Benchmarking for power consumption monitoring

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

Benchmarking for power consumption monitoring

Description of benchmarks desgined to expose power usage characteristics of parallel hardware systems, and preliminary results

Michèle Weiland · **Nick Johnson**

Received: date / Accepted: date

Abstract This paper presents a set of benchmarks that are designed to measure power consumption in parallel systems. The benchmarks range from low-level, single instructions or operations, to small kernels. In addition to describing the motivation behind developing the benchmarks and the design principles that were followed, the paper also introduces a metric to quantify the power-performance of a parallel system. Initial results are presented and help to illustrate the contribution of the paper.

Keywords Benchmarks · power consumption · energy efficiency metrics

1 Introduction

The quest for Exascale computing has put research into the power consumption of (parallel) software and hardware firmly on the agenda of the HPC community. Recent advances in HPC-specific hardware architectures, and the advent of low-power multi- and many-core architectures and accelerator technologies, have meant that developers of parallel software have had to adapt their programming techniques and models to exploit the full performance of todays systems. Developing parallel software that is efficient in both performance and power usage in an increasingly complex hardware landscape is one of the core Exascale challenges [2]. It would however be inaccurate to believe that this challenge only exists at the top end of parallel computing; it is also a crucial obstacle to overcome for smaller scale parallel systems, including mobile processing devices and applications [1]. In order to be able to implement energy efficient algorithms or choose the most power-performance efficient hardware, it is first necessary to understand and quantify any factors that dictate power consumption. Benchmarks can be used to gain a deeper understanding of how implementation and architecture choices can impact on the overall efficiency of software. If we can identify the power usage profiles of computational patterns, it will become possible to optimise for energy and power in the same way we optimise for performance today.

This paper describes the design and implementation of a set of benchmarks that can be used to measure both performance and power usage on a wide range of hardware architectures, and outlines the motivation for developing new benchmarks rather than using existing suites. The benchmarks introduced here are representative of the whole spectrum of parallel computing (from Embedded to HPC) and provide measurements that can be interpreted on a wide range of platforms. They also cover different levels of compute granularity, from single instructions and operations up to specific computational patterns in the form of kernels.

This paper makes contributions to and progresses the state-of-the-art in the following areas:

- The development of a set of benchmarks which expose system behaviour and allow measurement of power and energy consumption;

The Adept project is partially funded by the European Commission under the 7th Framework Programme, grant agreement number 610490.

Michèle Weiland
EPCC
The University of Edinburgh
Tel.: +44-131-6505030
Fax: +44-131-6506555
E-mail: m.weiland@epcc.ed.ac.uk

Nick Johnson
EPCC
The University of Edinburgh
Tel.: +44-131-6505030
Fax: +44-131-6506555
E-mail: Nick.Johnson@ed.ac.uk

– The desgin of a methodology to quantify power and energy consumption in a range of systems from embedded to HPC;
– Initial results exposing the power-performance characteristics of two different CPUs.

The work presented here is part of the research undertaken in the EU FP7 project Adept ("Addressing Energy in Parallel Technologies")[1], which at its heart has the objective to develop a tool that will allow the prediction of both performance and power usage of parallel software on a wide range of hardware architectures. The development of benchmarks to further our understanding of energy use and power consumption of software and hardware is central to fulfilling this objective.

2 Benchmarking for power consumption

Benchmarks are used to measure and quantify the performance of certain aspects of a given system; here, they are used specifically to measure the performance in combination with the power and energy usage of computations on a wide range of hardware platforms. The purpose of the benchmarks will not be to measure pure runtime performance. Rather they will be used to get an understanding of the power usage of a system (where system includes hardware, software environment, programming models and algorithms) to inform the development of a power usage model for different operations and computational patterns. Spanning the entire landscape of parallel computing from Embedded to large-scale HPC systems, the benchmarks need to reflect this span by being representative of the different computational demands on these systems. While a HPC application may focus on floating-point arithmetic, an Embedded application may be more concerned with thread management and low-level communication.

2.1 Motivation

The decision to develop a new set of benchmarks rather than using existing benchmarks (for example BenchIT [6], LMbench [10], or MultiMaps [8]) was motivated by two factors: firstly, to the best of our knowledge, no single benchmark suite incorporates computational patterns from both Embedded and high-performance computing; as parallel computing is no longer restricted to HPC, but is becoming increasingly commonplace in the Embedded sector as well, it is important that a benchmark suite for power measurement encompasses this branch of computing. Secondly, and more importantly, in order to measure power usage of particular fine-grained operations such as inter-process communication or basic arithmetic, it is necessary to have a clear understanding of any overheads that stem from the basic benchmark initialisation and management code. Developing the benchmarks from scratch means that they can be designed so that any overheads can be minimised and discounted whenever necessary. Existing benchmarks would likely need to be modified considerably in order to allow for clear distinction between what needs to be measured and what needs to be discounted. This second point is particularly important as it dictates the methodology that was followed in the design and implementation of the benchmarks, which will be elaborated on more in Section 3. In order to be able to measure power on a wide range of hardware platforms it is important to have a clear set of well-defined software benchmarks as a starting point to ensure accurate and reliable measurements. Our benchmarks are designed to expose the computational loads and patterns of typical workloads from both the embedded and HPC sectors in order to allow separate, platform dependant tools to extract power and energy usage information.

