CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Energy-based Tuning of Metaheuristics for Molecular Docking on Multi-GPUs

J. Pérez-Serrano[(1)], B. Imbernón[(2)], J.M. Cecilia[(2)], M. Ujaldón[(1)]

[(1)] *Computer Architecture Department, University of Malaga, Spain*
[(2)] *Computer Science Department, Catholic University San Antonio, Spain*

SUMMARY

Virtual Screening methods (VS) simulate molecular interactions in silico to look for the best chemical compound that interacts with a given molecular target. VS are becoming increasingly popular to accelerate the drug discovery process, and constitute hard optimization problems with a huge computational cost. To deal with these two challenges, we have created $METADOCK$, an application that (1) enables a wide range of metaheuristics through a parametrized schema, and (2) promotes the use of a multi-GPU environment within a heterogeneous cluster. Metaheuristics provide approximate solutions in a reasonable time frame, but given the stochastic nature of real-life procedures, the energy budget goes hand in hand with acceleration to validate the proposed solution.
This paper evaluates energy trade-offs and correlations with performance for a set of metaheuristics derived from $METADOCK$. We establish a solid inference from minimal power to maximal performance in GPUs, and from there to optimal energy consumption. This way, ideal heuristics can be chosen according not only to best accuracy and performance, but also to energy requirements. Our study starts with a preselection of parameterized metaheuristic functions, building blocks where we will find optimal patterns from power criteria while preserving parallelism through a GPU execution. We then establish a methodology to figure out the best instances of the parameterized kernels based on energy patterns obtained, which are analyzed from different viewpoints: Performance, average power and total energy consumed. We also compare the best workload distributions for optimal performance and power efficiency among Pascal and Maxwell GPUs on popular Titan models. Our experimental results demonstrate that the most power efficient GPU can be overloaded in order to reduce the total amount of energy required by as much as 20%, finding unique scenarios where Maxwell does it better in execution time, but with Pascal always ahead in performance per watt, reaching peaks of up to 40%.

Received . . .

KEY WORDS: Heterogeneous Computing, Low-Power, HPC, GPU, Metaheuristics, Molecular Docking

1. INTRODUCTION

The drug discovery and development process may take more than a decade to find a drug candidate to successfully advance from discovery to patient treatment [1]. This long process consists of different stages that a molecule must go through to become a drug. Virtual Screening methods (VS) have proven to be successful for enhancing the drug discovery process, saving time, money and resources [2, 3]. Such process analyzes large libraries of chemical compounds (namely *ligands*) to look for those molecules that are most likely to bind to a drug target, typically a protein receptor or enzyme [4].

*Correspondence to: Manuel Ujaldón

Ligand libraries may contain several million molecules [5], and with the analysis of larger databases to increase chances to find a good candidate, VS are always eager for high performance computing to keep the process within a reasonable time-frame and meet the expectations of the pharmaceutical industry. Following this trend, CPU-based clusters have been proposed by methods like Autodock [6], DOCK [7] or Glide [8] using Message Passing Interface (MPI) and/or multithreading techniques. On the other hand, BINDSURF [9] and more recently BUDE [10] propose the use of GPUs by dividing the whole protein surface into arbitrary independent regions (or spots). Finally, we propose $METADOCK$ [11], a multi GPU-based virtual screening methodology that is based on a parameterized metaheuristic schema. In short, $METADOCK$ is able to generate a wide range of metaheuristics, like genetic algorithms, scatter or tabu search, by setting up a group of parameters.

The combination of those two wavefronts has promoted a steady transition to heterogeneous computing systems [12], where nodes combine traditional multicore architectures (CPUs) with modern accelerators like GPUs. However, scalability is mainly limited by power constraints in large computational clusters [13], and so energy efficiency represents the cornerstone for future developments. Following the trend of many emerging applications, $METADOCK$ relies on metaheuristics to benefit from bioinspired processes that are inherently parallel by definition, but also stochastical, which may eventually lead to irregular memory footprints and computational patterns affecting power consumption in an unexpected manner. This paper provides a study to characterize $METADOCK$ on a performance per watt basis from a GPU perspective. We use CUDA to exploit massive parallelism at its best and to benefit from the minimal GFLOPS/w ratio in multi-GPU systems, now conquering the top 25 in the green500.org list [14]. We identify the energy patterns that are optimal while preserving their role as accelerator functions. Whenever one has the freedom to choose different ways to solve a problem with similar satisfactory results, these patterns will help users and programmers to choose the right version in different application areas. And within that version, the best values for the set of parameters involved.

Our work also compiles information about how energy evolves and is spent in typical metaheuristic parallel functions, to provide insights about margins of gain that can be exploited and the subsequent effort that is required to benefit from it. That way, a more sophisticated methodology can be elaborated from this departure point, and pros and cons can be easily derived from our current analysis and results.

The rest of the paper is organized as follows. Section 2 introduces some basic concepts. Section 3 defines metaheuristics, VS and METADOCK. Section 4 describes the GPU implementation. Section 5 introduces the experimental setup, and Section 6 the logistics for measuring energy. Section 7 provides and analyzes experimental results. Finally, Section 8 concludes summarizing the contributions of this work.

2. BACKGROUND

This section briefly introduces the main underlying concepts related to this paper, including some related work.

2.1. Virtual Screening methods (VS)

We originally described VS in [9, 15]. They are computational techniques used in assorted scientific areas, such as catalysts and energy materials [16], and mainly drug discovery [17], where experimental techniques can benefit from the predictions provided by simulation methods to help with the discovery of new drugs [18].

VS search for libraries of small molecules (called ligands) that can potentially bind to a drug target, usually a protein or enzyme (called receptor). Its computational cost comes from the exploration of large databases where it is typical to find millions of chemical compounds [19]. VS methods use a scoring function to quantify the binding affinity, i.e., the strength of association

DOI: 10.1002/cpe

between receptor and ligand conformation. Often, this function tries to be minimized on an NP-complete process involving a search algorithm.

VS may be divided according to accuracy and speed. Fast VS with atomic resolution may require few minutes per ligand [20], but those based on molecular dynamics can easily take weeks per ligand on a single CPU [21]. This has motivated GPUs to participate in the process, and even better, multi-GPU heterogeneous systems, which is our proposal in this work. Moreover, they look for (i.e., score) the optimal binding sites by providing a ranking of chemical compounds according to the estimated affinity or *scoring* [22].

In general, VS optimize *scoring functions*, which are mathematical models used to predict the strength of the non-covalent interaction between two molecules after docking [23]. The relevant non-bonded potentials used in VS calculations are the Coulomb, or electrostatic, and the Lennard-Jones potentials, to formulate the essential short- and long-range interactions between atoms of the protein-ligand system [24]. The calculation of non-bonded potentials takes the bulk of the workload, and may require up to 80% of the total execution time in molecular dynamics simulations [25].

Of particular interest to us within VS are protein-ligand docking techniques (see recent examples in [26] and [27]). From a computational viewpoint, we may distinguish CPU-based and GPU-based programs:

- Autodock [28] and Autodock VINA [28] are the most popular CPU-based approaches worldwide. Glide [29], DOCK [30], LeadFinder [31], SurFlex [32] and FMD [33] combine message passing interface (MPI) with multithreading. Major challenges of docking methods are the high computational cost and the predictive quality, so in order to relax the former and improve the latter, CPU-based simulations focus on the binding site once it is located, discarding remaining areas of the receptor.
- GPU-based approaches can afford not to be that restrictive. $BINDSURF$ [9], for example, divides the whole receptor surface into independent regions, and GPU parallelism explores them all simultaneously. This way, it is possible to find new binding sites on the surface of the protein by observing the results of the scoring function on each region. Another GPU-based software like BUDE [34] is another GPU-based approach that exploits the CPU-GPU heterogeneity using OpenCL for a better portability into hybrid architectures.

2.2. Metaheuristics

A wide range of optimization problems are NP-hard and, like VS, cannot afford to compute all possible solutions. In those scenarios, metaheuristics provide an abstraction layer to contribute with a good enough solution, which is found on a reduced search space focused just on promising areas. Over the past fifteen years, there has been a long list of successful cases tuned to particular problems in many areas of science [35, 36, 37, 38, 39]).

Metaheuristics introduce a high-level layer to provide a good enough solution for an optimization problem. They are very well-suited whenever there is limited computation capacity or inexact information [39]. Metaheuristics diminish the search space to focus only on the most interesting areas, at the expense of a low risk of eventually sacrificing the optimal solution. Of particular interest to us are two classes:

1. *Distributed metaheuristics*. They search within the entire solution space, working with populations that are combined to improve solutions progressively, for instance, Ant Colony, Particle Swarm Optimization, Genetic Algorithms and Scatter Search to name a few. The last two were chosen to be included within our experimental benchmark.
2. *Neighborhood metaheuristics*. They work with an element in the solution space and search for better elements in its neighborhood. Examples include Guided Local Search, Hill Climbing, Simulated Annealing, Tabu Search, Variable Neighborhood and the one we have chosen for our work: **GRASP** (Greedy Randomized Adaptive Search Procedures).

Diversity in metaheuristics [40] is often worth investigation. We can first define a subset of alternatives, and then follow a tuning process which turns fuzzy or even blind for the effects of

DOI: 10.1002/cpe

certain values in the experimental praxis. This work sheds some light on those scenarios with a guidance based on computational criteria: minimize execution time and energy spent among the spectrum of parameters with similar results. Still, the procedure may be different for each application area, and this way, the purpose of our experimental setup (see section 5) and results obtained (see section 7) primarily goes to quantify gains attained. Subsequently, a human expert or artificial intelligent system can take advantage of our findings to complete the selection process with clear benefits.

