Enhanced Energy Efficiency with the Actor Model on Heterogeneous Architectures

Yaroslav Hayduk, Anita Sobe, Pascal Felber

Enhanced Energy Efficiency with the Actor Model on Heterogeneous Architectures

Yaroslav Hayduk, Anita Sobe and Pascal Felber
first.last@unine.ch

University of Neuchâtel, Switzerland

Abstract. Due to rising energy costs, energy-efficient data centers have gained increasingly more attention in research and practice. Optimizations targeting energy efficiency are usually performed on an isolated level, either by producing more efficient hardware, by reducing the number of nodes simultaneously active in a data center, or by applying dynamic voltage and frequency scaling (DVFS). Energy consumption is, however, highly application dependent. We therefore argue that, for best energy efficiency, it is necessary to combine different measures both at the programming and at the runtime level. As there is a tradeoff between execution time and power consumption, we vary both independently to get insights on how they affect the total energy consumption. We choose frequency scaling for lowering the power consumption and heterogeneous processing units for reducing the execution time. While these options showed to be effective already in the literature, the lack of energy-efficient software in practice suggests missing incentives for energy-efficient programming. In fact, programming heterogeneous applications is a challenging task, due to different memory models of the underlying processors and the requirement of using different programming languages for the same tasks. We propose to use the actor model as a basis for efficient and simple programming, and extend it to run seamlessly on either a CPU or a GPU. In a second step, we automatically balance the load between the existing processing units. With heterogeneous actors we are able to save 40-80% of energy in comparison to CPU-only applications, additionally increasing programmability.

1 Introduction

Energy efficiency of data centers and clouds has become a major concern. As claimed in 2012 by a Greenpeace report [6], current cloud computing systems consume the same amount of energy as a whole country such as Germany and India. While going *green* is from the users' and operators' perspective often done voluntarily or for economic benefits, today's systems reach physical limitations—the so-called "power wall"—that enforce focusing on energy-efficiency [5].

Usually, work on improving energy efficiency is limited to isolated strategies. For instance, on a data center level, power consumption is reduced by adaptively shutting down nodes; on a single system level, power consumption is

reduced by providing more efficient hardware or runtime support and by dynamically adapting the CPU frequency using dynamic voltage and frequency scaling (DVFS) [4]. While these approaches are effective per se, we believe that software and hardware have to be considered together to best enable energy-efficient resource usage. In general, the energy consumption E of an application relates to its power consumption P and its execution time T ($E = P \cdot T$). Hence, to reduce energy consumption, one can either radically (1) reduce the power consumption (usually at the cost of execution time) or (2) reduce the execution time (usually at the cost of power consumption).

As shown by Trefethen et al. [24] the CPU frequency has a major impact on power consumption. We therefore exploit the CPU frequency scaling features of Linux where possible and use predefined "governors".

To reduce the execution time, a possible way is to exploit all available hardware resources, e.g., graphical processing units (GPUs). Programming applications that run both on CPUs and GPUs is a challenging task as parts of a program might be better targeted at a CPU, while other parts are data parallelizable and run more efficiently on GPUs. As it might be necessary to provide two versions of the same application (e.g., CUDA/C++), we focus our contributions especially on programmable solutions for heterogeneous applications.

As a basis we rely on the *actor model* [10]. The model offers a high degree of isolation between its main entities, called actors. Actors enable seamless interoperability between heterogeneous components [1], allowing us to differentiate between actors running on a CPU or GPU and consequently support *heterogeneous actors*. Actors are useful for data parallelizable applications; they, however, might cause overhead if applications are iterative and maintain state.

In this paper we investigate several strategies for implementing heterogeneous actors focusing on iterative applications. We start from a manually crafted and optimized implementation, in which an actor running in Scala calls the CUDA/GPU code written in C/C++ using the Java native interface (JNI). Later, we propose to decouple this design by using a middleware component, RabbitMQ.[1]

Another solution is to use a domain-specific language (DSL) for generating both CPU and GPU code. Frameworks such as Delite [21], which provide automatic code generation, expect the entire application to be written with the DSL and executed by the provided runtime. With actors it is desirable to be able to decide on a fine-grained level whether a task, encapsulated in an actor, should execute on a CPU or on the GPU. Therefore, we adapt the actor model by introducing heterogeneous actors, which can be programmed using Delite DSLs.

From a programmer's point of view we show that the heterogeneous actors based on DSLs represent the simplest solution and lead to a reduced energy consumption of up to 40% in comparison to CPU-only actor implementations, with JNI actors allowing for savings of up to 80%. We present our final contribution, which is a scheduler that balances workload among GPU and CPU resources.

