Porting Estimation of Distribution Algorithms to the Cell Broadband Engine

Carlos Pérez-Miguel, Jose Miguel-Alonso and Alexander Mendiburu
Department of Computer Architecture and Technology
The University of the Basque Country
{carlos.perezm, j.miguel, alexander.mendiburu}@ehu.es

Abstract

Current consumer-grade computers and game devices incorporate very powerful processors that can be used to accelerate many classes of scientific codes. However, programming multi-core chips, hybrid multi-processors or graphical processing units is not an easy task for those programmers that deal mainly with sequential codes. In this paper, we explore the ability of the Cell Broadband Engine to run a particular Estimation of Distribution Algorithm (Univariate Marginal Distribution Algorithm for the Continuous domain). From an initial sequential version, we develop a multi-threaded one that is afterwards reworked to run on a Cell. Both versions of the code show significant improvements in performance, compared to the sequential version. We analyze the results obtained and provide some clues about the performance/cost characteristics of the tested platforms.

1 Introduction

The recent popularization of consumer hardware with parallel capabilities, such as multi-core processors (including the Cell Broadband Engine) and GPUs, that we can find in personal computers and game consoles, brings out the potential of enormous computing power for a budget. However, exploiting this potential is not easy: programs have to be reworked in order to take advantage of these parallel, sometimes hybrid, processors. They require complex programming methods that are not easy for the casual programmer. Parallelism, a challenge by itself, is not the only issue. Unfamiliar memory models, limited instruction sets, explicit communications, etc. combine to make really hard the effective exploitation of theoretically powerful machines.

The availability of powerful computers (not necessarily parallel) with large amounts of memory has encouraged the design and implementation of non-trivial algorithms to deal with different kinds of complex optimization problems. Some of these problems can be solved via an exhaustive search over the solution space, but in most cases this brute force approach is unaffordable. In these situations, heuristic methods (deterministic or non deterministic) are often used, which search inside the space of promising solutions. Some heuristic approaches are specifically designed to find good solutions for a particular problem, but others are presented as a general framework adaptable to many different situations. Among this second group (general designs), there is a family of algorithms that has been widely used in the last decades: Evolutionary Algorithms (EAs). This family comprises, as main paradigms, Genetic Algorithms (GAs) [10, 13], Evolution Strategies [29], Evolutionary Programming [9] and Genetic Programming [16]. Even though processing speeds grow fast, the requirements of this class of algorithms grow even faster. No matter the computing power available, we can always find a harder problem that cannot run in our machines, or can do so but takes too long to run.

Therefore, we have the problem (the execution of complex optimization algorithms) and the platform (consumer-grade parallel hardware). We need to port the application to the target hardware, in order to efficiently exploit the latter, to accelerate program execution and/or obtain better solutions.

In this paper, we extend the work presented in [7], in which we tested the suitability of the Cell as a computing platform for one algorithm of the family of Estimation of Distribution Algorithms. In particular, we studied the Univariate Marginal Distribution Algorithm (UMDA), applied to solving the $OneMax$ function, a well-known toy problem. Due to the characteristics of UMDA (discrete variables and simple probabilistic model) and the problem to solve, we observed a discouragingly poor performance in terms of execution time and scalability, compared to that provided by a commodity multi-core personal computer. The main reason for this disappointing behavior has to be found in the computation/communication ratio of this particular program. As the vector units of the Cell have only a small local memory, the time needed to move data from/to the main

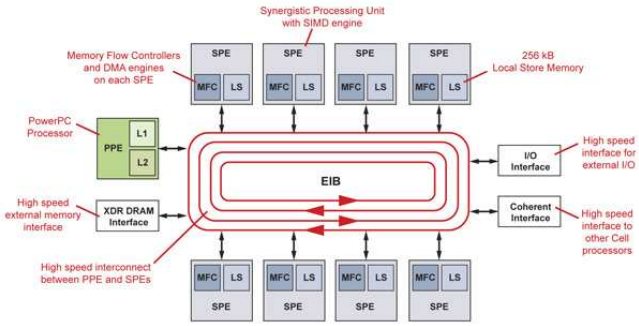

Figure 1. Cell Broadband Engine Architecture.

memory dominated the overall execution time, thus impeding the full exploitation of the Cell's power.