2.3. Tools for tuning performance and power

Nvidia started power-aware techniques including GPU Boost to control clock frequencies in Kepler, its third generation of CUDA hardware. For the user, the System Management Interface (nvidia-smi) is a command line utility based on top of the NVIDIA Management Library (NVML) to manage and monitor NVIDIA GPU devices. It has full support in Tesla and Quadro models, and limited scope in GeForces. For notebook models, PowerMizer controls performance and battery life. Similarly, for AMD products, PowerPlay reduces the dynamic power consumption by setting the GPU into low-, medium- and high-power states according to GPU utilization.

For all vendors and devices, Dynamic Voltage and Frequency Scaling (DVFS) adjusts power and speed settings on processors, controller chips and peripheral devices to optimize resource allotment for tasks and maximize power savings, battery life and longevity when those resources are not used. It is the most widely used mechanism for power management due to its ease of implementation and significant payoffs in terms of energy efficiency. DVFS allows the CPU, RAM and PCI-express devices to perform needed tasks with the minimum amount of required power.

2.4. Low power techniques

DVFS is usually controlled by the firmware layer in commodity PCs (BIOS), but it can be exploited at different layers. From higher to lower, we may distinguish:

- **Software:** For example, changing the frequency of the GPU core and video memory according to compute- and memory-bound CUDA kernels [41, 42]. Or combining DVFS with a concurrent kernel execution to improve the performance-per-watt behaviour compared their sequential execution [43].
- **Compiler:** Wu et al. [44] integrate a prototype of a DVFS mechanism into a dynamic compilation system which is fine-grained and code-aware However, both the code and the hardware have to be modified. Isci et al. [45] demonstrate a runtime phase predictor cooperating with the DVFS unit to analyze the history of branch predication in running applications, although keeping track of branches and making predictions also require extra hardware.
- **Operating System:** Lim et al. [46] present a message-passing interface runtime system that dynamically reduces the CPU performance during communication phases in MPI programs in order to save energy. Similarly, Choi et al. [47] proposed method aims at power saving during the CPU idle time on memory-bound codes.
- **Hardware:** Semeraro et al. [48] show and evaluate an online, dynamic DVFS algorithm for multiple clock domain processors that uses the attack/decay technique to reduce the frequency for energy savings at the expense of a little degradation in performance. David et al. [49] aimed at reducing the frequency of the memory system, and discovered that many workloads suffer minimal performance impact.

2.5. Hardware platforms for low-power

The greedy search for performance at any cost has led supercomputers to consume vast amounts of energy, not only for computing power but also for the cooling facilities required. Started in 2007, the Green500 List [50] emphasizes this issue by ranking the top 500 supercomputers in the world according to energy efficiency twice a year. Exploring the list, we immediately realize that Nvidia

DOI: 10.1002/cpe

GPUs represent the most solid alternative for sustainable computing nowadays, and so for our work we selected for our work the two most representative models for HPC manufactured by the company over the past couple of years: the Titan Maxwell (2014) and the Titan Pascal (2016), which are the flagships of the last two GPU generations (see Table II).

Vendors in modern times do not advertise raw GFLOPS anymore. Instead, GFLOPS/w has already become a widely used metric embraced by the research community. On 32-bit floating-point numbers, Nvidia GPUs have progressed notably in this field, with a steady transition towards low-power devices: The second CUDA generation, Fermi, scored 5-6 GFLOPS/w in 2010. The third generation, Kepler, aimed at 15-17 GFLOPS/w in 2012. Our Maxwell model, the fourth generation built two years later, exceeds 40 GFLOPS/w in power efficiency to benefit from a 8x improvement in barely four years.

3. METADOCK: APPLYING METAHEURISTICS ON VIRTUAL SCREENING METHODS

Algorithm 1 $METADOCK$ algorithm. The computation is based on a parameterized metaheuristic schema.

```
Initialize(S,ParamIni)
while no End condition(S) do
  Select(S,Ssel,ParamSel)
  Combine(Ssel,Scom,ParamCom)
  Improve(Scom,ParamImp)
  Include(Scom,S,ParamInc)
end while
```

$METADOCK$ divides the whole protein surface into arbitrary and independent regions (or spots). All these spots are independent of each other, and so this partitioning is very effective for data parallelism. Then, docking simulations for each ligand are performed at every spot simultaneously. $METADOCK$ randomly copies the same ligand at each of those spots, varying its orientation and position. Those copies are called individuals or conformations that may evolve in a different way depending on its translation, rotation and flexibility, and we look for an optimal *conformation* to minimize the free energy (given by the *scoring function*) in the entire system.

$METADOCK$ uses an optimization procedure where the scoring function, that models the non-bonded interactions between protein and ligand, is minimized throughout the execution. The scoring function is based on three terms: Electrostatic (ES), Van der Waals (VDW) and Hydrogen bond (HBOND). The calculation of the scoring function takes more than 95% of the overall execution time, and it is offloaded to our multi-GPU system to benefit from acceleration. The simulation starts trying to minimize the value of the scoring function by continuously making random or predefined perturbations of the initial population (*S*) at each spot. In particular, each candidate solution is a conformation (ligand-receptor) modified through a local search (like moving, translating and/or rotating with respect to a given region). Then, the new value of the scoring function for each candidate solution is obtained, being eventually accepted upon optimization criteria.

With that in mind, we introduce the optimization procedure used in $METADOCK$ before briefly describing the GPU implementation of the underlying scoring function. METADOCK uses a parameterized schema (see Algorithm 1) for the optimization procedure. This is based on the principle that all metaheuristics follow similar patterns, and particularly those based on populations share six basic functions (see Algorithm 1): Initialize, End Condition, Select, Combine, Improve and Include. Each of these functions works with various sets or populations (*S*, *Ssel* and *Scom*) and parameters (we refer the reader to [11] for insights). *S* represents the whole population of candidate solutions. In our case, a candidate solution (or individual) is a conformation, and several individuals are selected (*Ssel*) for their combination to generate a new set of elements, *Scom*. Candidate solutions can also be improved through a local search; i.e. moving, translating and/or rotating with respect to each spot ([51, 52]).

DOI: 10.1002/cpe

Table I. The seventeen metaheuristic parameters used within METADOCK.

Name	Description
INEIni	Number of initial ligand conformations.
PEIIni	Percentage of the best conformations that are improved in the Initialize function.
IIEIni	The intensification of the improvement in the Initialize function.
PBEIni	Percentage of best conformations to be included in the initial set for subsequent iterations.
PWEIni	Percentage of worst conformations to be included in the initial set for subsequent iterations.
PBESel	Percentage of the best conformations to be selected for combination.
PWESel	Percentage of the worst conformations to be selected for combination.
PBBCom	Percentage of best-best conformations to be combined.
PWWCom	Percentage of worst-worst conformations to be combined.
PBWCom	Percentage of best-worst conformations to be combined between them.
PMUCom	Percentage of best conformations of the combination to be muted.
IMUCom	The intensification of the mutation of elements generated by combination.
PEIImp	Percentage of best conformations of the combination to be improved.
IIEImp	The intensification of the improvement of elements generated by combination.
PBEInc	Percentage of best conformations to be included in the reference set.
MNIEnd	Maximum number of iterations.
NIREnd	Maximum number of steps without improvement.

Table I summarizes the set of seventeen parameters involved in METADOCK. We now briefly describe the basic functions it is composed of.

- **Initialize** returns an initial set of solutions. INEIni conformations are generated randomly for each of the m spots, and then a percentage (PEIIni) of the initial conformations of each spot is improved. The intensification of the improvement is indicated by IIEIni. Finally, (PBEIni+PWEIni)*INEIni conformations from each spot are selected for the execution of the subsequent functions. PBEIni and PWEIni represent the percentage of best and worst conformations according to the scoring function.
- **End condition** determines the stop criteria for METADOCK, which is either MNIEnd (the maximum number of iterations), or NIREnd (the maximum number of steps without improvement of the best solution among all the spots).
- **Select** chooses working conformations for subsequent phases. A percentage of the best (PBESel) and worst (PWESel) conformations relative to each spot is selected.
- **Combine** mixes conformations in pairs depending on their scoring. Three parameters represent the percentage of best-best, worst-worst and best-worst conformations to be combined: PBBCom, PWWCom and PBWCom, respectively. Combinations are performed among conformations at the same spot.
- **Mutation** maintains the diversity of conformations after the *Combine* stage. For those conformations affected by the mutation, its position in the space or its shape is modified randomly. Two parameters are involved in this function: PMUCom, to define the percentage of conformations that the mutation procedure receives as an input, and IMUCom, the intensification of the mutation.
- **Improve** performs a local search within the neighborhood of some of the conformations previously generated by *Combine*. Two metaheuristic parameters are considered for each spot: PEIImp, to define the percentage of conformations that the local search will apply to improve those conformations, and IIEImp, to establish the number of trials for the local search. Hence, METADOCK can generate hybrid metaheuristics with different degree of intensification, which can potentially be influenced at run-time by the energy budget.
- **Include** updates the reference set for the next iteration of the schema. Here, PBEInc establishes the percentage of best conformations associated to each spot to be included in its reference set. Remaining conformations to be included are randomly selected, and non-promising ones contribute to diversify the search, so avoiding stalling in local minima.

DOI: 10.1002/cpe

4. GPU IMPLEMENTATION FOR HETEROGENEOUS NODES

This section briefly describes the parallelization strategies on a heterogeneous cluster of CPUs and GPUs for the docking methodology we have introduced in Section 3. $METADOCK$ generates many OpenMP threads whenever GPUs are available at each node, information that can be obtained by executing the `cudaGetDeviceCount` function.