The rest of the paper is organized as follows. We introduce the generic ideas on power consumption and execution time reduction in Section 2, providing im-

[1] http://www.rabbitmq.org

plementation details on heterogeneous actors in Section 3. The load balancing of actor tasks is introduced in Section 4. In Section 5 we describe the hardware and software setup used for evaluation. We present and analyze results in Section 6 and discuss related work in Section 7. We conclude in Section 8.

2 Improving Energy Efficiency

One way to reduce the energy consumption is to decrease the power consumption ($E = P \cdot T$). This can be achieved either by influencing the hardware (e.g., by changing the frequency of a CPU), or by lowering the resource usage of the application itself (e.g., only use a single CPU with a sequential program). Another way is to focus on the improvement of the application's performance. In the following sections we discuss mechanisms for both approaches in detail.

2.1 Reducing Power Consumption

The overall power consumption of a machine is highly influenced by the power consumption of the CPU. Although CPUs become more and more energy-efficient, the overall energy consumption increases as we usually trade power for performance [3]. We focus on two strategies that are easy to configure: (1) the level of parallelism and (2) the voltage/frequency of a CPU.

If the level of parallelism (i.e., thread count) of an application is not properly chosen, the performance and the power consumption are negatively affected. For example, the system scheduler might interfere with the program execution, impeding the application's performance.

The power consumption of a CPU can be influenced by changing the CPU frequency. The Linux kernel provides a tool, `cpufreq`,[2] allowing us to configure *governors* that automatically set the desired CPU frequency. Specifically, we are interested in three governors. (1) **Performance**: the CPU will be automatically set to the highest available frequency; (2) **Powersave**: the CPU will be automatically set to the lowest available frequency; (3) **Ondemand (DVFS)**: the governor monitors the CPU utilization and, if it is on average more than 95%, the frequency will be increased. The dynamic approach with the *ondemand* governor is the most promising, as it provides DVFS to fit the needs of an application.

2.2 Reducing Execution Time

If the performance gain is significant, it can be translated into a reduction of the total energy consumption. Concurrent programming is one measure to reduce execution time. Programming with threads and locks, however, is challenging.

The actor model has been introduced by Hewitt et al. [10] as a popular mechanism for implementing parallel, distributed and scalable systems. An actor is an independent, asynchronous object with an encapsulated state that can only be

[2] https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

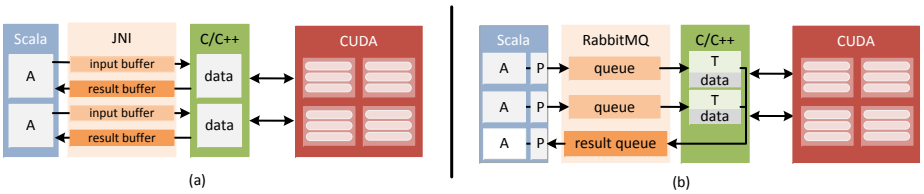

Fig. 1: Heterogeneous actors using (a) JNI and (b) RabbitMQ.

modified locally based on the exchange of messages. Considering a typical data-parallel algorithm as an example, we can easily design an application with a set of dedicated worker actors performing the required computations and a separate coordination entity actor that distributes the data and collects the results. In contrast to a classical multithreading approach we do not need to account for synchronizing shared memory accesses. For our actor implementations we use Akka[3], an official platform to manage actors in Scala.

Actors allow for interoperability not only on a single CPU but also across its boundaries. Communication, however, is not yet supported between different kinds of processors such as GPUs.

3 Enabling Heterogeneous Actors

To reduce the energy consumption while ensuring programmability, we exploit heterogeneous computing (CPU/GPU programming) with the help of actors. For GPU programming CUDA is a de facto standard.[4] CUDA provides a C/C++ binding for communicating with the GPU. As a GPU is a co-processor, the CPU is always necessary for communication, management, and data exchange. While with C/C++ and CUDA the program would be tightly interwoven, the actor model provides inherent decoupling by separating tasks into actors. As stated before, the communication between actors on different processors is not straightforward. As such, we provide support for actors that are able to run on either a GPU or a CPU, calling them *heterogeneous actors*. In what follows, we present three different possibilities for implementing heterogeneous actors.

JNI. The Java native interface (JNI) can be used for communicating with native libraries written in C/C++, supporting the communication with the GPU. In data-parallel programs, actors responsible for interacting with the GPU are initialized with a portion of input data (see Figure 1(a)). A copy of the actor-local data is propagated to the actor-local GPU memory as well. In each GPU actor the final result is then stored in a result buffer, which can be accessed from either C or Scala using JNI.