As the family of EDAs comprises algorithms in both discrete and continuous domains, we decided to extend our previous work by testing an algorithm similar to UMDA, but for continuous domains. It is called Univariate Marginal Distribution for the Continuous domain (UMDA$_c$). The main purpose is to evaluate the performance of the Cell platform when working with floating-point numbers.

We followed the same methodology used to parallelize UMDA. Starting from an initial sequential version, the algorithm was ported to a parallelized version capable of running on a multi-core, symmetric system (such as a Quad-Core Intel Xeon processor). The parallel version was, afterwards, reworked to run on a multi-core, hybrid system (the Cell Broadband Engine). To further accelerate the programs, vectorization techniques were applied in a subsequent step. The final results show that our parallel and vectorized implementations of UMDA$_c$ can very effectively exploit the potential of the Cell and of multi-core CPUs.

The rest of the paper is organized as follows. Section 2 discusses the architecture of the Cell Broadband Engine, with special emphasis on those characteristics visible to the programmer. Section 3 summarizes the main characteristics of Evolutionary Algorithms and, in particular, the family of Estimation of Distribution Algorithms in the continuous domain, as well as the problem to be solved with our implementation. Section 4 explains the porting of the algorithm and the vectorization process in the two platforms. Section 6 shows the experiments made in order to compare both implementations. Conclusions of this work are summarized in Section 7

2 Cell Broadband Engine

The Cell is a microprocessor system that integrates, into a single chip, a Power-based processor (Power Processing Element, PPE), eight vector co-processors (Synergistic Processing Elements, SPEs), a memory interface, input/output interfaces and a high-speed ring that acts as the interconnection fabric for the remaining elements [15] (see Figure 1, extracted from [31]). A programmer who wants to take full advantage of a Cell has to work with two different instruction sets: one for the PPE and another one for the SPEs. This usually means using two compilers, and dealing with different strategies to optimize code running in different processors. To mention just an example, the PPE *can* deal with vector operations to accelerate parts of the program, but the SPEs *must* work with vectors – its efficiency with scalars is not brilliant.

Another peculiarity of the Cell is its memory organization. From a programmer's point of view, the PPE has full, direct access to the system's main memory. However, the SPEs have direct access only to a very limited sized (256 KB) local store. All the data processed by an SPE has to be previously transferred to its local store, and the resulting data, if required by the PPE or another SPE, has to be explicitly transferred too. To that purpose, each SPE has a companion Memory Flow Controller (MFC) that takes care of this transfer. A good programmer can manage to make an SPE and its MFC work at the same time, processing pieces of data while transferring new ones. A careless programmer may try to simultaneously move too much data through the interconnection fabric, which would become a bottleneck because of its limited capacity.

Challenges for the programmer are, therefore, manifold: different instruction sets; real necessity of working with vector instructions; limited memory size; explicit transfer of data between different memory blocks, etc. Adaptation of an application to this architecture is not trivial: a few applications may have a natural mapping, but most require exhaustive reworking. A common model for organizing Cell applications, but by no means the only one, is to use the PPE as a main processor running most of the application's logic, using the SPEs as acceleration co-processors [14]. The most compute-intensive sections of the original PPE-only code are identified, and reworked to make them run in parallel in the available SPEs. As just mentioned, this is a complex task that requires careful organization of data structures, data movement, synchronization, vectorization, etc. In summary, using this computing platform will require the programmer to deal with uncommon techniques. For example, all the data movements between the PPE and the SPEs are controlled by the last ones (SPEs). They must manage their own buffers in the main memory and synchronize with the PPE (generally using mailboxes), and this must be done explicitly by the programmer.

Regarding hardware platforms, IBM sells Cell-based blades for use in general-purpose systems, but the most popular, consumer-available platform to get acquaintance with

this processor is Sony's PlayStation 3 game console. The PS3 can be easily converted in a GNU/Linux "computer" with all the necessary toolkits to develop and run Cell applications, by means of any of the several available Linux distributions. The main limitation of this platform is that only six SPEs are available: Sony guarantees just seven working SPEs (to increase manufacture yield), and one is always reserved for the operating system. In this work we use a PS3 running Fixstars' Yellow Dog Linux [1].