Algorithm 2 Interactions between protein and ligand.

```
for i=1 to N_CONFORMATION do
  for j=1 to N_ATOMS_RECEPTOR do
    for k=1 to N_ATOMS_LIGAND do
      Energy = ES + VDW + HBOND
      Scoring += Energy
    end for
  end for
  S_energy[i] = Scoring
  Scoring = 0
end for
```

Algorithm 2 outlines the scoring function calculation between a protein and several ligand conformations (i.e. the set S in algorithm 1), which is implemented by a single CUDA kernel. This kernel calculates all terms simultaneously by mapping each ligand conformation to a CUDA warp, and those conformations are grouped into blocks depending on the CUDA thread block granularity.

Algorithm 3 Computation of the scoring function on a Parameterized Metaheuristic for multicore+multiGPU.

```
1:  omp_set_num_threads(number_GPUs)
2:  #pragma omp parallel for
3:  for i=1 to number_GPUs do
4:    Select_device(Devices[i].id)
5:    Host_To_GPU(Scom,Stmp)
6:    Conformations=Devices[i].conformations
7:    threads=Devices[i].Threadsblock
8:    stride=Devices[i].stride
9:    Calculate_scoring<Conformations/threads,threads>
      (Stmp+Devices[i].stride)
10:   GPU_To_Host(Scom,Stmp)
11: end for
```

Algorithm 3 introduces the heterogeneous parallelization. OpenMP manages several CPU threads, where each thread is responsible for monitoring a GPU context (lines 2 and 3). Then, each GPU computes the scoring function for a set of ligand conformations. In an homogeneous distribution scheme, the same number of CUDA thread blocks are executed on each GPU. The GPU for the actual CPU-thread is selected (line 4) and the corresponding dataset is transferred from the CPU to the GPU (line 5). Each GPU then calculates the scoring function for a set of ligand conformations.

When the HPC cluster assembles GPUs coming from different generations, frequencies or memories, the number of ligand conformations assigned to each GPU should be done according to compute capabilities. In our previous work [11], uneven workload distributions were designed to maximize performance as a primary goal. We will use these as a departure point to illustrate that there is room for improvement when we include energy criteria to guide the partitioning process. In all cases, our methodology runs a short simulation with a reduced version of the problem to find the time and energy differences among GPUs, and distribute the workload according to the optimization criteria (maximize speed-up, minimize the GFLOPS/w ratio, or a combination of both).

DOI: 10.1002/cpe

Algorithm 4 Method to calculate scoring on GPU

```
pos = atom_position
individual = get_individual()
for i=1 to r do
  Energy = ES + VDW + HBOND
  Scoring += Energy
end for
synchronize_threads()
S_energy[individual] = Reduction_atoms_individual()
```

Table II. The GPUs used along our experimental study. We have a cluster composed of four GPUs: Two Maxwells and two Pascals, all of them coming from the high-end Titan model.

	GTX Titan	**Titan**
GPU generation	Maxwell	Pascal
Launching date (year)	2014	2016
Raw computational power:		
Number of cores	3072	3584
Cores frequency	1000 MHz	1417 MHz
Peak processing	6144 GFLOPS	10157 GFLOPS
CUDA Compute Capability	5.3	6.0
Memory:		
Type	GDDR5	HBM2
Size	12 GB.	12 GB.
Frequency	2x 3505 MHz	1400 MHz
Width	384 bits	4096 bits
Bandwidth	336.5 GB/s.	716.8 GB/s.
Cache:		
Shared memory / multipr.	96 KB.	64 KB.
L2 cache	2 MB.	2 MB.

Finally, Algorithm 4 briefly introduces the execution on the GPU side. Conformations are grouped into warps to optimize the execution, with threads dealing with the atom interactions between the conformation and the protein. A block-level synchronization is required to reduce all atom interactions calculated by each thread within the same conformation. The reduction is performed based on compute capabilities; i.e. we use the set of intrinsic $\_\_shfl$ instructions to accelerate the reduction at warp level. If the GPU does not provide such feature, the reduction is performed using shared-memory as an alternative.

Additional CUDA kernels in METADOCK include the support of ligand conformations as required by `Initialize` and `Improve` functions. Our implementation holds the information in device memory whenever possible to avoid costly communications through PCI-Express bus.

5. EXPERIMENTAL SETUP

5.1. Hardware Resources

We have conducted an experimental survey on a multi-GPU computer endowed with an Intel Xeon E5-2620 server and four PCI 3.0 slots to hold two Titan Pascal and two Titan Maxwell GPUs. Table II summarizes major features for these two GPUs. The CPU has eight cores running at 2100 MHz and 64 GB. of main memory running at 2400 MHz in a four-channel architecture. On the software side, Ubuntu 14.04.4 LTS 64 bits was installed as the operating system together with CUDA 8.0.

DOI: 10.1002/cpe

Table III. Values selected for the 17 parameters in METADOCK (in rows) for each of the 11 configurations of our benchmark (in columns).

	Algorithm: Genetic			Algorithm: Scatter-search			Algorithm: GRASP				
Parameter	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
INEIni	1024	2048	4096	512	1024	2048	1024	2048	4096	1024	1024
PEIIni	0	0	0	100	100	100	100	100	100	100	100
IIEIni	0	0	0	20	50	100	100	100	100	500	1000
PBEIni	100	100	100	4	4	4	100	100	100	100	100
PWEIni	0	0	0	4	4	4	0	0	0	0	0
PBESel	4	2	2	50	50	50	0	0	0	0	0
PWESel	0	0	0	50	50	50	0	0	0	0	0
PBBCom	100	100	100	100	100	100	0	0	0	0	0
PWWCom	0	0	0	100	100	100	0	0	0	0	0
PBWCom	0	0	0	100	100	100	0	0	0	0	0
PMUCom	20	30	50	0	0	0	0	0	0	0	0
IMUCom	5	10	50	0	0	0	0	0	0	0	0
PEIImp	0	0	0	100	100	100	0	0	0	0	0
IIEImp	0	0	0	20	100	200	0	0	0	0	0
PBEInc	0	0	0	100	100	100	0	0	0	0	0
MNIEnd	1	1	1	1	1	1	0	0	0	0	0
NIREnd	1	1	1	1	1	1	0	0	0	0	0

5.2. Benchmark

From the template shown in Algorithm 1, we have derived three assorted metaheuristics that were briefly introduced in Section 3. We label them now as follows:

- **Genetic:** For each spot in the receptor at the initialization stage, we will run this metaheuristic for three different populations (INEIni): 1024, 2048 and 4096 individuals. After initialization, only the best 500 individuals keep going with the computation ($PBEIni = 100$) to select the best candidates, which are combined and included for the next iteration ($PBBCom = 100$). No local search is included to improve the conformations.
- **Scatter-search:** This is another evolutionary method similar to a *Scatter Search* algorithm. We will start with a population of 512, 1024 and 2048 individuals. Many elements are improved after they have been generated, initially or by combination, through local search in the neighborhood of each element to obtain better solutions, and combinations between worst or best and worst elements are included. After the initialization phase, those elements selected are combined with each other, and a further improvement is applied to half of them.
- **GRASP:** This is a neighborhood-based metaheuristic, where local searches are applied to candidate solutions for a large initial set. In short, it can be seen as a GRASP method.

Our previous work [11] shows that these metaheuristics can be widely enriched with combinations of the above to derive multiple hybridations with different results and effects [53]. This time, we consider only those three as basic building blocks and, instead, we move the set of 17 parameters to study its influence on a performance per watt basis. This leads to a set of eleven configurations which are summarized in Table III.

Our goal is to create a representative and diverse testbed where metaheuristics may vary from compute-bound to memory-bound CUDA kernels, and from light to heavy workloads, in order to study its influence on energy spent. Table IV characterizes in this way each of the CUDA kernels involved.

5.3. Input Data Set

We use the Directory of Useful Decoys (DUD) [54] as chemical compounds to run our set of Virtual Screening methods. The DUD contains up to 40 sets protein-target with a ligand co-crystallized for each respective protein. This work uses the TK (Thymidine Kinase) target with 304 spots. 4691 and 31 atoms compose the target and its ligand, respectively.

DOI: 10.1002/cpe

Table IV. A preliminary characterization for the set of CUDA kernels belonging to our testbed, where we position each of its 11 configurations according to arithmetic intensity and workload.

Label for the Metaheuristic	CUDA kernel	- Arithmetic intensity + (memory-bound) (compute bound)	- Workload + (lighter) (heavier)
Genetic	Generation	C1 C2 C3	C1 C2 C3
	Combine	C1 C2 C3	C1 C2 C3
	Mutation	C1 C2 C3	C1 C2 C3
Scatter-search	Generation	C4 C5 C6	C4 C5 C6
	Enhance	C4 C5 C6	C4 C5 C6
	Combine	C4 C5 C6	C4 C5 C6
	Improve	C4 C5 C6	C4 C5 C6
GRASP	Generation	C7 C8 C9 C10 C11	C7 C8 C9 C10 C11
	Enhance	C7 C8 C9 C10 C11	C7 C8 C9 C10 C11

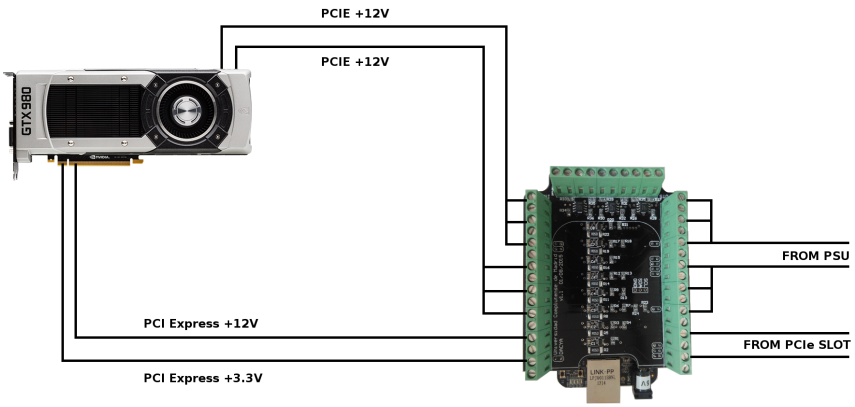

Figure 1. Wires, slots, cables and connectors for measuring energy on GPUs.