[1] www.adept-project.eu

2.2 Designing the benchmarks

As mentioned before, an important factor motivating the decision to write new benchmarks rather than use existing ones is that of low-level control over the implementation and the separation of overheads. The following design decisions also dictated the implementation:

Language All the benchmarks are implemented in C; the language was chosen for portability reasons, as well as for being closest to the system. C is used widely in both HPC and Embedded applications. In scenarios where C is too high level and too many overheads are introduced, or where the compiler may optimise the source code in an unpredictable manner, alternative assembler language implementations are provided. This is especially true for very low-level operations, such as basic arithmetic.

Optimisation The benchmarks are tested with a range of different C compilers such as GNU, PGI, Intel and Clang. The performance and power usage of the benchmarks should represent real-life performance whenever possible and compiling the benchmarks with optimisation enabled should be the default. However, for some of the benchmarks (e.g. function calls), enabling optimisation would simply result in the operation that we want to measure being removed (i.e. by inlining the function calls). In such cases the benchmarks are built with all optimisations disabled.

Assembler In order to ensure that the C version of the benchmark codes results in the correct instructions being executed, the development process involves disassembling the source code and inspecting the machine code. This way it is possible to verify exactly which instructions are executed for each benchmark, and which optimisations the compilers perform.

Overheads It is not possible to eliminate overheads entirely. For instance, in order to measure the time and energy used to perform a single multiplication, it is necessary to perform this operation many times in a loop. The overhead (both in term of performance and power) that is introduced by the loop is significant when compared to the multiplication itself. It is therefore necessary to measure the overhead on its own in order to discount it for the overall measurements. This is achieved by adding empty loops containing a `nop` operation to the benchmarks, thus making sure that the time and energy spent in an empty loop can be measured:

```
for(i=0; i<max_rep; i++) {
    __asm__ (''nop'');
}
```

Timing Runtime measurements are currently taken using the stdlib call `gettimeofday()`, however alternatives with finer granularity are being investigated, such as `clock_gettime()` operating in the `CLOCK_MONOTONIC_RAW` mode. When taking measurements, the convention is to take a minimum of 10 readings and retain the measurement with the lowest runtime (i.e. the best possible observed performance).

Warm-up Measuring the power usage of a system "from cold" may give false and misleading results. It is therefore important that the benchmarks put the hardware into a known and stable state by warming it up, i.e. making sure the CPU is running at a relevant clock-speed, and that the pipeline and caches are filled. This is especially important for the low-level benchmarks that measure small operations; in a real-life application those operations are not isolated and measuring them in a "warm" system takes this into account. The warm-up is achieved by executing a moderate number of the benchmark operations before taking any measurements.

3 Adept benchmarks

In the interest of brevity, this section only describes a selection of the benchmarks that were developed to test power-performance and scaling; a full list is given in Table1. These benchmarks were chosen to represent operations and algorithms of interest to both HPC and Embedded systems engineers. Parentheses indicate that this benchmark is partly relevant in this area, or that it is not a commonly used operation or workload.

Table 1 Complete list of micro- and kernel-level power measurement benchmarks and an indication of their relevance for either Embedded or HPC computing.

Benchmark	Embedded	HPC
Bus transfer	✓	✓
Memory	✓	✓
Basic arithmetic	✓	✓
SIMD instructions	✓	✓
Network I/O	✓	(✓)
Disk I/O	(✓)	✓
Jump & Branch	✓	✓
Function calls	✓	✓
Cache misses	✓	✓
IPC	✓	
Thread & process management	✓	
BLAS	✓	✓
File parsing		✓
Pattern matching	✓	✓
Kernel invocation	✓	✓
FFT	✓	✓
Stencil operations	✓	✓

3.1 Arithmetic Operations

The basic algebra benchmark exercises four basic numerical operations: addition, subtraction, multiplication and division. For each operation, different data types may be tested to compare performance. Measuring a single operation may be beyond the measurement capabilities of the system under test, especially for in-band measurement. Therefore a number of operations N of the same data type are performed in a loop, in which the number of iterations R is user-specifiable.

For each numerical operation, multiple tests are performed. The first begins with a single operation ($N = 1$) being executed R times. In each subsequent test, N is increased whilst R is decreased such that the product ($N \times R$) remains constant. The motivation is to expose any differences in performance for what should remain a constant volume of work with a reduced overhead being incurred by the loop due to the execution of fewer iterations. Because this benchmark in particular deals with small, very basic operations, the work loops are implemented both in C and in assembly language. While it is possible to measure single arithmetic instructions using the assembly implementation, the C implementation will include load and store instructions in addition to the arithmetic.