3 Evolutionary Algorithms

The main characteristic of Evolutionary Algorithms is that they use techniques inspired by the natural evolution of the species. In nature, species change across time; individuals evolve, adapting to the characteristics of the environment. This evolution leads to individuals with better characteristics. This idea can be translated to the world of computation, using similar concepts:

Individual: Represents a possible solution for the problem to be solved. Each individual has a set of characteristics (genes) and a fitness value (based on its genes) that denotes the quality of the solution it represents.

Population: In order to look for the best solution, a group of individuals is managed. An initial population is created randomly, and will change across time, evolving towards members with different (and supposedly better) characteristics.

Breeding: Several operators can be used to emulate the breeding process present in nature: mixing different individuals (crossover) or changing a particular one (mutation). These operators are used to obtain new individuals, expected to be better than the previous ones.

In the last two decades, Genetic Algorithms have been widely used to solve different problems, improving in many cases the results obtained by previous approaches. However, GAs require a large number of parameters (for example, those that control the creation of new individuals) that need to be correctly tuned in order to obtain good results. Generally, only experienced users can do this correctly and, moreover, the task of selecting the best choice of values for all these parameters has been suggested to constitute itself an optimization problem [11]. In addition, GAs show a poor performance in some problems (deceptive and separable problems) in which the existing crossover and mutation operators do not guarantee that better individuals will be obtained changing or combining existing ones.

Some authors [13] have pointed out that making use of the relations between genes can be useful to drive a more "intelligent" search through the solution space. This concept, together with the limitations of GAs, motivated the creation of a new type of algorithms grouped under the name of Estimation of Distribution Algorithms (EDAs).

Pseudo-code for the EDA framework.

Step 1. Generate the first population D_0 of M individuals and evaluate all of them
Step 2. **Repeat** at each generation l until a stopping criterion is fulfilled
Step 3. Select N individuals (D_l^{Se}) from the D_l population following a selection method
Step 4. Obtain from D_l^{Se} an n dimensional probability model that shows the interdependencies between variables
Step 5. Generate a new population D_{l+1} of M individuals based on the sampling of the probability distribution $p_l(\boldsymbol{x})$ learnt in the previous step

Figure 2. Common outline for Estimation of Distribution Algorithms (EDAs).

EDAs were introduced in the field of Evolutionary Computation in [22], although similar approaches can be previously found in [33]. In EDAs there are neither crossover nor mutation operators. Instead, the new population of individuals is sampled from a probability distribution, which is estimated from a database that contains the selected individuals from the current generation. Thus, the interrelations between the different variables that represent the individuals are explicitly expressed through the joint probability distribution associated with the individuals selected at each generation. A common pseudo-code for all EDAs is presented in Fig. 2.

Steps 3, 4 and 5 will be repeated until a certain stop criterion is met (e.g., a maximum number of generations, a homogeneous population or no improvement after a specified number of generations). The probabilistic model learnt at step 4 has a significant influence on the behavior of the EDA from the point of view of complexity and performance.

For detailed information about the characteristics of EDAs, and the algorithms that form part of this family, see [19, 27, 20, 28].

4 The UMDA$_c$ Algorithm

The Univariate Marginal Distribution Algorithm for the continuous domain (UMDA$_c$) [17, 18] is an EDA in which it is assume that there are no dependencies between the variables involved in the problem. It assumes that the joint density function follows a n-dimensional normal distribution, which is factorized by a product of one-dimensional and independent normal densities. In every generation and for every variable, the UMDA$_c$ performs some statistical tests in order to find the density function that best fits the sampling of that variable.