6. MONITORING ENERGY

6.1. Hardware Infrastructure

We have built a system to measure current, voltage and wattage based on a Beaglebone Black, an open-source hardware [55] combined with the Accelpower module [56], which has eight INA219 sensors [57]. Inspired by [58], wires taken into account are two power pins on the PCI-express slot (12 and 3.3 volts) plus six external 12 volt pins coming from the power supply unit (PSU) in the form of two supplementary 6-pin connectors (half of the pins used for grounding). See Figure 1 for details.

6.2. Software tool

Accelpower uses a modified version of `pmlib` library [59], a software package specifically created for monitoring energy. It consists of a server daemon that collects power data from devices and sends them to the clients, together with a client library for communication and synchronization with the server.

6.3. Methodology for Measuring Energy

The methodology for measuring energy begins with a start-up of the server daemon. Then, the source code of the application where the energy wants to be measured has to be modified to (1) declare `pmlib` variables, (2) clear and set the wires which are connected to the server, (3) create a counter and (4) start it. Once the code is over, we (5) stop the counter, (6) get the data, (7) save them to a .csv file, and (8) finalize the counter. See Figure 2 for a flow chart.

DOI: 10.1002/cpe

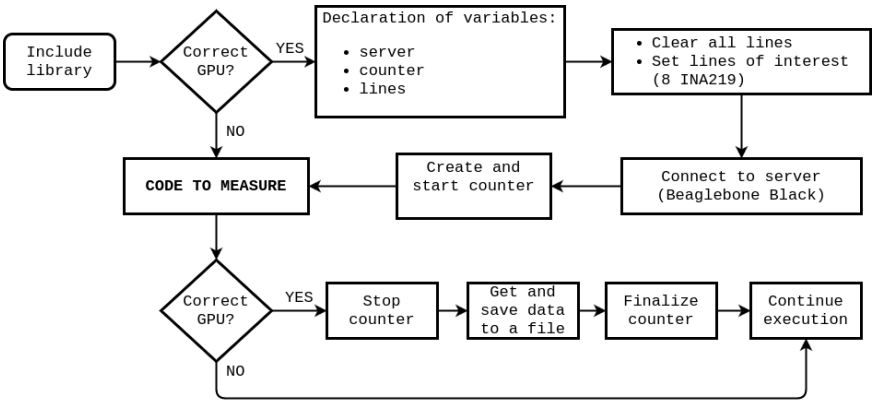

Figure 2. Flow diagram for measuring energy on a code excerpt when running on the GPU.

Table V. Guide for an easy location of our rich set of experimental results. How contents, sections, tables and figures are related to each other.

GPU	Algorithm (configs.)	Quantitative analysis (Section 7.1)		Qualitative analysis (Section 7.2)
		Section	Table	Table
Pascal	Genetic (C1-C3)	7.1.1	VII	VI
	Scatter (C4-C6)	7.1.2	VIII	VI (and Figure 3)
	GRASP (C7-C11)	7.1.3	IX	VI
Maxwell	Genetic (C1-C3)	7.1.1	X	VI
	Scatter (C4-C6)	7.1.2	XI	VI (and Figure 4)
	GRASP (C7-C11)	7.1.3	XII	VI

Headline	Discussion	Section	Table
Pascal vs. Maxwell	Hardware	7.3	XIII
Energy vs. Perform.	Policies	7.4	XIII
Parameters selection	Case study	7.5	XIV (and Figure 5)
	Optimal choices	7.6	XV

7. EXPERIMENTAL RESULTS

Our study covers a quantitative and qualitative analysis for each of the four GPUs, two Pascals and two Maxwells, being part of our quad-GPU system. Each GPU executes a subset or partition of the computation according to the workload distribution determined at a preliminary runtime analysis (as shown in the left side of Table XIII).

On the software side, we analyze results for each of the 11 configurations (C1-C11, involving 3 case studies for the algorithm of genetic type, 3 for scatter-search and 5 for GRASP) and stages within each algorithm.

We start with the quantitative approach followed by the qualitative analysis. Table V provides the reader an index for an easy location of results related to each section from now on.

7.1. Quantitative Analysis

7.1.1. Genetic algorithm. Table VII presents results for Pascal and Table X for Maxwell. Power, time and energy behave similarly on both GPUs. C1 and C2 are more energy efficient in Pascal, but C3 shows mixed results, with generation and combine stages performing better in Maxwell and warm-up and mutation stages favoring Pascal. Mutation holds the bulk of the computation, and therefore, is mainly responsible for Pascal's victory in the power and energy battle. However, Maxwell shows lower execution times for the workload distribution of 31% on every Pascal and 19% on every Maxwell (see Table XIII). The average power (shown in the same table, middle

Prepared using cpeauth.cls
DOI: 10.1002/cpe

columns) suggests that the two Pascals have to increase their workload by as much as 3.66%, 7.78% and 1.24% for C1, C2 and C3, respectively (see the right column) to optimize the performance per watt ratio. This causes a further delay in Pascal execution, and therefore, if we want to optimize the energy consumption globally, our multi-GPU system will run the algorithm with a more unbalanced workload distribution among GPUs.

7.1.2. Scatter-search algorithm. Tables VIII and XI contain results for Pascal and Maxwell, respectively. In line with previous numbers, Pascal consumes less power and Maxwell is faster. This way, to optimize the performance per watt ratio in our multi-GPU system, we have to increase the workload in Pascals by 2.51% for C4, 4.82% for C5 and 17.34% for C6. This would cause an additional unbalance for the execution time, so we have to establish a priority. Focusing on the longest stage by far, C6-Improve, we learn here that the GPU behaves like a car: The more you push the throttle, much more fuel it is required. Maxwell attains a 20.42% speed-up versus Pascal, but at the expense of a 35.29% more energy required. Therefore, many users may prefer to slowdown the execution to relax the energy demand.

7.1.3. GRASP algorithm. Table IX presents the results for Pascal and Table XII those for Maxwell. Differences in energy spent widens in this algorithm in favor of Pascal, which consumes around 40% less power on average. The algorithm is composed of three stages. In the first two, `Warm-up` and `Generation`, execution time is tiny and very similar in the two GPUs, so even though Pascal consumes much less, its effect is negligible. The last stage, `Enhance`, is responsible for the bulk of the computation, and for C9, C10 and C11, Pascal executes it quite slowly. If we wish to optimize the performance per watt ratio, power numbers suggest to increase the workload in Pascal between 13% and 20%, but fortunately, smaller increments are produced for the C10 and C11 cases (12.82% and 13.30%, respectively), which show higher differences in execution time from Pascal to Maxwell. This is also good news for C7 and C8, where we have to overload Pascal by more than 18%, but departure execution times are quite similar.

7.2. Qualitative Analysis

We have compiled a mosaic of charts to characterize the dynamic behaviour of the energy required on every moment for each configuration and stage of the scatter-search algorithm, which was found to be the most representative one in this respect. Figure 3 presents results for Pascal GPU and Figure 4 for Maxwell. The layout matches the deployment we used for tables in the previous section, that is, configurations occupying rows and stages within each algorithm are deployed in columns. The straight horizontal line drawn on each chart provides the average for wattage over time (that is, those values shown in previous tables).

Note that vertical scales (for power) and horizontal scales (for times) are not always the same on neighbour charts, so visually one has to establish a relative comparison. But maybe the most remarkable information provided here is the shape of the chart. We see plateaus, stairs, spikes and jagged lines, often sharing shapes in vertical alignments. That is, every stage has its own personality no matter which configuration we choose. Later in Section 7.6 we will identify certain parameters in Table I capable of moving power consumption quite considerably. This provides us attractive patterns in terms of energy efficiency which we would want to identify with useful simulations in engineer practice for an optimal combination of usefulness and green computing.

We now identify each pattern to define clusters of pairs configuration-stage sharing similar shapes.

- **Plateau:** The GPU reaches a stable point where energy is consumed on a constant rate, and after a while, goes back to a valley. It is defined as a binary state of minimal thermal stress, the most positive scenario for reliable and everlasting silicon chips. We found this pattern to be the most popular in our analysis. 29 out of the 84 kernels compiled in Table VI behave this way.
- **Stairs:** Power increases step by step drawing periods of stability on the way, and after holding on top, goes back to its departure point. This pattern usually ends up with higher power on

DOI: 10.1002/cpe

average than plateau, as little by little it enters more demanding scenarios. We have found kernels within this category that start playing with scalars on registers (first step), then activate memory to access data volume (second step), and finally compute intensively with gathered data to define the summit scenario. But other instances activate even more steps. Overall, 22 kernels were found within this category in Table VI.

- **Jagged:** The GPU switches continuously between two states of different power demands, that way suffering from maximum thermal stress. We have found this pattern to be behind most of the kernels where average power exceeded 100 W, and the distance between the two states usually places around 50 W. 16 kernels follow this pattern in Table VI.
- **Spikes:** Occasionally, a stable kernel increases power consumption dramatically, but does not hold the situation in time, fading away soon to recover stability. Spikes are not harmful as long as they do not exceed the Thermal Design Power (TDP) of the GPU, which is usually established around the 300 W threshold. We only saw four kernels experiencing spikes: Configurations 1 and 2 for Genetic-Mutation showed spikes of 195 W for Pascal and 180 W for Maxwell.