3.2 Memory Benchmarks

This benchmark is designed to test the performance of each level of memory and observe conditions when hierarchical boundaries are crossed, for example, from L1 cache to L2 cache.

The benchmark performs reads or writes to a block of memory, the size of which is user specified. Accesses to this block are made in one of three ways: contiguous, strided, or random, each of which is detailed below. For write benchmarks, a single data value is pre-computed and assigned to each desired element of the array. For read benchmarks, the array is pre-filled with random data.

- For the contiguous-access case, elements of the array are accessed in order of monotonically increasing index. Each element of the array is accessed once, and once only.

- For the strided-access case, the array is treated as a *quasi-circular* buffer. The indices of the elements to be accessed increase by a constant, the stride length, which begins at two elements, and doubles on each pass to a maximum value requested by the user. For example, if the user requests a stride length of 4, the benchmark will be run twice, first using a stride length of 2 and then again using a stride length of 4. Because the array is considered quasi-circular, all elements of the array are accessed for each stride length. Quasi-circular, in this case, means that for each pass through the array, the offset increases by 1. For example, with an array of length 10, and a stride length of 2, the elements would be accessed in the following order: 0, 2, 4, 6, 8, 1, 3, 5, 7, 9. A true circular buffer would see only elements 0, 2, 4, 6 and 8 are each accessed twice. Here, when the end of the array is reached, the offset, initially 0, is increased by 1, allowing access to elements 1 (0+1), 3 (2+1), 5 (4+1), 7 (6+1) and 9 (8+1). Each element of the array is accessed once, and once only, for each stride length.

- For the random-access case, the element of the array to be accessed is determined randomly, once per iteration; the number of iterations is equal to the number of elements in the array. The random-access case does not store a list of previously accessed elements so it is likely that some elements may be accessed more than once and some never accessed.

The memory benchmark also has an option to measure a calloc operation, i.e. assigning and zero-ing a block of memory, for a user-specified amount of memory.

3.3 Function Calls

This benchmark exposes data relating to the overheads incurred in calling a function. Many optimising compilers will attempt to inline functions wherever feasbile, however this is not always possible and being able to quantify the implications of a function call on performance and energy use is therefore of interest. This benchmark uses a single code, an iterative approximation algorithm for π, in all three cases tested: inline, nested and recursive, each of which is detailed below. For each case, the entire approximation is repeated a number of times, R, as specified by the user and within each repeat, the number of iterations of the approximation, N, may also be specified.

- In the inline case, the code is inlined to the caller, giving a baseline for measurement.

- In the nested case, the caller executes a single function, which contains the code for each repeat, R, of the calculation. The called function performs N iterations of the approximation.

- In the recursive case, the caller executes a function for each repeat, R, which calls itself, for each iteration, N. Because systems have a maximum recursion depth, the number of repetitions may have to be broken. When this is the case, and $R_{user} > R_{max}$, a fraction of the repeats R_{frac} is executed M times, i.e. $R_{frac} = {}^{R_{user}}/_{M}$. The extra overhead introduced through additional loop is discounted.

3.4 IPC operations

The inter-process communication (IPC) benchmark exercises three mechanisms of IPC, FIFO buffers, UNIX domain sockets and shared memory segments.

In each case, two processes are spawned using the pThreads library. One thread runs a server code that sends a timestamp via the selected IPC mechanism to the other thread, which runs a client code. The client receives the timestamp and adds it to an array along with a timestamp representing when it received the data from the server. After a user specified number of repetitions the differences between pairs of timestamps (server and client) are computed to give an approximation to the transit time of the IPC mechanism.

Both the socket and FIFO methods have an implicit buffer which negates the need for explicit signalling between server and client for the sending of each timestamp, provided there is consensus about readiness prior to the measurement loop. The shared memory method does require explicit signalling for each timestamp and to establish readiness consensus. Both these requirements are handled using the pThreads conditional signalling and mutex operations.

3.5 BLAS operations

This benchmark is a naïve implementation of selected BLAS routines for dense data, for example: dot product, vector

product, Euclidian norm, matrix-vector product and scalar-vector product. In normal coding, it would always be preferable to make use of a BLAS library, often provided by the CPU or system vendor which has been tuned to provide best performance for the system in question. In this case, we are interested in a simple code that is portable between platforms (to compare system performance), portable between programming methodologies (parallelisation in OpenMP[7] versus UPC[11] for example) and allows coding with different compiler options and optimisation flags. The baseline implementation will allow for direct comparison of the implications that different methods of programming, compilation and execution have on power consumption and performance.