RabbitMQ. An alternative for decoupling CPU and GPU code is to use a middleware component like RabbitMQ.[5] RabbitMQ enables the communication

[3] http://akka.io
[4] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[5] http://www.rabbitmq.org

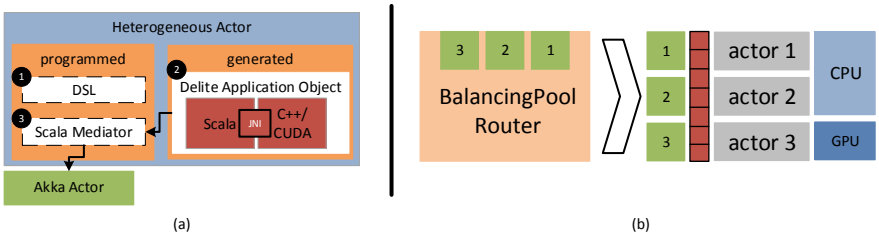

Fig. 2: (a) Heterogeneous actors using DSLs from the programmer's view. (b) BalancingPool Router in Akka.

(via queues) between programs written in different languages and amongst distributed machines. By using RabbitMQ we can connect CPU actors with GPU actors. Communication is supported via a proxy (P in Figure 1(b)) that passes data from Akka to RabbitMQ. On the C/C++ side, each actor is associated with one thread (T) that waits for work in its RabbitMQ queue and, once available, fetches and forwards the data to the GPU for processing. The data is still isolated and accesses do not have to be synchronized. Upon completion, threads dispatch their result to the shared RabbitMQ result queue. The results are then collected and merged by a coordination actor in Akka. With RabbitMQ it is still necessary to provide both the CPU and the GPU implementations. It also requires the development of custom code to interact with the communication middleware (the proxy is not part of Akka).

DSL. For the DSL implementation we base our efforts on Delite [20], a framework that provides high level DSLs and runtime for heterogeneous programming. Delite expects that the programmer writes the entire application in the provided DSLs and executes the generated code in a dedicated runtime environment. As it is not always feasible to write the entire application in a DSL, our goal is to provide finer control to the programmer such that only some parts of his application have to be written in a DSL. In particular, only heterogeneous actors will be written in one of the Scala-like intuitive DSLs provided by Delite.

To support the execution of generated code from the actor environment, we need to provide custom communication facilities. Delite currently supports communication to generated Scala code with an intermediate packaging step into a *Delite application object*. To interact with this object (stored in a JAR file), so-called *Scopes* are needed as entry points [21]. They are limited, however, to mapping simple Scala data types to generated Scala code and they cannot forward data from Scala to the generated C++/CUDA code. As a first measure, we enhanced Scopes with a JNI method for forwarding data to the generated C++/CUDA code. We further extended Scopes to automatically load the generated C++/CUDA code and enable the interaction between Scala and C++/CUDA, which was not supported by Delite.

Another limitation of Delite is the lack of support for applications that maintain state between iterations. Typically, upon start-up the generated C++ code allocates main memory and GPU memory for storing input and intermediate data. Before completion, Delite cleans all the memory that it used during its

execution. We adapted Delite such that the state-relevant memory (e.g., input dataset chunks copied to GPU memory) is only cleaned after the last iteration of the application has been executed. With this measure we avoid copying data between CPU and GPU at each actor message exchange.

In Figure 2 we show the overall heterogeneous actor approach. The programmer must provide actor code targeting the GPU in the Delite DSL (1), which will generate and build the Delite application object (2). A lightweight mediation part in Scala (3) is required to convert Akka messages into data structures for the Delite application object and vice versa. We also provided support for Delite-generated CUDA code to return the result to the calling Scala code.

4 Resource Load Balancing with Heterogeneous Actors

Since the CPU and the GPU have different performance characteristics, load imbalances can happen. Hence, this section focuses on efficient workload balancing strategies for runtime and energy reduction.

To distribute work among actors on a CPU, Akka provides so called *Routers* that schedule messages targeted to a set of actors accomplishing a similar task. Specifically, the *BalancingPool router* embraces "work-stealing"[6] by balancing workload dynamically among worker actors. When an actor accesses its mailbox to fetch the next available message to be processed, Akka transparently forwards that request to a shared message queue started by the BalancingPool Router (see Figure 2). Since the mailbox queue is shared, any worker actor should be capable of processing any message in the queue. Hence, Akka imposes a requirement for worker actors to be stateless, thus limiting its usage for iterative applications.