Those four patterns were listed from more to less frequent within our experiments. If we want to rate them according to what is a desirable pattern for reliability (anti-aging) and savings (low power), plateau would be the best, followed by stairs, spikes and finally jagged the worst. That sets up our preferences to model the pattern exhibited by a GPU kernel via a parameters selection from the universe previously shown in Table I.

7.3. Pascal versus Maxwell

We are now curious about differences in energy spent among GPU generations. When initially announced by Nvidia, Maxwell was claimed the most power efficient GPU ever built [60]. Compared to its predecessor Kepler, multiprocessors were reduced to 128 cores and its layout was reorganized into quadrants to shorten wires length. Communications and power lines were identified primary factors in energy consumption, so it was no surprise to find Maxwell ahead a 2x factor in performance per watt.

Enhancements in Pascal versus Maxwell were driven by performance and energy, but with certain tradeoffs. Focusing on Titan models to be fair, Table II summarizes features for the two GPUs used in our study. The Maxwell model contains 3072 cores at 1000 MHz clock rate, whereas the Pascal counterpart has 3584 cores running at 1417 MHz. The number of transistors on a chip and its frequency affect power in a linear way, which leads us to estimate Pascal around 65% higher on energy demand. But there was also good news for Pascal on a performance-per-watt basis: Multiprocessors were reduced to 64 cores and, overall, manufacturing process evolved from planar 28 nm. transistors to 16 nm. fin-FET ones [61]. Therefore, it is complex to assess pros and cons to determine a winner of the energy battle, and even more challenging to put differences in numbers.

Our experiments may shed some light on this issue driven by praxis. Summary numbers that we have compiled in the two central columns of Table XIII indicate that Pascal is consistently more power efficient than Maxwell: Starting with 2.48% for C3, ending with 40.00% for C9, and 21.90% on average.

For a qualitative analysis, Table VI performs a chart-by-chart comparison of power functions over time for all kernels running on Pascal (compiled in Figure 3 for the most illustrative case of the scatter algorithm) and Maxwell (shown in Figure 4). In general, energy consumption evolves similarly in all chips: Each GPU executes 12 kernels of genetic type, 15 kernels of scatter-search type and 15 kernels of GRASP type, and we only found minor differences in 10 cases (which have been framed on corresponding cells belonging to the Maxwell GPU, see lower half of the Table). We also found that occasionally Maxwell consumes less power than Pascal. An example is given in configuration 4 during Enhance and Combine stages, because the number of individuals is small. Whenever we increase that number (see configurations 5 and 6, each doubling the amount), computation is heavier and Pascal recovers the leadership.

DOI: 10.1002/cpe

Table VI. Characterizing each combination of GPU (upper:Pascal, lower:Maxwell), configuration (in rows) and algorithm/stage (in columns) according to its dynamic energy consumption. We classify 84 combinations into clusters according to common shapes, finding 29 plateaus, 22 stairs, 16 jagged and 4 spikes. There are also 13 hybrid shapes in a miscellaneous class. 10 charts are identified as different in Pascal and Maxwell (see those cells framed in the Maxwell tables near the bottom).

Genetic on Pascal	Warm-up	Generation	Combine	Mutation
C1	Stairs	Stairs	Plateau	Spikes
C2	Plateau	Plateau	Plateau	Spikes
C3	Plateau	Plateau	Plateau	Plateau-Jagged

Scatter on Pascal	Warm-up	Generation	Enhance	Combine	Improve
C4	Stairs	Stairs	Jagged	Plateau	Jagged
C5	Stairs	Stairs-Plateau	Jagged	Plateau	Jagged
C6	Stairs	Plateau	Jagged	Spikes-Jagged	Jagged

GRASP on Pascal	Warm-up	Generation	Enhance
C7	Stairs	Stairs	Jagged
C8	Stairs	Plateau-Spikes	Jagged
C9	Stairs	Plateau-Spikes	Jagged
C10	Stairs	Plateau	Jagged-Plateau
C11	Stairs	Stairs	Jagged-Plateau

Genetic on Maxwell	Warm-up	Generation	Combine	Mutation
C1	Stairs-Plateau	Plateau	Plateau	Spikes
C2	Plateau	Plateau	Plateau	Spikes
C3	Plateau	Plateau	Plateau	Plateau-Jagged

Scatter on Maxwell	Warm-up	Generation	Enhance	Combine	Improve
C4	Stairs	Plateau	Jagged	Plateau	Jagged
C5	Stairs	Plateau	Jagged-Plateaus	Plateau	Jagged
C6	Stairs	Plateau	Jagged-Plateaus	Plateau	Jagged

GRASP on Maxwell	Warm-up	Generation	Enhance
C7	Stairs	Plateau	Jagged
C8	Stairs	Plateau	Jagged
C9	Stairs	Plateau	Jagged
C10	Stairs	Plateau	Jagged-Plateau
C11	Stairs	Stairs	Jagged-Plateau

7.4. Energy versus performance

In a previous work [11], we describe a methodology for establishing the workload distribution on a multi-GPU environment, which is shown in Table XIII, second column. This would supposedly balance execution time in all GPUs, as our former policy was to assign more work to more powerful GPUs. But we have just seen that kernels running on Pascal consume less watts on average. Therefore, if we trade time by energy and change the policy to reach the most power efficient execution, Pascal GPUs have to be overloaded. The last column of the table reveals to what extent,

DOI: 10.1002/cpe

and that savings can reduce the total amount of energy required to run each configuration by as much as 20%. That way, our methodology can easily be modified to benefit from those findings.

In general, the GPU evolution has demonstrated that performance does not correlate ideally with energy efficiency, but this paper demonstrates that the Pascal generation does it quite well in this respect, encouraging you to press the throttle because you will not end up paying more on fuel.

7.5. Parameters selection: Case study

We dedicate our final analysis to illustrate how metaheuristic parameters may influence energy spent by each GPU kernel. Given the wide range of parameters and kernels, we preferred to focus on a case study, shown in Table XIV and Figure 5. It corresponds to Pascal GPU, C5 configuration and Improve kernel, under three values for IIEImp: 20, 100 and 200. In terms of power, and based on a quantitative analysis, IIEImp=100 is the more appealing value, since average power reaches the minimum of the three: 100.94 W (see second column in Table XIV).

Charts in Figure 5 explain why this is the best choice: It is the only case where wattage relaxes from a jagged shape down to a 20W plateau after 1300 seconds. All remaining charts keep consistently over the 100W mark. The conclusion is: Should all metaheuristics produce similar results, the tie must be solved in favor of IIEImp=100.

7.6. Parameters selection: Optimal choices

Following a similar procedure, we have summarized in Table XV the best choice for each parameter under three different criteria: Performance, energy spent and performance per watt. According to performance, the variant with minimum execution time is acclaimed as winner (c1, c4 and c7), and that often leads to minimize energy too. Workload usually increases with the size of the population in an attempt to improve accuracy, and parameters like INEIni influence simulations that way. On the other hand, performance per watt is optimal when we minimize the global average power, which is shown in the upper part of the last column on every table analyzed.

Our final analysis goes to identify our best choices with memory-bound and compute-bound kernels according to Table IV. Winner kernels in performance and energy are always memory-bound and lightweight. However, when we move to performance per watt criteria, the situation changes: For the genetic and GRASP cases, optimal choices remain the same, but for scatter-search, the winners are compute-bound and heavyweight. This reflects tradeoffs between high-performance and low-power as we already suspected. However, our results demonstrate that energy penalties are worth it compared to speed-up benefits, so we conclude that performance may be established on a higher priority, particularly for the HPC community in this time period of the GPU evolution.

With our results having been obtained from the latest model of the Pascal generation, which will be minimally upgraded with the arrival of Volta chips (unless you execute deep learning algorithms, according to what Nvidia has already announced), we foresee this architecture to be the flagship GPU at least for the next couple of years.

8. CONCLUSIONS

This paper evaluates energy tradeoffs and correlations with performance for a sample metaheuristic application running on a heterogeneous cluster of CPUs and GPUs.

Metaheuristics allow us to create a rich testbed composed of 9 kernels and 11 parameters, carefully chosen to represent all possibilities within a CUDA code: from low to high arithmetic intensity and from light to heavy workload. Every programmer may have a similar instance within that benchmark to represent his code, and follow implications to extract solid lessons from our subsequent analysis. We then identify those features that better characterize a good behaviour in performance and power consumption, establish margins of benefit and provide insights to develop a methodology for kernels selection according to energy efficiency. Those kernels may then assist in many application areas to provide solutions tailored to green computing.

DOI: 10.1002/cpe

By implementing METADOCK algorithms on GPUs using CUDA, we have established a solid inference from minimal power to maximal performance, and from there to optimal energy consumption. Experimental results demonstrate the benefits of our analysis: Whenever we identify the most energy-efficient GPU model, Pascal in our case, we can decide to overload it in order to reduce the total amount of energy required by as much as 20%. Or the other way around: Sacrifice that 20% of energy to optimize parallelism and minimize the execution time.

Our final contribution is to compare Pascal and Maxwell GPUs on popular Titan models from performance and energy criteria. We find that, in certain scenarios, Maxwell can overtake Pascal in execution time, but as far as wattage is concerned, Pascal is always ahead, reaching peaks of up to 40%.

From this work, we foresee GPUs increase their role as high performance and low power devices in future GPU generations, particularly after the introduction in late 2016 of the 3D memory within Pascal models. This computational power may be combined with the energy-tuning techniques applied here for an optimal selection of parameters in diverse application areas, just by following our illustrative example on metaheuristics via Virtual Screening methods.

