tried to obtain energy numbers 10 times. Whenever we obtain a measurement, we deem it a hit; otherwise, we call it a miss. We know precisely

```
int main(int argc, char *argv[]){
  if (argc < 2) {
    printf("Syntax: %s INTERVAL\n", argv[0]);
  } else {
    unsigned INTERVAL = atoi(argv[1]);
    ACTIVATE_ENERGY();
    unsigned j=0;
    while(j<INTERVAL){
      j++;
    }
    DEACTIVATE_ENERGY();
  }
}
```

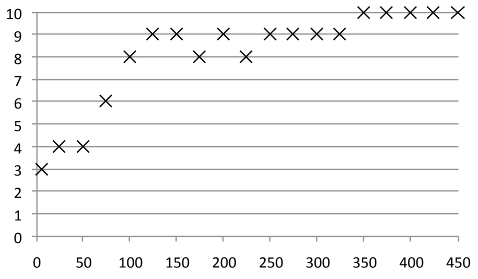


Fig. 6: (Left) Program used to measure precision of our apparatus. (Right) Chart relating the number of correct measurements with the value of `INTERVAL` in the program on the left. The Y axis gives us number of hits, out of 10 tries; the X axis gives us the value of `INTERVAL` (in thousands).

if we get a hit or a miss on each sample because we can probe the state of the relay after we run the experiment. We started with `INTERVAL` equals 5,000, and then moved on to 25,000. From there, we incremented `INTERVAL` by 25K, until reaching 450,000. For `INTERVAL` equal to 5,000, we have been able to switch the relay 3 out of 10 times. After we go past 325,000, we obtain 10 hits out of each 10 tries. These numbers are in accordance with the expected switching time of our relay: less than one milisecond. Given that our ARM CPUs run at 2.3GHz, we should expect no more than 2.3 million instructions per milisecond. From this experiment, we believe that we can measure – with very high confidence – energy of events that take around 400,000 instructions to finish.

4.2 CPU vs GPU

We open this section by comparing the energy consumption of a program running on the CPU, versus the energy consumption of similar code running on the GPU. In this experiment, our benchmark suite is made of six programs, which we took from Etino, a tool that analyzes the asymptotic complexity of algorithms [2]. These programs are mostly related to linear algebra: Cholesky and LU decomposition, matrix multiplication and matrix sum. The other two programs are *Collinear List*, which finds collinear points among a set of samples, and *Str. Matching*, which finds patterns within strings. All these are written in standard C, without any adaptations for a Graphics Processing Unit (GPU). To compile these programs to the Tegra's GPU, we have marked their mains loops with OpenAcc directives. OpenAcc is an annotation system that lets developers indicate to the compiler which program parts are embarrassingly parallel, and can run on the graphics card. We have used accULL [13] to produce GPU binaries out of annotated programs. Therefore, in this experiment we are comparing, in essence, the product of different compilers, – targeting different processors

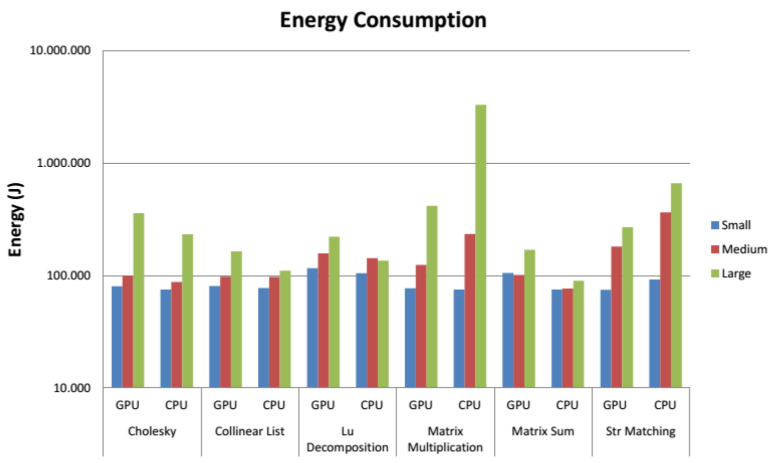


Fig. 7: Energy consumed by different programs, running either on the CPU, or on the GPU.