To overcome this limitation, we propose to use the following strategies. First, to enable iterative applications to be used with routers, we encapsulate all state required for the execution into messages. Each message contains the required context for having it processed on either the GPU or the CPU. For example, to avoid copying input data on each iteration, we store a pointer to it in a message. Also, to avoid synchronization issues between the CPU and the GPU memory, the message also contains a result object, stored in CPU memory, to which all implementations write intermediate results for the next iteration.

For actors running on both processing units, both implementations are required and any of the strategies discussed in Section 3 can be used. Also, since for iterative applications, the behavior will be repetitive, it is likely that the number of actors running on a CPU/GPU will not change at runtime. Hence, it is sufficient to find the optimal actor CPU/GPU configuration at startup. As such, at application start we introduce a brief profiling phase. For each configuration (e.g., 0 GPU actors/8 CPU actors; 1 GPU actor/7 CPU actors; etc.), we measure the execution time using 1% of messages to be processed. Once finished, we select the configuration with the lowest execution time and use it for the processing of remaining messages.

[6] The actual implementation more precisely follows a *work-sharing* approach.

To summarize, our approach enables actors, independent of whether they run on the CPU or GPU, to request work when required, thus leading to reduced idle time and more balanced workloads.

5 Experimental Setup

Hardware. Our experiments are executed on a server equipped with an AMD FX-8120 (8 cores, no hyperthreading) CPU and an NVIDIA GeForce GTX 780 Ti (2880 CUDA cores) with 3GB of RAM. We use a hardware power meter (Alciom PowerSpy v2.0) that periodically reports the system power in Watts.

Software. We base our evaluations on the well-known k-means [2] algorithm used for splitting an input dataset into different clusters. K-means is a good case study as it exhibits iterative and processing-intensive characteristics representative for data-parallelism. We further focus on k-means as it is a well-understood algorithm that can be represented in a straightforward manner in Delite's OptiML DSL. As such, we chose depth over breadth regarding our analysis, presenting the results of k-means only. Despite exclusively focusing on k-means, the core premise of heterogeneous actors is applicable for implementing other iterative algorithms (e.g., coordinate descent, logistic regression, deep belief learning with a restricted Boltzmann machine).

We used the thread-based STAMP [17] implementation of k-means as a basis for creating the actor version. For the actor-based implementation the following data structures are required: (1) input matrix; (2) current cluster center matrix; (3) points to cluster center *map* (holds the current cluster center index for each input point); (4) per-cluster member count structure (holds the number of points assigned to each cluster).

Parallel implementation with actors. Our actor-based algorithm uses two types of actors: *iteration actors* (Algorithm 1) and *worker actors* (Algorithm 2). While a typical thread-based version maintains a shared copy of the current cluster center matrix and the per-cluster member counts, the actor-based version maintains a private copy of these data structures in each of the worker actors. The iteration actor then merges the data sent by each of the worker actors to calculate the final cluster centers.

Algorithm 1: K-means iteration actor.

```
Data: input set, number of clusters, number of workers
Result: clusters
Initialize K cluster centers
foreach Worker do
 | Create workers and pass partial input set
while Termination condition is not met do
 | Send current cluster centers to worker actors
 | foreach Worker do
 |  | Receive partial results
 | Compute final cluster centers by merging partial results
```

Algorithm 2: K-means worker actor.

Data: partial input set, current cluster center
Result: local cluster centers, local member count
foreach *Assigned input point* **do**
    Assign point to the closest cluster center
    Update the local cluster centers matrix and member count
Send local cluster centers and cluster counts to iteration actor

Heterogeneous implementation with JNI. For the heterogeneous implementation we extend the baseline actor implementation. Specifically, we execute the worker actor code on the GPU, while leaving the iteration actor unchanged for execution on the CPU. We further preserve the communication patterns between worker actors and the iteration actor. We reimplemented the worker actor to access the GPU resources by calling the C/CUDA code using JNI with the help of shared byte buffers as shown in Figure 1(a). Each worker actor connects to a C implementation that starts two CUDA kernels, one for finding the closest cluster center for each input point (on block memory), and one for finding the total number of points that changed clusters as compared to the previous iteration (on GPU global memory). Once the GPU execution has finished, the results are transferred to the result buffer and to the iteration actor.