The UMDA$_c$ is a structure identification algorithm because the density components of the model to be learnt are identified via hypothesis tests. This estimation of parameters is performed, after the densities are identified, by their maximum likelihood estimates. If all the univariate distributions are normal, then the two parameters to be estimated at each generation and for each variable are the mean, μ_i^l, and the standard deviation, σ_i^l. It is well known that their respective maximum likelihood estimates are:

$$\widehat{\mu_i^l} = \overline{X_i^l} = \frac{1}{N}\sum_{r=1}^{N} x_{i,r}^l \quad (1)$$

$$\widehat{\sigma_i^l} = \sqrt{\frac{1}{N}\sum_{r=1}^{N}(x_{i,r}^l - \overline{X_i^l})^2} \quad (2)$$

Being N the number of individuals in the population and $x_{i,r}^l$ the different variables which compound each individual. This two parameters, μ and σ, will be used later to generate new individuals. For this purpose, an adaptation of the Probabilistic Logic Sampling (PLS) proposed in [12] is used. In this method the instances are generated one variable at a time in a forward way. For the simulation of a univariate normal distribution, a simple method based on the sum of 12 uniform variables is applied [30].

5 Parallelization of UMDA$_c$

Evolutionary Algorithms require, in general, long execution times. For this reason, researchers often apply parallel techniques to reduce running times. These techniques are also useful to improve accuracy or to manage larger problems with the same time budget [5, 13].

There are two basic approaches to parallelize EAs: parallelization of program loops, and division of the population into several independent subpopulations (islands model). When using the islands approach, the single population used in sequential algorithms is split into several subpopulations (islands). These islands evolve independently, and exchange information about their best individuals with a predefined frequency. These models are suitable to distributed systems, because each island can be mapped onto a separate processor, and the amount of communications required between islands is not very large. A more conservative approach starts with a sequential algorithm like that described in Figure 2, parallelizing parts of it in order to speed-up the execution time but without changing the semantics of the algorithm. There exist different proposals for GAs [8] or EDAs [24, 25, 21]. The most time-consuming portions of the code are identified and rewritten to take advantage of a parallel computer. Among the techniques to parallelize the code, or portions of it, the Manager-Worker model is a popular one: a Manager task runs the program, and delegates CPU-intensive parts to a collection of Worker tasks.

The selection of the parallelization paradigm has to be done taking into account the characteristics of the available computing platform. We will not work with a cluster of computers, but with on-chip multiprocessors – in particular, with a Cell system. The limited memory of the Cell's SPEs does not allow us to run a complete EDA (an island) in each SPE. Therefore, we have to opt for the Manager-Worker approach. The Manager task will run on the PPE and the Workers on the SPEs.

The parallel approach has been done starting from a sequential version of UMDA$_c$ written in C++. Before designing the parallel version for the Cell, we considered interesting, for comparison purposes, to design a parallel version based on the Posix Threads (pthreads) library [6], that could be executed on a multi-core personal computer. Also, as both multi-core Intel-based machines and the Cell include vector instructions (that can accelerate our application), we have also evaluated manually vectorized versions of the codes. Both versions, the Pthread's one and the Cell's one, were developed using the GNU Compilers; more specifically the version 4.3.2 over the Intel platform and the 4.1.1 one for the Cell.

Following the Manager-Worker scheme, the UMDA$_c$ algorithm will be executed by the main thread (Manager), and when necessary, it will ask the Workers for help. In particular, workers will collaborate in these phases:

- Learning the probabilistic model: As introduced previously, UMDA$_c$ uses a very simple probabilistic model, that assumes that there is no relation between the variables. Therefore, once the population has been selected, the Manager will ask the workers to obtain parts of the mean and standard deviation –our implementation assumes normal distributions–, for a subset of the selected population. Once each Worker has finished, the Manager creates the main model based on the partial values.

- Sampling and Evaluation: These two steps usually

Pseudo-code for the EDA over the Cell's PPE.

Step 1. Generate the first population D_0 of M individuals and evaluate all of them.
Step 2. **Repeat** at each generation l until a stopping criterion is fulfilled
Step 3. Select N individuals (D_l^{Se}) from the D_l population following a selection method
Step 4. Obtain from D_l^{Se} an n dimensional probability model that shows the interdependencies between variables
Synchronization via Mailboxes.
Step 5. Ask the SPEs to generate N individuals.
Synchronization via Mailboxes.