3.6 FFT

This benchmark is again a naïve implementation of a 2-dimensional FFT, based on the well-known Cooley-Tukey algorithm[3]. Whilst alternative implementations exist, this has been well-studied and provides a good comparison with implementation in libraries such as FFTW. Much like the BLAS benchmark in many applications, a tuned, platform specific implementation would be used. However, the desire here is for a portable implementation which has no target-specific tuning, but exposes the computational patterns typical of an FFT computation.

3.7 Stencil algorithms

This benchmark is a naïve implementation of a 5, 9, 19 & 27 point stencil operation. Stencil algorithms are widely used in numerical HPC codes and the computational patterns they exhibit therefore are of interest. The stencil operations are computed multiple times to ensure a runtime large enough for measurement and consist of performing the stencil operation (N-point average) on the complete dataset each time with the intermediate result at each iteration saved to an out-of-place buffer which becomes the input buffer for the next iteration, whilst the current input buffer becomes the storage buffer.

4 Metrics

Perhaps one of the most important issues when benchmarking power consumption and performance is how best to report and analyse the data. Traditionally, performance is reported in units of *operations of interest per second* (*op/s*). In this case, an operation of interest may be bytes written to disk, packets sent across a network interface, or the number of dot-products calculated.

The biggest problem with this approach is that it takes account of neither energy consumption nor other system components. When considering energy consumption, the whole system should be accounted for, rather than a specific component, which implies adding idle, or unused, components to any metric.

We therefore propose a metric of *operations-of-interest per second per Watt* ($op/s/W$) which allows a comparison of the power efficiency of different components in a given system. From this we can derive a metric for the energy scaling performance of a system.

$$E(1) = E_A \times 1 + E_I \times (N_T - 1) \tag{1a}$$

$$E(N_A) = E_A \times N_A + E_I \times (N_T - N_A) \tag{1b}$$

$$\frac{E(N_A)}{E(1)} = \frac{E_A \times N_A + E_I \times (N_T - N_A)}{E_A + E_I \times (N_T - 1)} \tag{1c}$$

Consider Equation 1 where E represents energy consumed, N represents the number of components and the subcripts A, I & T represent active, idle and total respectively. For this explanation, it is assumed that the components in question are cores in a multi-core CPU. The equation is cast in terms of energy and operations of interest, ie the time element has been factored out. This is because we seek to compare the performance and efficiency of a system using a fixed volume of computational work. The runtime of this work will vary with changes in the system configuration (for example choice of CPU or number of active threads) hence normalizing by runtime. We are also concerned with the energy consumed by the system doing the work, not the peak-power which is an instantaneous measurement and a function of the design of the system.

Equation 1a gives the total energy consumed by one active core as the energy consumed by an active core (E_A) plus the energy consumed by $N_T - 1$ idle cores. Equation 1b generalizes this to an arbitrary mix of idle and active cores. Finally, Equation 1c gives the scaling ratio; that is, the ratio of the energy consumption of one active core (with the remainder idle) to an arbitrary number of active cores. In a system where idle cores consume no energy, this would result in a linear scaling with N_A, but this may not be the case in practice.

This approach could be further generalised by including terms for all system components such as memory, GPU and disk in both active and idle states.

5 Early results

In this section we show results from selected multi-threaded BLAS benchmarks, namely **AXPY** and **dot-product** using a vector length of 50 million elements, as well as a simple arithmetic benchmark. The results were obtained by using the power measurement systems available on an ODROID XU+E platform [4]. The motivation for using this system is that it provides easy access to power measurements. The benchmarks were run on the platform's *performance* CPU, an ARM A15 and on the *powersaving* CPU, an ARM A7. Ordinarily, the system is free to migrate loads between processors, however, for this test the load (the benchmark) was fixed to one CPU. In all cases, the energy consumption figures are for the CPU in question only, to give an estimate of performance scaling for the CPU. Additionally, the A15 core is complex and offers out of order execution in a similar manner to a x86-based system. More details of this system can be found in Appendix A. In the case of an HPC system, whole-node power readings may be obtained from the system itself, but more usually provided by the job scheduler upon job completion. The Cray XC30 is an example of a system that already offers this functionality [5].

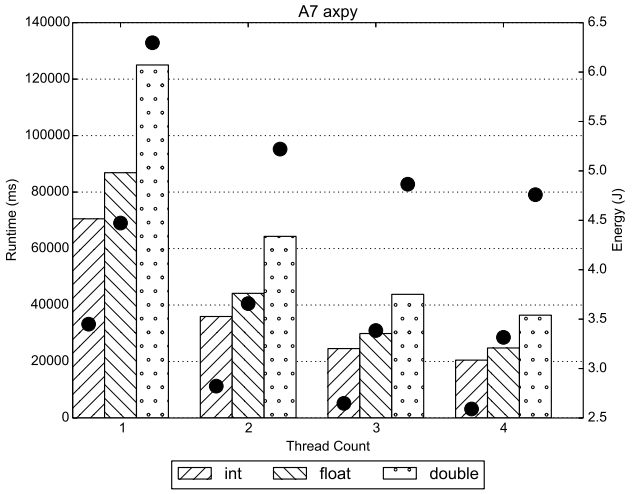

Fig. 1 AXPY benchmark as run on A7 processor for three data types: int, float & double.