Heterogeneous implementation with RabbitMQ. In this implementation we reused the CUDA code of the worker actor from the JNI implementation, but adapted the communication pattern between Scala and C/CUDA. Each worker actor now includes a proxy (as shown in Figure 1(b)) that is responsible for marshaling the messages and sending them to the RabbitMQ queue. Once work is available, the aforementioned CUDA implementation is launched, omitting the shared byte buffers. In the end, a proxy actor connecting to the iteration actor transfers the results.

Heterogeneous implementation using a DSL. We define the worker actor's logic using OptiML [19]—a Delite DSL. Next, we write the mediation code to connect to the generated code (Figure 2). The mediation code extracts the current cluster centers from an Akka message, converts them to a Delite array (to map the `Rep` data structures in the DSL), and then calls that generated code with the array as input. Once the result is available, the mediation code converts it to an Akka message and forwards it to the iteration actor.

Heterogeneous work balancing implementation. For the implementation of the work balancing use case any of the before mentioned implementations can be used. We decided to use JNI as it showed the best performance (see Section 6). We define the worker actor code just like in the heterogeneous implementation with JNI, but the worker actors are able to execute on both the CPU and the GPU. To enable load balancing, we require stateless actors, hence moved their state to messages (see Section 4). The profiling uses 1% of the overall workload for testing each possible configuration, hence we allow the programmer to set the number of desired iterations manually.

System Configuration. K-means is a representative candidate for this evaluation as it is able to work on different input sizes. For the first three implemen-

tations of k-means (CPU/GPU), we chose a default data set from the STAMP benchmark with 65,536 input rows and 16 clusters. To test the profiling and selection process of the best share of CPU/GPU actors we use three different datasets: *small* (4,096 rows), *medium* (10,240), *large* (131,072). We set the worker actor count to match the CPU core count (i.e., 8). To enable efficient load balancing, the iteration actor divides the work into more tasks than the number of worker actors (32 tasks per iteration). As the run times can be considerably reduced with a GPU, we increased the load to gather reasonable results. The profiling takes 450 iterations per configuration, with an overall benchmark length of 5,000 iterations. We run each implementation 5 times and take the median execution time and power readings; the energy is then calculated out of these two values.

6 Results and Discussion

In this section we discuss the results of the different k-means implementations from Section 5 with respect to power consumption and execution time, and relate them to energy consumption.

6.1 Reducing Power Consumption

We investigate the reduction of power consumption by varying the number of workers, as well as the governors impacting the frequency of the CPU. The default governor is *ondemand*; its goal is to provide good performance when work is available and downscaling of the frequency otherwise (DVFS).

On the left side of Figure 3 we present the power consumption of the three CPU-only Scala implementations (seq: sequential, par: parallel thread-based, act: actor). We scale the number of threads/actors (4, 8, 16, 32) in separate runs and average the results for each chosen frequency. The sequential implementation consumes the least power since only one core is used while the others are idling. The *ondemand* governor depends on CPU utilization and, since k-means is CPU-intense, power consumption of the *ondemand* and *performance* governors is comparable. The *powersave* governor sets the CPU to the lowest frequency, hence power consumption is reduced. The difference between the *powersave* and other two governors is 70 W for the parallel implementations and around 40 W for the sequential implementation.

The middle part of Figure 3 shows the impact of the governors on the execution time. The sequential algorithm using the *powersave* governor is about 3 times slower than with any other governor. The parallel implementations exhibit a slowdown of about 2 times if the *powersave* governor is used.

Based on power and execution time measurements, we can compute the energy consumption as shown in Figure 3 (right). The actor implementation outperforms the parallel and the sequential implementations. We can further see that, while the *powersave* governor decreases the power consumption, the execution time is significantly higher. This leads to higher energy consumption than when using the *ondemand* and *performance* governors. In general, the *ondemand* governor seems to be the best choice independent of the type of implementation.