Figure 3. Common outline for Parallelized EDAs over the Cell's PPE.

come together. That is, based on the model learnt in the previous step, new individuals will be created and evaluated. Again, the Manager will ask the workers to create (and evaluate) a subset of individuals. The number of individuals to be managed by each worker can be established statically, or assigned dynamically using an on-demand scheme. That is, when the evaluation of the individuals takes always the same computing time, a static assignation can be done. However, if the time required to evaluate the individual depends on the values it takes, the on-demand scheme would be preferable. In our experiments we use static workload distribution.

For our UMDA$_c$ algorithm, the chunks of code that conform the kernel to be executed in the SPEs will be the code to generate individuals, to evaluate them. We will not execute the learning phase in the SPU because that would imply the continuous movement of individual between LS and main memory to order and reduce the population, and vice versa. In any case, the amount of code to execute for each individual in the learning process is little in comparison with the amount of time required to send an individual via DMA.

In the Figures 3 and 4 we can see the pseudo-code for the parallelized version of the UMDA$_c$ executed in the PPE and in the SPEs. We can compare it with the pseudo-code in Figure 2 which contains the code of the sequential version.

In order to test the performance of our different implementations of UMDA$_c$, we used them to solve an artificial problem, the *Griewangk* minimization problem [32]. The

Pseudo-code for the EDA over the Cell's SPEs.

Synchronization via Mailboxes.
Step 1. Transfer this model from the PPE to each SPE.
Step 2. Generate a new population D_{l+1} of $M/6$ individuals based on the sampling of the probability distribution $p_l(\boldsymbol{x})$ learnt in the previous step. Each individual generated will be transferred via DMA from the SPE to the PPE.
Synchronization via Mailboxes.

Figure 4. Common outline for Parallelized EDAs over the Cell's SPEs.

fitness function is defined as follows:

$$F(x) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) \qquad (3)$$

The range of all the variables of the individual is $-600 \leq x_i \leq 600, i = 1, \ldots, n$, and the fittest individual corresponds to a value of 0, that only can be obtained when all the variables of the individual are 0. This well-known problem has been selected to deal with a fitness function that does not require too much execution time.

6 Measuring the performance of the UMDA$_c$

In order to test the performance of the multi-threaded version of the code, we completed several experiments on an Intel Xeon Quad-Core computer. Different individual sizes were used (ranging from 100 to 3,000 variables), using a population size of $2,5L$, being L the number of variables of the individual. Tests were performed with 1-4 threads. UMDA$_c$ was stopped after computing 50 generations. The results of these experiments are shown in Figure 5.

According to the results, it can be seen that the multi-threaded version has an adequate behavior from the point of view of scalability. Therefore, the second step of this work would be to test the suitability of the Cell for the execution of this EDA.

We made the same experiments with the Cell version executed over a PS3. In this case, individual sizes range from 100 to 1,800 variables (due to limitations on the SPE's local store). The number of threads (running in different SPEs) was varied from 1 to 6. The rest of the parameters – population size, stopping criterion – were as in the previous experiments. Results can be seen in Figure 6. Clearly, the solution scales with the number of SPEs, providing good

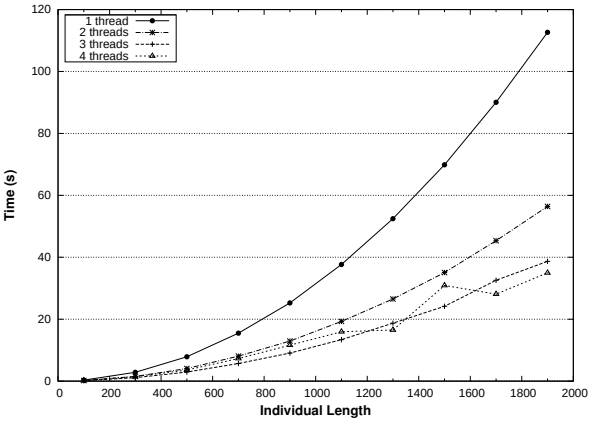

Figure 5. Execution times for the Quad-Core Intel Xeon (using Pthreads).