In Figure 1 we see the A7 processor performance and power consumption for three data types when running the AXPY benchmark. It can be clearly seen that using the *double* data type consumes more energy and takes longer to complete than the *float* data type, which may naïvely be expected. The energy consumption scales reasonably well for all three data types, although is definitely sub-linear. It may at first seem counter-intuitive that the total energy consumption decreases with increasing core count, however this is a result of the reduced total runtime. It is clear that, on this low-power CPU, there is a benefit in running the AXPY benchmark with as many cores as possible to get the best performance in terms of runtime and energy.

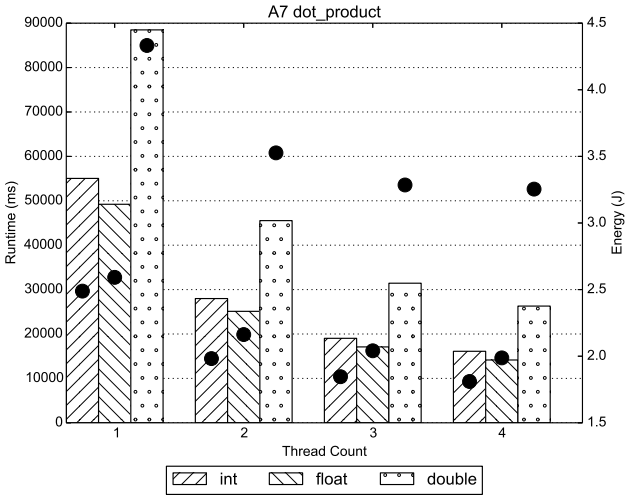

Fig. 2 Dot-product benchmark as run on A7 processor for three data types: int, float & double.

In Figure 2 we see the same processor executing the dot-product benchmark. Again, the scaling performance of both runtime and energy consumption is good, but sub-linear. Interestingly, for this benchmark on the A7 CPU, the performance of the *float* data type is better (faster runtime, but higher energy consumption) than for the integer data type.

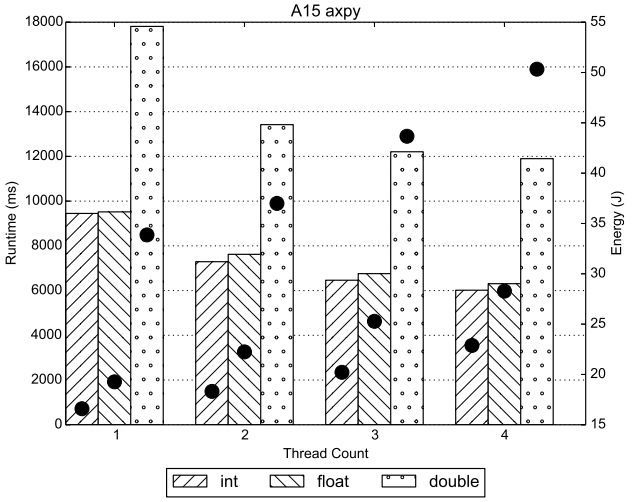

Fig. 3 AXPY benchmark as run on A15 processor for three data types: int, float & double.

In Figure 3 we see the performance of the A15 processor running the AXPY benchmark. Compared with Figure 1, it is clear that this processor consumes a much larger amount of power, of the order of 7 times that of the A7, for the single core case. Runtime performance is much improved compared to the A7 CPU, with this benchmark taking around one order of magnitude less time to complete for the single core case. However, the scaling is poor up to 4 threads with little runtime decrease from additional cores, but a marked increase in power consumption.

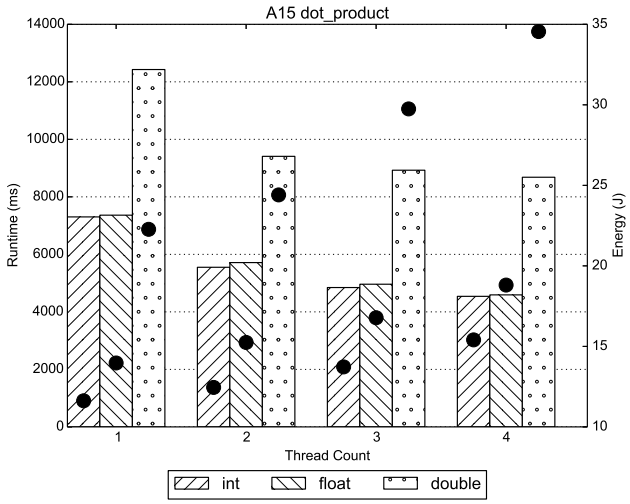

Fig. 4 Dot-product benchmark as run on A15 processor for three data types: int, float & double.