ACKNOWLEDGEMENTS

This work was supported by the Ministry of Education of Spain under Project TIN2013-42253-P, TIN2016-78799-P (AEI/FEDER, UE) and by the Junta de Andalucia under Project of Excellence P12-TIC-1741. We thank Nvidia for hardware donations within GPU Education Center 2011-2016 and GPU Research Center 2012-2016 awards at the University of Malaga (Spain). We also thank Francisco D. Igual and Luis Piñuel from the Computer Architecture and Automated Department at the Complutense University of Madrid (Spain) for providing us Accelpower modules to measure power during our experimental survey. Our measuring system is based on a tool being continuously upgraded as reported in http://accelpowercape.dacya.ucm.es.

REFERENCES

1. Hajduk PJ, Greer J. A decade of fragment-based drug design: strategic advances and lessons learned. *Nature Reviews Drug discovery* 2007; **6**(3):211–219.
2. Jorgensen WL. The Many Roles of Computation in Drug Discovery. *Science* 2004; **303**:1813–1818, doi: 10.1126/science.1096361.
3. Yuan S, Chan JFW, den Haan H, Chik KKH, Zhang AJ, Chan CCS, Poon VKM, Yip CCY, Mak WWN, Zhu Z, *et al.*. Structure-based discovery of clinically approved drugs as zika virus ns2b-ns3 protease inhibitors that potently inhibit zika virus infection in vitro and in vivo. *Antiviral Research* 2017; .
4. Rollinger JM, Stuppner H, Langer T. Virtual screening for the discovery of bioactive natural products. *Natural Compounds as Drugs Volume I*. Springer, 2008; 211–249.
5. Irwin JJ, Shoichet BK. ZINC–a free database of commercially available compounds for virtual screening. *Journal of Chemical Information and Modeling* 2005; **45**(1):177–182.
6. Morris GM, Goodsell DS, Halliday RS, Huey R, Hart WE, Belew RK, Olson AJ. Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry* 1998; **19**(14):1639–1662.
7. Ewing TJA, Makino S, Skillman AG, Kuntz ID. DOCK 4.0: Search strategies for automated molecular docking of flexible molecule databases. *Journal of Computer-Aided Molecular Design* 2001; **15**(5):411–428.
8. Friesner RA, et al. Glide: A New Approach For Rapid, Accurate Docking and Scoring: Method and Assessment of Docking Accuracy. *Journal of Medicinal Chemistry* 2004; **47**(7):1739–1749.
9. Sánchez-Linares I, Pérez-Sánchez H, Cecilia JM, García JM. High-throughput parallel blind virtual screening using BINDSURF. *BMC Bioinformatics* 2012; **13**(Suppl 14):S13.
10. McIntosh-Smith S, Price J, Sessions RB, Ibarra AA. High performance in silico virtual drug screening on many-core processors. *International Journal of High Performance Computing Applications* 2014; :1094342014528 252.
11. Imbernón B, Cecilia J, Pérez-Sánchez H, Giménez D. METADOCK: A Parallel Metaheuristic Schema for Virtual Screening Methods. *The Intl. Journal of High Performance Computing Applications* March 2017; .
12. Top500. Top500 supercomputer site. `http://www.top500.org/` 2016. (accessed, October, 4th, 2016).
13. Carretero J, Garcia-Blas J, Singh DE, Isaila F, Fahringer T, Prodan R, Bosilca G, Lastovetsky A, Symeonidou C, Perez-Sanchez H, *et al.*. Optimizations to enhance sustainability of mpi applications. *Proceedings of the 21st European MPI Users' Group Meeting*, ACM, 2014; 145.
14. The Green 500 Supercomputers List. http://www.top500.org/green500.
15. Guerrero GD, Cebrián JM, Pérez-Sánchez H, García JM, Ujaldón M, Cecilia JM. Toward energy efficiency in heterogeneous processors: findings on virtual screening methods. *Concurrency and Computation: Practice and Experience* 2014; **26**(10):1832–1846.
16. Franco AA. Multiscale modelling and numerical simulation of rechargeable lithium ion batteries: concepts, methods and challenges. *RSC Advances* 2013; .

DOI: 10.1002/cpe

17. Kitchen DB, Decornez H, Furr JR, Bajorath J. Docking and scoring in virtual screening for drug discovery: methods and applications. *Nature Reviews Drug Discovery* 2004; **3**(11):935–949.
18. Jorgensen WL. The Many Roles of Computation in Drug Discovery. *Science* 2004; **303**:1813–1818.
19. Irwin JJ, Shoichet BK. ZINC–a free database of commercially available compounds for virtual screening. *Journal of Chemical Information and Modeling* 2005; **45**(1):177–182.
20. Zhou Z, Felts AK, Friesner RA, Levy RM. Comparative performance of several flexible docking programs and scoring functions: enrichment studies for a diverse set of pharmaceutically relevant targets. *Journal of Chemical Information and Modeling* 2007; **47**(4):1599–1608.
21. Wang J, Deng Y, Roux B. Absolute Binding Free Energy Calculations Using Molecular Dynamics Simulations with Restraining Potentials. *Biophys J* Oct 2006; **91**(8):2798–2814.
22. Schneider G. Virtual screening and fast automated docking methods. *Drug Discovery Today* Jan 2002; **7**:64–70.
23. Jain AN. Scoring functions for protein-ligand docking. *Current Protein and Peptide Science* 2006; **7**(5):407–420.
24. Imbernón B, Cecilia JM, Giménez D. Enhancing metaheuristic-based virtual screening methods on massively parallel and heterogeneous systems. *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, ACM, 2016; 50–58.
25. Kuntz SK, Murphy RC, Niemier MT, Izaguirre JA, Kogge PM. Petaflop Computing for Protein Folding. *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, 2001; 12–14.
26. Yuriev E, Ramsland P. Challenges and Advances in Computational Docking: 2009 in Review. *Journal of Molecular Recognition* 2011; **24**(2):149–164.
27. Huang S, Zou X. Advances and Challenges in Protein-Ligand Docking. *International Journal of Molecular Sciences* 2010; **11**(8):3016–3034.
28. Morris GM, Goodsell DS, Halliday RS, Huey R, Hart WE, Belew RK, Olson AJ. Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry* 1998; **19**(14):1639–1662.
29. Friesner RA, Banks JL, Murphy RB, Halgren TA, Klicic JJ, Mainz DT, Repasky MP, Knoll EH, Shelley M, Perry JK, *et al.*. Glide: A New Approach For Rapid, Accurate Docking and Scoring: Method and Assessment of Docking Accuracy. *Journal of Medicinal Chemistry* 2004; **47**(7):1739–1749.
30. Ewing TJA, Makino S, Skillman AG, Kuntz ID. DOCK 4.0: Search strategies for automated molecular docking of flexible molecule databases. *Journal of Computer-Aided Molecular Design* 2001; **15**(5):411–428.
31. Stroganov OV, Novikov FN, Stroylov VS, Kulkov V, Chilov GG. Lead finder: an approach to improve accuracy of protein- ligand docking, binding energy estimation, and virtual screening. *Journal of Chemical Information and Modeling* 2008; **48**(12):2371–2385.
32. Jain AN. Surflex: fully automatic flexible molecular docking using a molecular similarity-based search engine. *Journal of Medicinal Chemistry* 2003; **46**(4):499–511.
33. Dolezal R, Ramalho TC, França TC, Kuca K. Parallel flexible molecular docking in computational chemistry on high performance computing clusters. *Computational Collective Intelligence*. Springer, 2015; 418–427.
34. McIntosh-Smith S, Price J, Sessions RB, Ibarra AA. High performance in silico virtual drug screening on many-core processors. *International Journal of High Performance Computing Applications* 2014; :1094342014528 252.
35. Dréo J, Pétrowski A, Siarry P, Taillard E. *Metaheuristics for Hard Optimization*. Springer, 2005.
36. Glover F, Kochenberger GA. *Handbook of Metaheuristics*. Kluwer, 2003.
37. Hromkovič J. *Algorithmics for Hard Problems*. Second edn., Springer, 2003.
38. Michalewicz Z, Fogel DB. *How to Solve It: Modern Heuristics*. Springer, 2002.
39. Bianchi L, Dorigo M, Gambardella L, Gutjahr W. A Survey on Metaheuristics for Stochastic Combinatorial Optimization. *Natural Computing* 2009; **8**(2):239–287.
40. Blum C, Roli A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)* 2003; **35**(3):268–308.
41. Jiao Y, Lin H, Balaji P, Feng W. Power and Performance Characterization of Computational Kernels on the GPU. *Intl. Conference on Green Computing and Communications*, 2010.
42. Lee J, Sathisha V, Schulte M, Compton K, Kim N. Improving Throughput of Power-Constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling. *International Conference on Parallel Architectures and Compilation Techniques*, 2011.
43. Jiao Q, Lu M, Huynh H, Mitra T. Improving GPGPU Energy-Efficiency Through Concurrent Kernel Execution and DVFS. *IEEE/ACM Intl. Symposium on Code Generation and Optimization*, 2015.
44. Wu Q, Reddi V, Wu Y, Lee J, Connors D, Brooks D, Martonosi M, Clark D. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. *Proceedings 38th Annual IEEE/ACM Intl. Symposium on Microarchitecture*, 2005; 271–282.
45. Isci C, Contreras G, Martonosi M. Runtime Phase Monitoring and Prediction on Real Systems with Applications to Dynamic Power Management. *Proceedings 39th Annual IEEE/ACM Intl. Symposium on Microarchitecture*, 2006; 359–370.
46. Lim M, Freeh V, Lowenthal D. Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs. *Proceedings ACM/IEEE Intl. Conference for High Performance Computing, Networking, Storage and Analysis*, 2006; 14–14.
47. Choi K, Soma R, Pedram M. Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Tradeoff Based on the Ratio of Off-chip Access to On-chip Computation Times. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2005; **24**(1):18–28.
48. Semeraro G, Albonesi D, Dropsho S, Magklis G, Dwarkadas S, Scott M. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture. *Proceedings 35th IEEE/ACM Intl. Symposium on Microarchitecture*, 2002; 356–367.
49. Fallins HDC, Gorbatov E, Hanebutte U, Mutlu O. Memory Power Management via Dynamic Voltage/Frequency Scaling. *Proceedings ACM Intl. Conference on Autonomic Computing*, 2011; 31–40.
50. URL `www.green500.org`.