– when given the same source code. The code that runs on the CPU has been produced with gcc 4.2.1, at the -O3 optimization level.

Figure 7 shows the amount of energy consumed by each benchmark. For each one, we have used inputs of different sizes: small, medium and large. As we can see, usually the GPU binaries spend more energy than their CPU counterparts. The only two exceptions that we have observed are Matrix Multiplication and String Matching. Figure 8 shows the runtime of each benchmark, for each input size, on each processor. The GPU version is faster – for large inputs – in four cases: Cholesky, Collinear List, Matrix Multiplication and String Matching. Notice that this runtime, as well as the energy numbers, represent the entire execution of the kernel, including the time to transfer data between CPU and GPU. However, in either case we omit the time to initialize and check results, which happen in the CPU, even for the GPU-based benchmarks. We can eliminate these phases from our experiment – which are the same for both CPU and GPU-based samples – because of our ability to turn off the energy measurement hardware whenever we find it necessary.

Comparing the runtime chart with the energy consumption one, we realize that, even though the GPU execution is faster for most programs, it usually consumes more energy than the CPU. In fact, only "Matrix Multiplication" and "Str Matching" give us the opposite behavior, in which the GPU consumes less energy than the CPU. This result corroborates some of the conclusions drawn by Pinto *et al.* [12], who have shown that after a certain threshold, an excessive number of threads may be less energy efficient, even for data-parallel applications. Notice that they have gotten their results comparing code running on a multi-core CPU with a different number of cores enabled each time. Figure 9

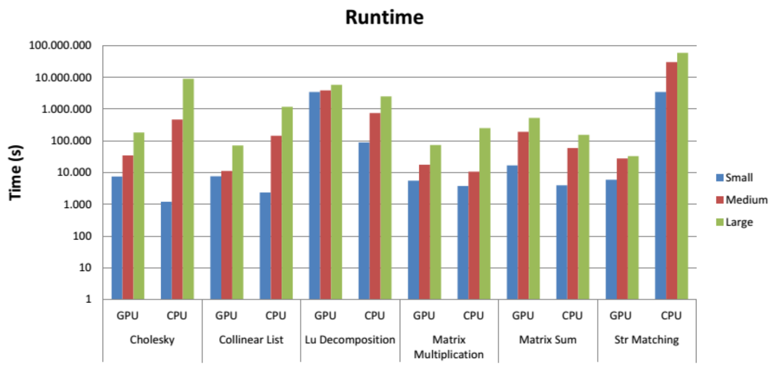


Fig. 8: Runtime for different programs, running either on the CPU, or on the GPU.

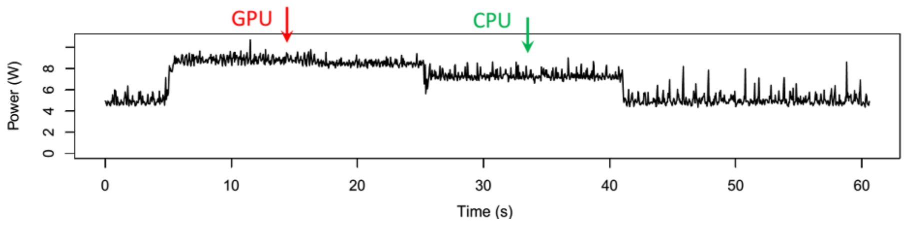


Fig. 9: A chart that illustrates the difference between power consumption by a program running on the GPU and on the CPU.

supports our observation. It shows a program that performs matrix summation, first on the GPU, and then on the CPU. The difference in power consumption makes it easy to tell each phase apart. During the whole execution of the GPU, its power dissipation is higher than the CPU's. We believe that these results are particularly interesting, because they show very clearly that in some scenarios, runtime is not always proportional to energy consumption.