In Figure 4 we see the same processor running the dot-product benchmark. The result, when comparing with Figure 2, correlates to that of the AXPY case: the runtime performance improvement is good, with a marked reduction for the single core case, however the scaling is poor and power consumption markedly increases with increasing core counts.

Taken together, these results show that balancing the power and performance of the two BLAS benchmarks is more complex than might be initially assumed. With a simple processor such as the A7, results scale well in terms of runtime and power consumption as the number of active cores is increased. In both benchmarks shown, the most efficient operating mode for this processor is to use all available cores, regardless of data type. For a more complex processor such as the A15, it is more difficult to balance runtime against power consumption. Increasing the core count can reduce the runtime, but markedly increases power consumption, even with the saving of a reduced runtime.

Figure 5 shows the performance for 1 billion integer additions as part of the **Basic Arithmetic** benchmark. The "nop"

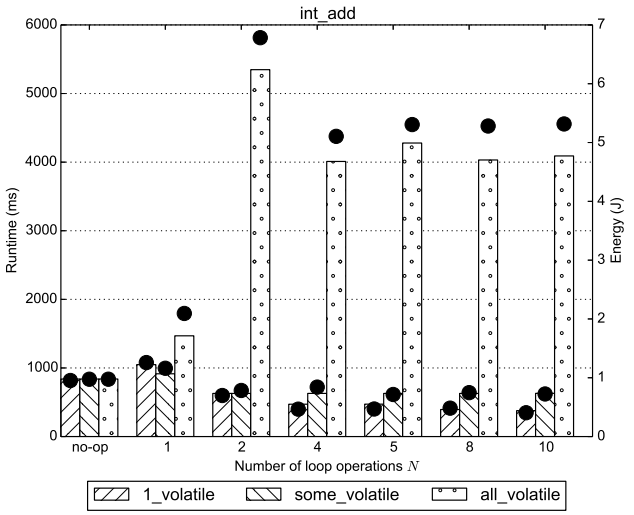

Fig. 5 Performance of integer additions benchmark

instance of the benchmark is used to quantify the overhead that is introduced by an loop with 1 billion iteration. It is possible to see that changing the number of operations inside the loop (N) has a minimal effect on both runtime and energy consumption. What affects these quantities most is the data locality, such as whether a value is in the CPU register, in the cache or in main memory, as well as the number of iterations used to compute the operations. In the `1_volatile` case, only one of the variables inside the work loop is declared `volatile`. In the `some_volatile` and `all_volatile` cases, some or all of the variables are declared as `volatile` respectively; in the `some_volatile` case the number of accesses to volatile variables per iteration remains constant for all values of N (i.e. the total number of volatile memory accesses reduces as N increases), whereas the total number of volatile memory accesses is independent of N in the `all_volatile` scenario. It can be seen then that, as expected, the more variables are volatile, the more the power consumption increases noticeably for ≥ 3 operations in the work loop ($N \geq 3$). This is because non-volatile variable may be highly-efficiently stored in the CPU registers, whereas the use of `volatile` tells the compiler to re-read the variables with every use. The use of the `volatile` keyword is widespread in Embedded systems programming, though less commonly used in HPC.

Figures 6 and 7 show the energy efficiency as described by Equation 1 for the AXPY and dot-product benchmarks. The data used to compute these results is the same as used for the previous figures; the workload is fixed and strong scaling behaviour is shown. What can be seen is that for the benchmarks shown the CPU-power efficiency of the A15 scales sub-linearly (scaling factor > 1), whereas the CPU-power efficiency of the A7 is super-linear (< 1). What the figures represent is the efficiency of changing processors