DOI: 10.1002/cpe

Table VII. Power, execution time and energy consumption on every **Pascal GPU** for each configuration (in rows) and stage (in columns) of the **genetic algorithm**.

Configuration	Warm-up	Generation	Combine	Mutation	All
Average power (watts per GPU)					
C1	68.54 W	87.12 W	111.96 W	86.14 W	88.13 W
C2	96.24 W	121.17 W	110.54 W	72.70 W	94.87 W
C3	99.25 W	137.05 W	153.82 W	122.60 W	130.42 W
Execution time (seconds)					
C1	4.09 s	4.68 s	4.41 s	9.91 s	23.10 s
C2	4.39 s	4.86 s	5.22 s	9.72 s	24.20 s
C3	4.42 s	7.32 s	8.98 s	15.43 s	36.15 s
Energy consumption (kilojoules per GPU)					
C1	0.28 kJ	0.40 kJ	0.49 kJ	0.85 kJ	2.03 kJ
C2	0.42 kJ	0.58 kJ	0.57 kJ	0.70 kJ	2.29 kJ
C3	0.43 kJ	1.00 kJ	1.38 kJ	1.89 kJ	4.71 kJ

Table VIII. Power, execution times and energy consumption on every **Pascal GPU** for each configuration and stage of the **scatter-search algorithm**.

Configuration	Warm-up	Generation	Enhance	Combine	Improve	All
Average power (watts per GPU)						
C4	72.46 W	77.67 W	174.26 W	128.21 W	154.46 W	151.08 W
C5	71.69 W	89.45 W	185.11 W	158.64 W	127.40 W	130.78 W
C6	70.00 W	110.48 W	147.00 W	124.51 W	119.98 W	120.81 W
Execution time (seconds)						
C4	4.03 s	4.33 s	19.46 s	5.32 s	90.29 s	123.46 s
C5	4.08 s	4.41 s	91.74 s	12.35 s	1453.87 s	1566.46 s
C6	4.02 s	5.06 s	396.32 s	42.17 s	12358.52 s	12806.11 s
Energy consumption (kilojoules per GPU)						
C4	0.29 kJ	0.33 kJ	3.39 kJ	0.68 kJ	13.94 kJ	18.65 kJ
C5	0.29 kJ	0.39 kJ	16.98 kJ	1.95 kJ	185.23 kJ	204.86 kJ
C6	0.28 kJ	0.55 kJ	58.26 kJ	5.25 kJ	1482.79 kJ	1547.14 kJ

51. Raidl GR. A unified view on hybrid metaheuristics. *Hybrid Metaheuristics*. Springer, 2006; 1–12.
52. Vaessens RJ, Aarts EH, Lenstra JK. A local search template. *Computers & Operations Research* 1998; **25**(11):969–979.
53. Almeida F, Giménez D, López-Espín JJ, Pérez-Pérez M. Parameterised schemes of metaheuristics: basic ideas and applications with Genetic algorithms, Scatter Search and GRASP. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans* 2013; **43**(3):570–586.
54. DUD. Directory of Useful Decoys. `http://dud.docking.org/` 2006. (accessed, October, 4th, 2016).
55. BeagleBone. Beaglebone black. http://beagleboard.org/BLACK.
56. González-Rincón J. Sistema basado en open source hardware para la monitorización del consumo de un computador. *Master Thesis Project. Universidad Complutense de Madrid* 2015; .
57. Ada L. Adafruit INA219 Current Sensor Breakout. `https://learn.adafruit.com/adafruit-ina219--current-sensor-breakout`.
58. Igual F, Jara L, Gómez J, Piñuel L, Prieto M. A Power Measurement Environment for PCIe Accelerators. *Computer Science - Research and Development* May 2015; **30**(2):115–124.
59. Alonso P, Badía R, Labarta J, Barreda M, Dolz M, Mayo R, Quintana-Ortí E, Reyes R. Tools for power-energy modelling and analysis of parallel scientific applications. *Proceedings 41st Intl. Conference on Parallel Processing (ICPP'12)*, IEEE Computer Society, 2012; 420–429.
60. Nvidia. NVIDIA GeForce GTX 980: Featuring Maxwell, The Most Advanced GPU Ever Made. *Whitepaper*, Corporation N (ed.), 2014.
61. NVIDIA Tesla P100. The Most Advanced Datacenter Accelerator Ever Built. *Whitepaper*, Corporation N (ed.), 2016.

DOI: 10.1002/cpe

Table IX. Power, execution times and energy consumption on every **Pascal GPU** for each configuration and stage of the **GRASP algorithm**.

Configuration	Warm-up	Generation	Enhance	All
Average power (watts per GPU)				
C7	67.40 W	73.67 W	118.84 W	115.47 W
C8	72.83 W	96.85 W	109.82 W	108.97 W
C9	68.63 W	87.41 W	108.98 W	108.34 W
C10	77.52 W	107.30 W	127.92 W	127.36 W
C11	74.91 W	98.08 W	125.43 W	125.40 W
Execution time (seconds)				
C7	4.05 s	4.55 s	114.24 s	122.84 s
C8	4.10 s	4.95 s	243.44 s	252.50 s
C9	4.04 s	7.44 s	490.41 s	501.91 s
C10	4.10 s	4.48 s	528.47 s	537.06 s
C11	4.65 s	4.51 s	11040.64 s	11049.80 s
Energy consumption (kilojoules per GPU)				
C7	0.27 kJ	0.33 kJ	13.57 kJ	14.18 kJ
C8	0.29 kJ	0.48 kJ	26.73 kJ	27.51 kJ
C9	0.27 kJ	0.65 kJ	53.44 kJ	54.37 kJ
C10	0.31 kJ	0.48 kJ	67.60 kJ	68.40 kJ
C11	0.34 kJ	0.44 kJ	1384.92 kJ	1385.71 kJ

Table X. Power, execution time and energy consumption on every **Maxwell GPU** for each configuration and stage of the **genetic algorithm**.

Configuration	Warm-up	Generation	Combine	Mutation	All
Average power (watts per GPU)					
C1	79.02 W	106.74 W	114.25 W	86.71 W	94.59 W
C2	110.00 W	120.62 W	116.49 W	100.55 W	109.65 W
C3	114.86 W	127.15 W	145.74 W	134.61 W	133.67 W
Execution time (seconds)					
C1	4.53 s	4.11 s	4.57 s	8.77 s	22.00 s
C2	4.41 s	5.07 s	4.74 s	9.79 s	24.03 s
C3	4.26 s	7.05 s	9.22 s	15.80 s	36.35 s
Energy consumption (kilojoules per GPU)					
C1	0.35 kJ	0.43 kJ	0.52 kJ	0.76 kJ	2.08 kJ
C2	0.48 kJ	0.61 kJ	0.55 kJ	0.98 kJ	2.63 kJ
C3	0.48 kJ	0.89 kJ	1.34 kJ	2.12 kJ	4.85 kJ

Table XI. Power, execution times and energy consumption on every **Maxwell GPU** for each configuration and stage of the **scatter-search algorithm**.

Configuration	Warm-up	Generation	Enhance	Combine	Improve	All
Average power (watts per GPU)						
C4	89.41 W	99.45 W	172.00 W	124.01 W	164.48 W	158.69 W
C5	100.46 W	115.64 W	149.20 W	153.35 W	143.12 W	143.39 W
C6	91.72 W	120.36 W	157.01 W	161.61 W	163.73 W	163.45 W
Execution time (seconds)						
C4	4.68 s	4.12 s	18.44 s	5.38 s	87.71 s	120.34 s
C5	4.69 s	4.52 s	103.23 s	11.42 s	1440.93 s	1564.81 s
C6	4.26 s	4.86 s	353.23 s	45.81 s	10262.10 s	10670.29 s
Energy consumption (kilojoules per GPU)						
C4	0.41 kJ	0.40 kJ	3.17 kJ	0.66 kJ	14.42 kJ	19.09 kJ
C5	0.47 kJ	0.52 kJ	15.40 kJ	1.75 kJ	206.23 kJ	224.38 kJ
C6	0.39 kJ	0.58 kJ	55.46 kJ	7.40 kJ	1680.28 kJ	1744.13 kJ

DOI: 10.1002/cpe

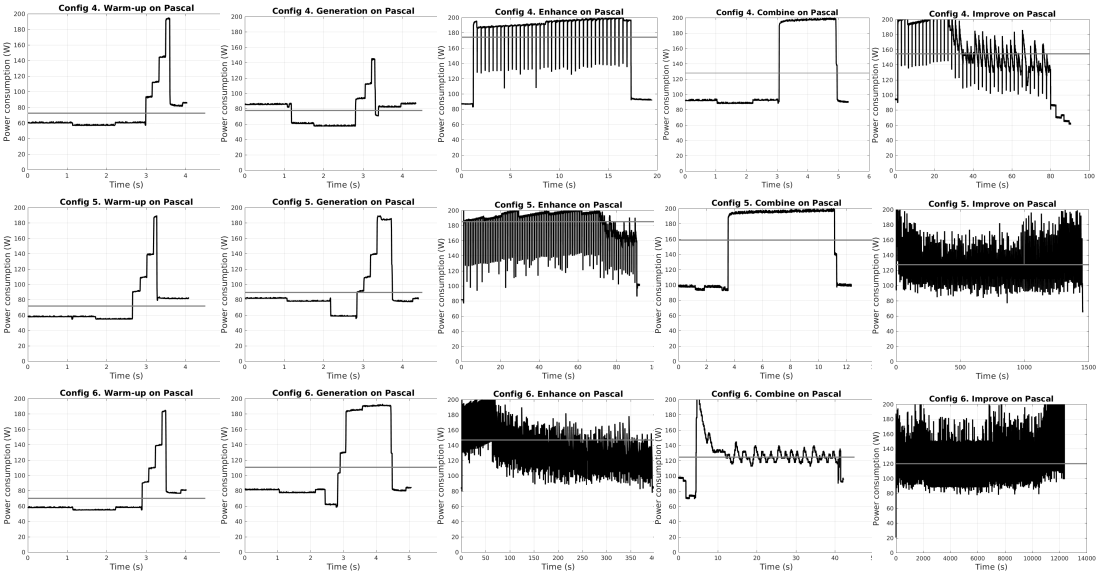

Figure 3. Power over time on every **Pascal GPU** for the 3 configurations and 5 stages (columns) of the **scatter-search algorithm**.