4.3 Local x Cloud

In our third round of experiments, we compare the execution of two different benchmarks, e.g., Matrix Multiplication and Matrix Addition, when running locally on the GPU, on the CPU, or in the cloud. Figure 10 shows how much energy is spent for each program, running on each location. We compare only the energy spent to transfer data between devices, plus the energy spent to run the computation itself. We measure only energy consumed at the Jetson board; thus, in the cloud case, we do not measure the energy spent by the

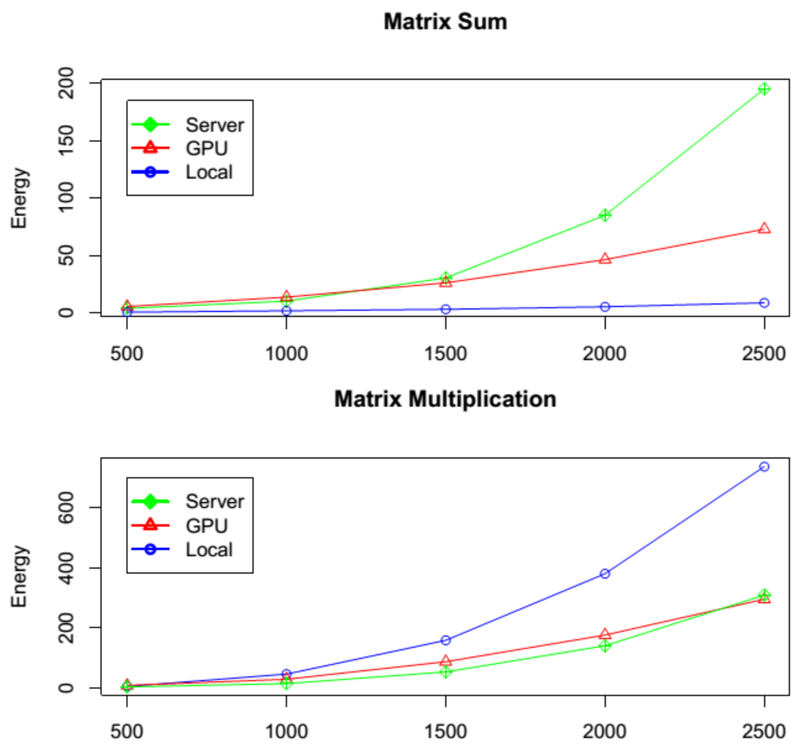


Fig. 10: Energy consumed by different versions of a matrix multiplication and a matrix addition routine.

remote server to perform the computation. In the cloud-based version, most of the energy consumed is spent on networking. As we have seen in Figure 1, the instantaneous power consumed on networking is slightly higher than the power spent by CPU intensive computations.

Figure 10 shows that matrix addition consumes less energy when done locally. This is a consequence of its asymptotic complexity: matrix addition involves $O(N^2)$ floating-point operations on $O(N^2)$ elements. Therefore, its computation over data ratio is $O(1)$. Thus, the time to transfer data between devices already shadows any gains from parallelism and offloading. On the other hand, when it comes to the multiplication of matrices, sending the data to a server is beneficial after a certain threshold. Matrix multiplication has higher asymptotic complexity than matrix addition, e.g., the former performs $O(N^2)$ floating-point operations. Yet, the amount of data that both algorithms manipulate is still the same: $O(N^2)$. Thus, in the case of matrix multiplication we have a linear ratio of computation over data, a fact that makes offloading much more advantageous.