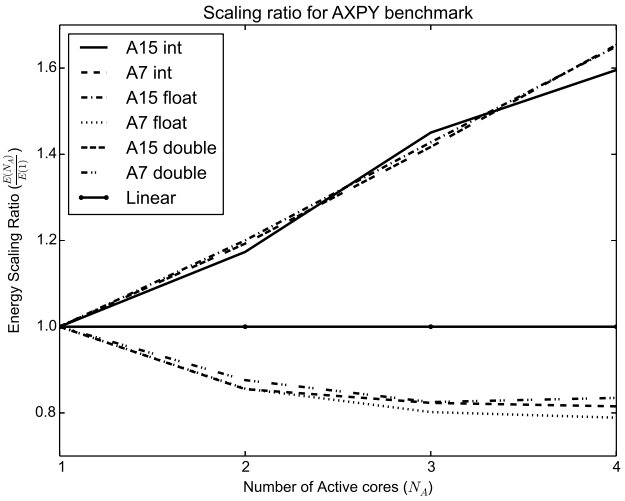

Fig. 6 Energy scaling for AXPY benchmark

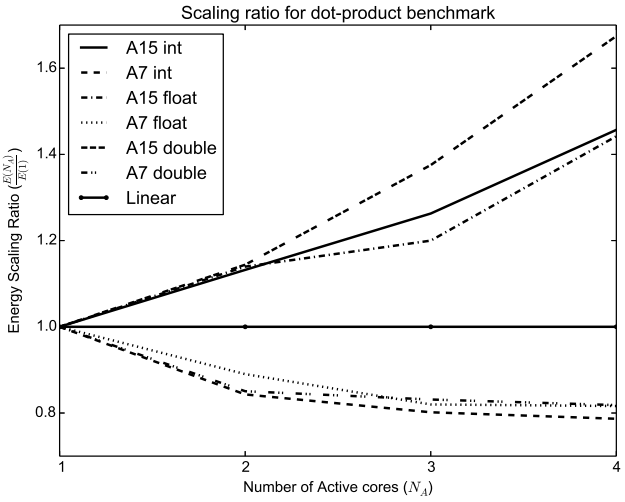

Fig. 7 Energy scaling for dot-product benchmark

from idle to active. If idle CPUs consumed zero energy, and if perfect scaling of energy usage of active cores were assumed, the efficiency ratio would be 1 for all core counts. However this is not the case, firstly because idle cores consume power (namely $0.0354W$ per A15 core and $0.0125W$ per A7 core[2]) and therefore need to be taken into account for the overall energy usage of the CPU, and secondly because $E(NA) \neq N \times E(1)$.

The A15 processor, in both benchmarks cases, consumes $\sim 1.6 \times E(N_A)$ Joules when using 4 actives cores ($N_A = 4$ and $E(N_A) = 24.13$ Joules), which means that for the improved runtime achieved by using 4 threads a 60% energy usage penalty is incurred (with $E(1) = 16.06$ Joules).

[2] These numbers were extracted from the system when the respective CPU was idle

The A7 processor, however, is the opposite. Using more cores results in a lower overall energy consumption, $\sim 0.8 \times E(N_A)$ Joules when using 4 actives cores ($N_A = 4$). The implication here is that the more cores used, the more efficient (in terms of energy consumed) the processor becomes and computing the benchmarks. An indication of this is also that the difference between the idle and active power for a single core on this processor is small, whilst it is much more significant for the A15.

6 Future Work

The work presented in this paper is still in the early stages of research. The baseline implementations of the benchmarks have been developed and tested on a variety of platforms to ensure portability and correctness; the next steps involve developing alternative implementations and parallelisation strategies using different programming models and, where applicable, different algorithms. To date, we have been restricted to perform power measurements on the two ARM CPUs offered by the ODROID platform. As part of the Adept project, we are working on designing a flexible and accurate power measurement solution that will allow us to run the benchmarks on a wider range of platforms.

7 Conclusions

Understanding the power consumption profile of an application on a given hardware architecture is the all-important first step in being able to optimise this application for energy and power usage. Without this prerequisite understanding, trying to achieve good power-performance efficiency is akin to implementing code optimisations without knowing the performance hotspots. This is where the Adept benchmarks, and the associated metrics, come in: they provide detailed, quantifiable and comparable information that will deepen the understanding of software and hardware power usage profiles.

A ODROID Specifications

The board used in the evaluation section of this paper is an ODROID XU+E. This is a complete System-on-Chip based on the Samsung Exynos 5410 Octa processor with two quad-core ARM CPUs [9]: the *performance* CPU, a complex out-of-order ARM A15 running at 1.6GHz, and the *powersaving* CPU, a simple in-order ARM A7, with a clock speed of 200MHz. Both CPUs have 32KB L1 instruction and data caches per compute core. However the L2 cache (which is shared between all core of the CPU) for the A15 is 2MB, as opposed to only 512KB for the A7. The ODROID has 2GB of LPDDR3 DRAM, which runs at 800MHz and has a maximum bandwidth of 12.8GB/s. Ordinarily, the system is free to migrate loads between processors, however, for all results in this paper the load (the benchmark) was fixed to one CPU.

The ODROID has built-in power measurement sensors for both the SoC and board, allowing easy access to power usage data without external instrumentation. These sensors can measure the voltage, current and power consumption of each the CPUs, as well as the memory and the on-board GPU. The sensor readings are reported via the Linux filesystem. The update period for the sensors is set to the default of 262ms although it can be lowered to measure shorter loads at a cost of an increased overhead in sampling, as for any in-band measurement system. The measurements themselves are taken by INA231 sensor modules from TI which use 16bit ADCs with an accuracy of $2.5\mu V$.

A block diagram for the ODROID is shown in Figure 8.

Acknowledgements Thanks to James Perry and Iakovos Panourgias, both EPCC, for testing/reviewing the benchmarks, and to Andrew McCormick from Alpha Data Parallel Systems Ltd for deriving the energy scaling metrics.