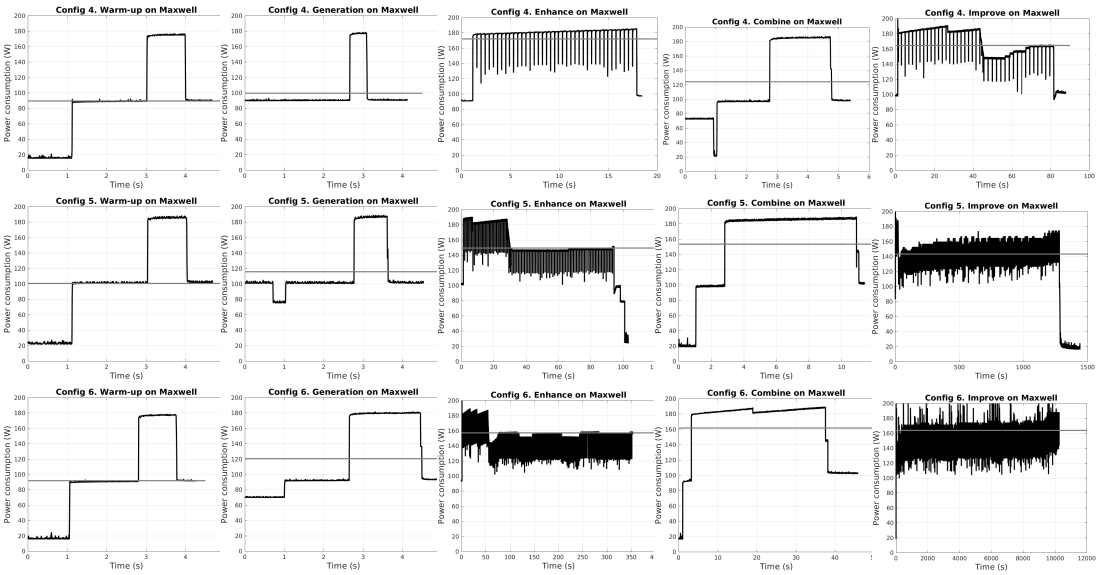

Figure 4. Power over time on every **Maxwell GPU** for the 3 configurations and 5 stages of the **scatter-search algorithm**.

DOI: 10.1002/cpe

Table XII. Power, execution times and energy consumption on every **Maxwell GPU** for each configuration and stage of the **GRASP algorithm**.

Config.	Warm-up	Generation	Enhance	All
Average power (watts per GPU)				
C7	92.92 W	111.68 W	163.07 W	158.68 W
C8	98.39 W	122.23 W	150.42 W	148.78 W
C9	100.61 W	140.23 W	153.20 W	152.53 W
C10	87.09 W	101.60 W	161.37 W	160.05 W
C11	112.82 W	118.14 W	159.20 W	158.77 W
Execution time (seconds)				
C7	4.68 s	4.19 s	115.31 s	124.19 s
C8	4.53 s	6.35 s	241.97 s	252.85 s
C9	4.27 s	6.96 s	454.75 s	465.99 s
C10	4.78 s	4.84 s	476.30 s	485.92 s
C11	4.75 s	4.62 s	932.39 s	941.77 s
Energy consumption (kilojoules per GPU)				
C7	0.43 kJ	0.46 kJ	18.80 kJ	19.70 kJ
C8	0.44 kJ	0.77 kJ	36.39 kJ	37.62 kJ
C9	0.43 kJ	0.97 kJ	69.67 kJ	71.07 kJ
C10	0.41 kJ	0.49 kJ	76.86 kJ	77.77 kJ
C11	0.53 kJ	0.54 kJ	148.44 kJ	149.52 kJ

Table XIII. Workload distribution in our multi-GPU system according to performance and energy criteria.

Con-figur.	Best workload distribution according to performance	Average Power		For optimal performance/watt, increase workload in Pascals by
		Twin Pascals	Twin Maxwells	
C1	31% Pascal, 19% Maxwell	88.13 W	94.59 W	3.66%
C2	32.5% Pascal, 17.5% Maxwell	94.87 W	109.65 W	7.78%
C3	32% Pascal, 18% Maxwell	130.42 W	133.67 W	1.24%
C4	31.5% Pascal, 18.5% Maxwell	151.08 W	158.69 W	2.51%
C5	31% Pascal, 19% Maxwell	130.78 W	143.39 W	4.82%
C6	31% Pascal, 19% Maxwell	120.81 W	163.45 W	17.34%
C7	30.5% Pascal, 19.5% Maxwell	115.47 W	158.68 W	18.70%
C8	31% Pascal, 19% Maxwell	108.97 W	148.78 W	18.26%
C9	30% Pascal, 20% Maxwell	108.34 W	152.53 W	**20.00%**
C10	32% Pascal, 18% Maxwell	127.36 W	160.05 W	12.82%
C11	31.5% Pascal, 18.5% Maxwell	125.40 W	158.77 W	13.30%

Table XIV. Power, execution times and energy consumption on every Pascal GPU for each IIEImp value of the Improve stage on configuration 5 (C5) within the scatter-search algorithm.

IIEImp value within C5	Average Power	Execution Time	Energy consumption
IIEImp= 20	115.22 W	667.96 s.	76.96 kJ.
IIEImp= 100	100.94 W	1557.06 s.	157.17 kJ.
IIEImp= 200	118.37 W	2491.61 s.	294.93 kJ.

DOI: 10.1002/cpe

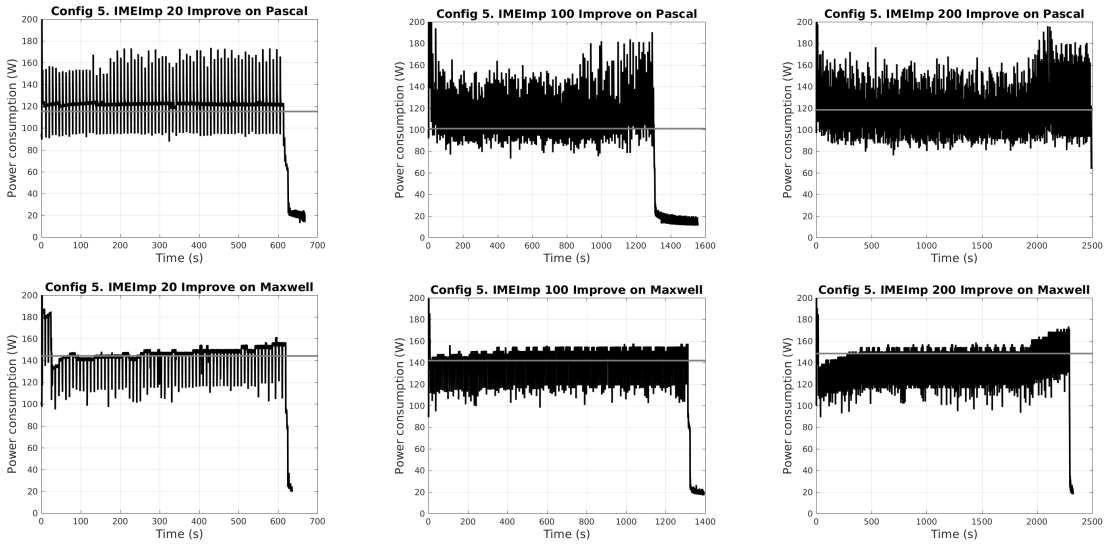

Figure 5. Power over time on every Pascal (upper) and Maxwell (lower) GPU for C5 and IIEImp equal to 20, 100 and 200.

Table XV. Set of parameters sensitive to each metaheuristic within our testbed, range of values studied and best choice according to performance and energy criteria on Pascal GPUs.

Metaheuristic involved	Moving parameter	Set of values analyzed	Best choice according to:		
			Performance	Energy	Perf./Watt
Genetic	`INEIni`	{1024, 2048, 4096}	**1024** [C1]	**1024** [c1]	**1024** [c1]
Scatter-search	`INEIni`	{512, 1024, 2048}	**512** [c4]	**512** [c4]	**2048** [c6]
	`IIEIni`	{20, 50, 100}	**20** [c4]	**20** [c4]	**100** [c6]
	`IIEImp`	{20, 100, 200}	**20** [c4]	**20** [c4]	**100** [c5]
GRASP	`INEIni`	{1024, 2048, 4096}	**1024** [c7]	**1024** [c7]	**4096** [c9]
	`IIEIni`	{100, 500, 1000}	**100** [c7]	**100** [c7]	**100** [c7]

DOI: 10.1002/cpe