5 Related Work

Much has been done, recently, to enable the reliable acquisition of power data from computing machinery. In this section we go over a few related work, focusing on the unique characteristics of our JetsonLeap. Before we commence our discussion, we emphasize a point: much related literature uses energy models to derive metrics [3, 17]. Even though we do not contest the validity of these results, we are interested in direct energy probing. Thus, models, i.e., indirect estimation, are not part of this survey. Nevertheless, we believe that an infra-structure such as our JetsonLeap can be used to calibrate new analytical models.

The most direct inspiration of this work has been AtomLeap [11]. Like us, AtomLeap is also a system to measure energy in a System on a Chip device. However, Singh *et al.* have chosen to use the Intel Atom board as their platform of choice. Furthermore, they do not use a circuit, like we do, to toggle energy measurement. Instead, they synchronize the Atom's clock with a global watch used by the energy measurement infra-structure. By logging the time when particular events take place during the execution of a program, they are able to estimate the amount of energy consumed during a period of interest. They have not reported on the accuracy of this technique, so we cannot compare it against our approach. We tried to use the Atom board instead of the Nvidia platform as our standard experimental ground. We gave up, after realizing that the amount of energy consumed by that hardware is almost constant, even when there is no program running on it, other than its operating system. Thus, we believe that the Nvidia setup gives us the opportunity to log more interesting results.

There is previous work that attempt to recognize programming events by means of border detection algorithms. This is, for instance, the approach of Silva *et al.* [16], or Nazare *et al* [8]. The idea is simple: if we assume that the hardware consumes more energy when it runs a program, then we can expect an isolated, flat-topped hill on its energy skyline. Thus, the amount of energy on this clearly visible area corresponds to the amount of energy spent by the program. Such a methodology works to measure the energy spent by a program that runs for a relatively long time; however, it cannot be applied to probe short programming events, like we do in this paper. The reasons for this limitation are two-fold. First, internal program events might not produce visual clues that denounce their existence. Second, our own experience reveals that border detection requires a considerable number of sampling points to work reliably. This requirement would reduce greatly its precision when necessary to detect fast events.

A final technique that is worth mentioning relies on *hardware counters*, such as Intel's RAPL (Running average power limit). Different hardware provides different kinds of performance counters, which might log runtime, memory traffic or energy. RAPL registers can be used to keep track of very fast programming events, as demonstrated by Hähnel et al [5]. However, only a limited range of computing machinery provides such tools. Thus, direct measurement techniques such as ours are still essential for simpler hardware. Additionally, direct approaches tend to enjoy more the trust of the research community [21].

Contrary to AtomLEAP and similar approaches [4, 7], our infra-structure does not allow us to measure the power dissipation of separate components within the hardware, such as RAM, disks and processors. This limitation is a consequence of the heavy integration that exists between the many components that form the Nvidia TK1 board. Implementing energy measurement in such environment, at component level is outside the scope of this work. Nevertheless, a comparison with the work of Ge *et al.* [4] is illustrative. They use two data acquisition devices to probe different parts of the hardware simultaneously. Synchronisation is performed through a client-server architecture, via time-stamps. Although the authors have not reported the length of programming events that they can measure, we believe that our approach enables finer measurements, as we do not experiment network delays. Besides, our infra-structure is cheaper: the fact that we control the acquisition circuitry from within the target program lets us use a simpler power meter, with only one channel.

6 Conclusion

This paper has presented JetsonLeap, an apparatus to measure energy consumption in programs running on the Nvidia Tegra board. JetsonLeap offers a number of advantages to developers and compiler writers, when compared to similar alternatives. First, it allows acquiring energy data from very brief programming events: our experiments reveal a precision of about 400,000 instructions. Such granularity enables the measurement of power-aware compiler optimizations. Second, our infra-structure is cheap: the entire framework can be constructed with less than $ 500.00. Finally, it is general: we have built it on top of a specific platform: the Nvidia Jetson TK1 board. However, the only essential feature that we require on the target hardware is the existence of a general purpose input-output port. Such port is part of the design of several different kinds of System-on-a-Chip devices, including open-source hardware, such as the Arduino.