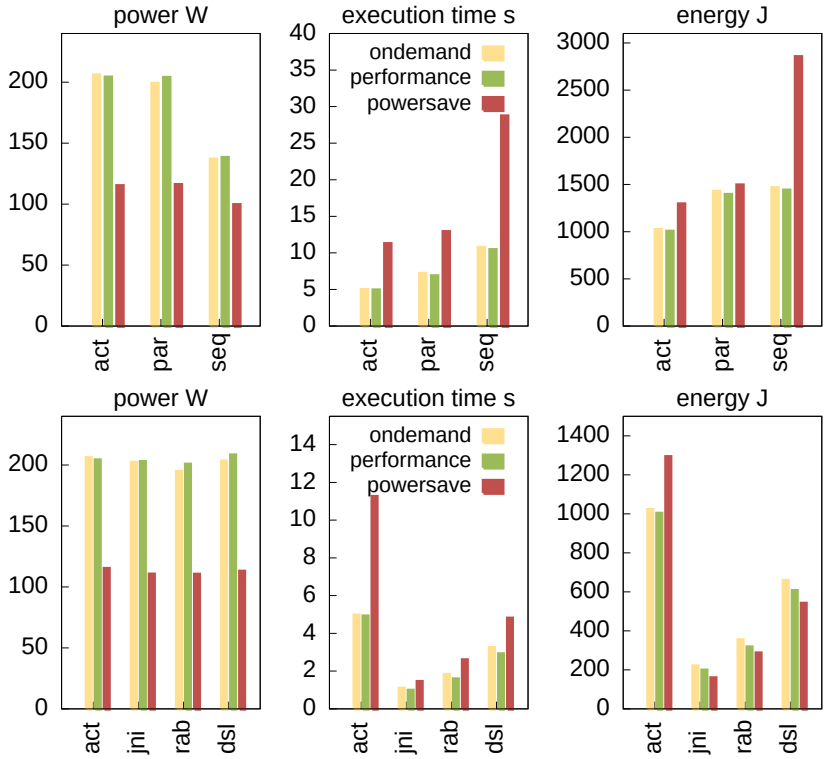

Fig. 3: Comparison of the power consumption (left), execution time (middle), and energy (right) using different governors. The top graphs refer to CPU execution. The bottom graphs include one bar for CPU-only execution and three bars for mixed execution (CPU/GPU).

6.2 Reducing Execution Time

This section focuses on execution time reduction and its impact on energy consumption. In the heterogeneous implementation, CPU actors cooperate with the GPU in different ways. We compare the JNI implementation (jni) with RabbitMQ (rab) and DSL actors (dsl) as described in Section 5. We also vary the frequencies for the CPU running the remaining code.

Figure 3 shows the power consumption (left), execution time (middle) and energy consumption (right). We see that the power consumption is not impacted by the usage of the GPU. The reason is that in this hardware setup the GPU is more energy-efficient than the CPU, hence running code on the CPU is more expensive in terms of power consumption. All GPU implementations execute faster than CPU implementations, yielding lower total energy consumption (Figure 3 (right)). The DSL implementation in *powersave* mode consumes 540 J, which is lower than the 1,001 J of the actor implementation in *ondemand* mode. In contrast to the CPU-only execution, we see that reducing the CPU frequency with the *powersave* governor does not have a drastic impact on performance.

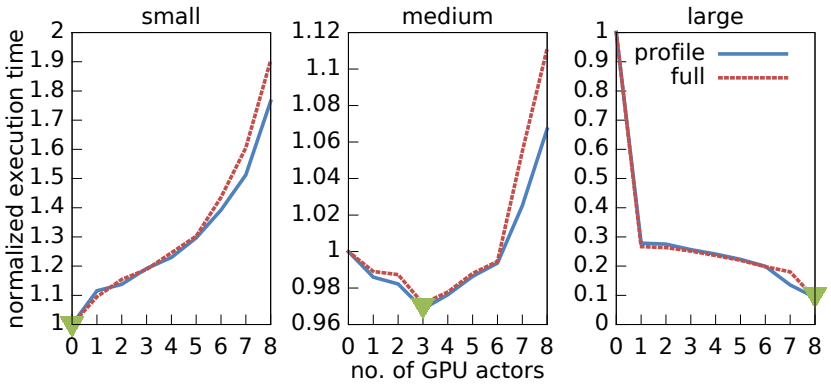

Fig. 4: Normalized execution time of the profiling phase as well as of the remaining workload using different configurations of CPU/GPU actors.

Therefore, the best choice would be the *powersave* governor in the heterogeneous scenarios.

With respect to the different implementations, JNI provides the most direct way of communication with the GPU. This implementation does not provide the decoupling nor the flexibility for seamlessly exchanging the code to be executed on the GPU or the CPU. While the RabbitMQ implementation provides the possibility of exchanging worker (GPU) code, it still requires the programmer to implement the actual GPU code in C/CUDA. In comparison, by using heterogeneous actors with the DSL, programmers only need to provide the worker code for message processing in one of the Delite DSLs.

In terms of the lines of code required to implement k-means, JNI implementation is the most energy-efficient one, it requires 336 lines for the k-means logic written in CUDA and C, as well as 67 lines for providing JNI functionality (communication logic). RabbitMQ uses the same k-means logic as JNI, but requires another 337 lines for the communication logic. In comparison, our DSL implementation does not require communication logic because it is automatically generated by Delite. Hence, the overall effort for the DSL implementation is as low as 39 lines of code.