References

1. Towards a breakthrough in software for advanced computing systems. Report from a Workshop organised by the European Commission in preparation for HORIZON 2020 (2012)
2. Amarasinghe, S., Campbell, D., Carlson, W., Chien, A., Dally, W., Elnohazy, E., Harrison, R., Harrod, W., Hiller, J., Karp, S., Koelbel, C., Koester, D., Kogge, P., Levesque, J., Reed, D., Schreiber, R., Richards, M., Scarpelli, A., Shalf, J., Snavely, A., Sterling, T.: Exascale software study: Software challenges in extreme scale systems (2009)
3. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation **19**(90), 297297 (1965). DOI 10.1090/s0025-5718-1965-0178586-1. URL http://dx.doi.org/10.1090/S0025-5718-1965-0178586-1
4. Hardkernel: ODROID XU+E Specification. Online. URL http://bit.ly/1sLd62v
5. Hart, A., Richardson, H., Doleschal, J., Ilsche, T., Bielert, M., Kappel, M.: User-level power monitoring and application performance on cray xc30 supercomputers. In: In Proceedings of the Cray User Group (CUG) 2014, Lugano, Switzerland (2014)
6. Juckeland, G., et al.: BenchIT – Performance measurement and comparison for scientific applications. In: G. Joubert, W. Nagel, F. Peters, W. Walter (eds.) Parallel Computing Software Technology, Algorithms, Architectures and Applications, *Advances in Parallel Computing*, vol. 13, pp. 501 – 508. North-Holland (2004)
7. OpenMP ARB: OpenMP Specification (2013)
8. PMaC: MultiMaps. URL http://bit.ly/1hG2vwr
9. Samsung: Samsung Exynos 5 Octa Specification. URL http://bit.ly/OOsOcZ
10. Staelin, C., packard Laboratories, H.: lmbench: Portable tools for performance analysis. In: In USENIX Annual Technical Conference, pp. 279–294 (1996)
11. UPC Consortium: UPC Language Specications (2005)

Dr Michèle Weiland is a Project Manager at EPCC, the supercomputing centre at the University of Edinburgh. She is the Coordinator of the EU FP7-funded Adept project; her main research interest are in power-performance optimisaiton of HPC applications.

Dr Nick Johnson is an Applications Consultant at EPCC, the supercomputing centre at the University of Edinburgh. He currently works on the EU FP7-funded Adept project; his research interests are the power-performance optimisation of applications, and methods for the measurement and quantification of power in computer systems.

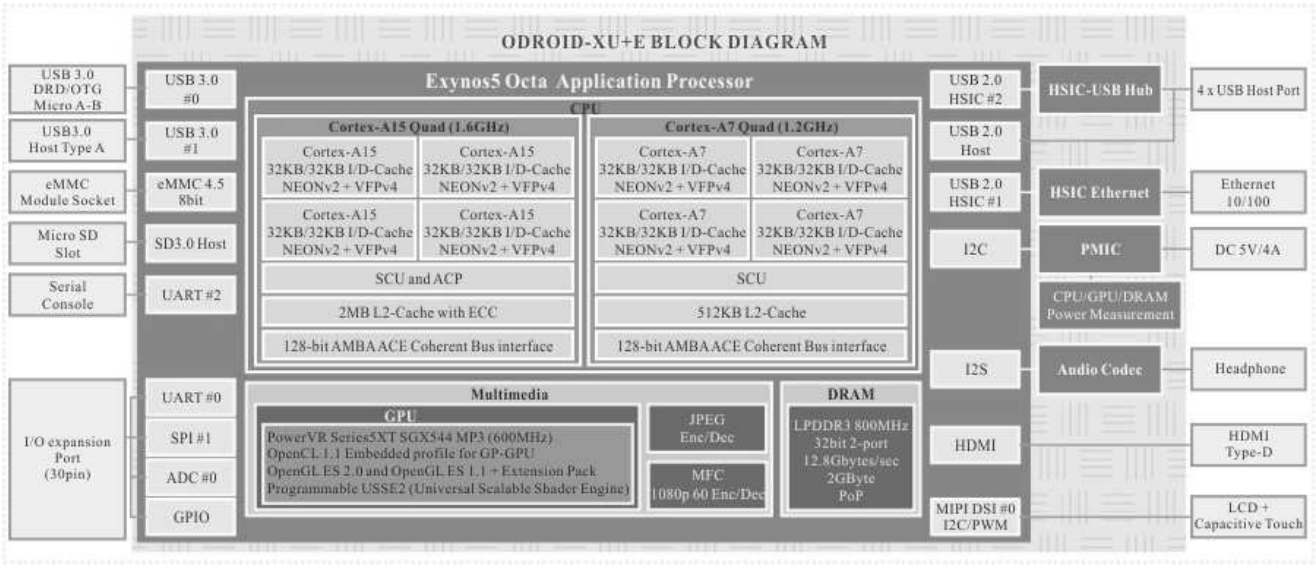

Fig. 8 ODROID block diagram, courtesy of *HardKernel*.