References

1. Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Gener. Comput. Syst.*, 28(5):755–768, 2012.
2. Francisco Demontiê, Junio Cezar, Mariza Andrade da Silva Bigonha, Frederico Campos, and Fernando Magno Quintão Pereira. Automatic inference of loop complexity through polynomial interpolation. In *SBLP*, pages 1–15. Springer, 2015.
3. Adam Dunkels, Fredrik Osterlind, Nicolas Tsiftes, and Zhitao He. Software-based on-line energy estimation for sensor nodes. In *EmNets*, pages 28–32. ACM, 2007.
4. Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Trans. Parallel Distrib. Syst.*, 21(5):658–671, 2010.
5. Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, 2012.

6. Dustin McIntire, Kei Ho, Bernie Yip, Amarjeet Singh, Winston Wu, and William J. Kaiser. The low power energy aware processing (leap)embedded networked sensor system. In *IPSN*, pages 449–457. ACM, 2006.
7. Dustin McIntire, Thanos Stathopoulos, Sasank Reddy, Thomas Schmidt, and William J. Kaiser. Energy-efficient sensing with the low power, energy aware processing (LEAP) architecture. *ACM Trans. Embedded Comput. Syst.*, 11(2):27, 2012.
8. Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Validation of memory accesses through symbolic analyses. In *In OOPSLA*, pages 791–809. ACM, 2014.
9. Jose Leal Domingues Neto. ULOOF: User-Level Online Offloading Framework. Master's thesis, UFMG, 2016.
10. Jose Leal Domingues Neto, Daniel F. Macedo, and Jose Marcos S. Nogueira. A location aware decision engine to offload mobile computation to the cloud. In *NOMS*, pages 831–838, 2016.
11. Peter A. H. Peterson, Digvijay Singh, William J. Kaiser, and Peter L. Reiher. Investigating energy and security trade-offs in the classroom with the Atom LEAP testbed. In *CSET*, pages 1–11. USENIX, 2011.
12. Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA*, pages 345–360. ACM, 2014.
13. Ruym$#225;n Reyes, Iv$#225;n López-Rodríguez, Juan J. Fumero, and Francisco de Sande. accull: An openacc implementation with cuda and opencl support. In *Euro-Par*, pages 871–882. Springer, 2012.
14. H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *SCOPES*, pages 2–11. ACM, 2002.
15. John Sartori and Rakesh Kumar. Compiling for energy efficiency on timing speculative processors. In *DAC*, pages 1301–1308. ACM, 2012.
16. Bruno L. B. Silva, Eduardo Antonio Guimarães Tavares, Paulo Romero Martins Maciel, Bruno Costa e Silva Nogueira, Jeisa Oliveira, Antonio Vicente Lourenço Damaso, and Nelson Souto Rosa. AMALGHMA -an environment for measuring execution time and energy consumption in embedded systems. In *SMC*, pages 3364–3369. IEEE, 2014.
17. S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE*, pages 409–415. IEEE, 2002.
18. Kristoffer Robin Stokke, Håkon Kvale Stensland, Carsten Griwodz, and Pål Halvorsen. Energy efficient video encoding using the tegra k1 mobile processor. In *MMSys*, pages 81–84. ACM, 2015.
19. Madhavi Valluri and Lizy K. John. *Interaction between Compilers and Computer Architectures*, chapter Is Compiling for Performance — Compiling for Power?, pages 101–115. Springer, 2001.
20. Antonio Vetro, Luca Ardito, Giuseppe Procaccianti, and Maurizio Morisio. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In *ENERGY*, pages 34–39, 2013.
21. Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. Measuring energy and power with papi. In *ICPPW*, pages 262–268. IEEE, 2012.
22. Tomofumi Yuki and Sanjay V. Rajopadhye. Folklore confirmed: Compiling for speed = compiling for energy. In *LCPC*, pages 169–184, 2013.