From a programmer's point of view heterogeneous actors with DSL represents the best solution, with better energy efficiency than CPU-only implementations. However, from an energy-efficiency point of view heterogeneous actors with JNI are preferable, as with it is possible to reduce the energy consumption of up to 80% in comparison to a solution running on the CPU.

6.3 Resource Load Balancing with Heterogeneous Actors

This section presents our proposed load balancing approach. We execute the profiling phase as well as the full run on all datasets and combine the prediction capability of the profiling phase. Clearly, the execution time of the full running phase will be a multitude higher than the execution time of the profiling phase.

Therefore, we normalize all values to the execution time of the first configuration. Figure 4 presents the results for different dataset sizes showing that the profiling phase can mimic the execution time of the remaining workload reasonably well. In more detail, for the *small* dataset we see that the problem is not scaled significantly to amortize overheads associated with executions on a GPU. When using the *medium* dataset we see that the most efficient configuration is composed of 3 GPU actors and 5 CPU actors. When we use a *large* dataset (e.g., also the STAMP sample dataset from our former experiments), it is always beneficial to process all the workload using GPU actors.

The energy consumption reveals the same trends, however, we further noticed that with increased load, the CPU tends to use its turbo frequencies and hence draws about 20 W more than in the experiments before. In the case of the *large* dataset the CPU has a significantly higher power value if it shares work with the GPU while the execution time is not reduced significantly. Therefore, from an energy consumption point of view any sharing of work with the CPU in our hardware setup would be disadvantageous. With the *medium* and *small* dataset the load is not as high, hence sharing the work between the CPU and the GPU leads to slightly increased power consumption but lower execution time, and in total to lower energy consumption.

7 Related Work

In general, research on hybrid computing rarely considers energy efficiency. Researchers focus more on performance improvements (e.g., [25]) or develop power estimation models [14, 11]. The trend of using graphical processing units (GPUs) for scientific programming became popular as there is a potential for significant performance improvement over executing only on a CPU. With the radical reduction of execution time, GPUs can in turn reduce the total energy consumption, providing means for energy-efficient programming [13, 18]. Nevertheless, if the gains in execution time of GPU implementations are not high enough, the energy consumption might increase as compared to a CPU-only implementation [18].

The PEACH framework [9], for example, combines performance and power metrics to guide the scheduling on both CPU and GPU, but it focuses on defining a theoretical model rather than a practical implementation capable of working with real-world applications. Researchers working on SEEP [12] aim at helping programmers to produce energy-aware software. Their approach considers continuous energy monitoring of specific code paths helping to identify energy-hungry code. They mainly target, however, embedded systems capable of executing a single task. On the programming language level the authors of [8] divide a program into phases for which specific CPU frequencies are assigned. This approach does not only necessitate fine-grained monitoring of energy and execution time, but also requires that a program exclusively occupies a single core of a CPU. In [15] the authors propose a hybrid OpenMP/MPI programming model for power-aware programming. They use this model to steer the level of parallelism as well as the current frequency of a CPU.

The SPRAT [22] environment can automatically select the proper execution processor (either CPU or GPU) at runtime for energy efficiency. Migration is, however, quite expensive as the current state of the application must be saved when moving from one processor to another. There are a number of approaches for scheduling work between the CPU and the GPU. They can be broadly divided into performance/cost models (e.g., HEFT [23]), offline training [16], as well as work stealing [7]. A performance/cost model requires determining the approximate runtimes and data transfer times beforehand for each processing unit. For developers this requirement is hard to meet.

8 Conclusion

In this paper we tackle the problem of reducing energy consumption of parallel programs in heterogeneous environments. As energy depends on both power consumption and execution time, we investigate the impact of each independently. We first reduce the power consumption with the help of frequency scaling. We then reduce the execution time by running parts of an application on a GPU, while the sequential parts remain on the CPU. We evaluate a number of strategies for heterogeneous actors regarding their energy efficiency and programmability. JNI and RabbitMQ provide a more direct way of accessing a GPU, while the DSL implementations provide a concise and simple way for building heterogeneous actors. In a first step all heterogeneous implementations require manual assignment to the best processing unit. Hence, our final contribution enables automatic sharing of resources among actors yielding the highest energy efficiency. Our contributions lead to significant reductions of energy consumption in the range of 40-80% as compared to CPU-only implementations.

References

1. Agha, G.: Actors programming for the mobile cloud. In: Symposium on Parallel and Distributed Computing (ISPDCP). pp. 3–9. IEEE (2014)
2. Alpaydin, E.: Introduction to machine learning. MIT press (2004)
3. Barroso, L.A., Hölzle, U.: The case for energy-proportional computing. IEEE Computer 40(12), 33–37 (2007)
4. Beloglazov, A., Buyya, R., Lee, Y.C., Zomaya, A., et al.: A taxonomy and survey of energy-efficient data centers and cloud computing systems. Elsevier Advances in Computers 82(2), 47–111 (2011)
5. Cai, C., Wang, L., Khan, S.U., Tao, J.: Energy-aware high performance computing: A taxonomy study. In: International Conference on Parallel and Distributed Systems (ICPADS). pp. 953–958. IEEE (2011)
6. Cook, G.: How clean is your cloud? Report, Greenpeace International, April 2012
7. Faxén, K.F.: Wool-A work stealing library. ACM Computer Architecture News 36(5), 93–100 (2009)
8. Freeh, V.W., Lowenthal, D.K.: Using multiple energy gears in MPI programs on a power-scalable cluster. In: Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 164–173. ACM (2005)

9. Ge, R., Feng, X., Burtscher, M., Zong, Z.: PEACH: A model for performance and energy aware cooperative hybrid computing. In: Conference on Computing Frontiers. pp. 1–24. ACM (2014)
10. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 235–245. Morgan Kaufmann Publishers (1973)
11. Hong, S., Kim, H.: An integrated GPU power and performance model. In: International Symposium on Computer Architecture (ISCA). pp. 280–289. ACM (2010)
12. Hönig, T., Eibel, C., Kapitza, R., Schröder-Preikschat, W.: SEEP: Exploiting symbolic execution for energy-aware programming. In: Workshop on Power-Aware Computing and Systems (HotPower). pp. 1–4. ACM (2011)
13. Huang, S., Xiao, S., Feng, W.: On the energy efficiency of graphics processing units for scientific computing. In: International Parallel & Distributed Processing Symposium (IPDPS). pp. 1–8. IEEE (2009)
14. Kasichayanula, K., Terpstra, D., Luszczek, P., Tomov, S., Moore, S., Peterson, G.D.: Power aware computing on GPUs. In: Symposium on Application Accelerators in High-Performance Computing (SAAHPC). pp. 64–73. IEEE (2012)
15. Li, D., de Supinski, B.R., Schulz, M., Cameron, K., Nikolopoulos, D.S.: Hybrid MPI/OpenMP power-aware computing. In: International Parallel & Distributed Processing Symposium (IPDPS). pp. 1–12. IEEE (2010)
16. Luk, C.K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: IEEE/ACM International Symposium on Microarchitecture (Micro). pp. 45–55. ACM (2009)
17. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: International Symposium on Workload Characterization (IISWC). pp. 35–46. IEEE (2008)
18. Rofouei, M., Stathopoulos, T., Ryffel, S., Kaiser, W., Sarrafzadeh, M.: Energy-aware high performance computing with graphic processing units. In: Workshop on Power Aware Computing and Systems (HotPower). pp. 11–11. ACM (2008)
19. Sujeeth, A., Lee, H., Brown, K., Rompf, T., Wu, M., Atreya, A., Odersky, M., Olukotun, K.: OptiML: An implicitly parallel domain-specific language for machine learning. In: International Conference on Machine Learning (ICML). pp. 609–616. ACM (2011)
20. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Odersky, M., Olukotun, K.: Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Transactions on Embedded Computing Systems 13(4s), 1–25 (2014)
21. Sujeeth, A.K., Rompf, T., Brown, K.J., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., Olukotun, K.: Composition and reuse with compiled domain-specific languages. In: European Conference on Object-Oriented Programming (ECOOP). pp. 52–78. Springer (2013)
22. Takizawa, H., Sato, K.: SPRAT: Runtime processor selection for energy-aware computing. In: International Conference on Cluster Computing (Cluster). pp. 386–393. IEEE (2008)
23. Topcuouglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)
24. Trefethen, A.E., Thiyagalingam, J.: Energy-aware software: Challenges, opportunities and strategies. Elsevier Journal of Computational Science 4(6), 444–449 (2013)
25. Yang, C., Wang, F., Du, Y., Chen, J., Liu, J., Yi, H., Lu, K.: Adaptive optimization for petascale heterogeneous CPU/GPU computing. In: International Conference on Cluster Computing (Cluster). pp. 19–28. IEEE (2010)