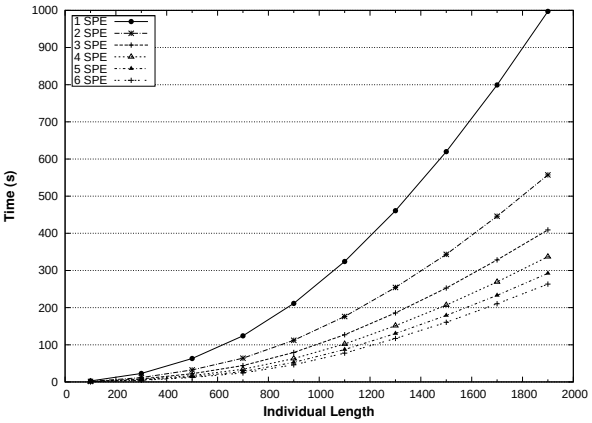

Figure 6. Execution times for the Cell.

performance levels. This is good news because, as reported in our previous work [7], for the UMDA applied to discrete domains, the performance of Cell was very poor. Therefore, we can state that the Cell is a more suitable platform to run UMDA$_c$ (continuous domain, floating-point numbers).

Vector processors give us the possibility to operate over multiple data elements with a single machine instruction. As the two platforms used in our experiments integrate support for vector operations, we decided to modify the UMDA$_c$ implementations to take advantage of this feature. One of the more expensive parts of the UMDA$_c$ is the sampling and evaluation of new individuals. In the sampling step, an adaptation of the Probabilistic Logic Sampling (PLS) for the continuous domain is used. This technique needs random values to create new individuals, and thanks to the vectorization, the process of obtaining these random values can be done in groups of four. A similar idea was applied to the evaluation of the individuals, rewriting the evaluation function to compute the different operations: square root, cosine, square, etc. in groups of four.

The vectorized version of the Cell implementation was made using SPU intrinsics [4] and the *libmisc* library included in the IBM Cell SDK [2], which implements vectorized versions of a uniform random number generator. For the Intel platform, we used SSE intrinsics [3] and implemented our own vectorized version of a linear congruential generator [26].

We repeated the experiments with the vectorized implementations of UMDA$_c$. Results are shown in Figures 7 and 8. As we can observe, the acceleration levels obtained via vectorization are impressive for both platforms. In Figure 9 we have plotted the scalar/vector acceleration ratio (speedup) for 1 thread (Xeon) and 1 SPU (Cell) for different individual sizes. We can see that for the Xeon platform the maximum speedup is achieved for individuals of size larger than 300, and that this figure remains stable for longer individuals. However, for the Cell, a peak is observed for individuals of size 500, but for larger problems speedup drops. We need to further explore this issue, but we suspect that the cause is on the bottlenecks in the internal Cell interconnection network, which saturates when large DMA transfers occur too often – that is, when the application is too communication-biased. These issues, such as memory and communication management, must be carefully studied when implementing Evolutionary Algorithms in architectures such us Cell or GPUs [23].

When comparing the two platforms, Figure 5 versus Figure 7, and Figure 6 versus Figure 8, we can observe that, even using vectorized code, the Xeon Quad-Core is one order of magnitude faster. In the selection of the best platform we can not forget the cost of each solution. In terms of code development, the Xeon Quad-Core is an easier platform to program multi-thread code, but harder at the vectorization process than the Cell. Also if we compare their prices, the cheapest Cell platform, the PlayStation 3, is about half price the Xeon Quad-Core, giving us an enormous computing power for a budget.

7 Conclusions

In this paper we have evaluated several parallel implementations of UMDA$_c$ on two different platforms: a Quad-Core Intel Xeon and a Cell (PlayStation 3) system. We have shown that the well-known Manager-Worker parallelization scheme can be successfully applied to port our algorithm to these platforms: in both cases we are capable of exploiting the availability of multiple cores, homogeneous for the Xeon system, and heterogeneous for the Cell. Additionally, as all the processors involved in our experiments integrate support for vector operations, we have tested manually vectorized versions of the programs, which were implemented

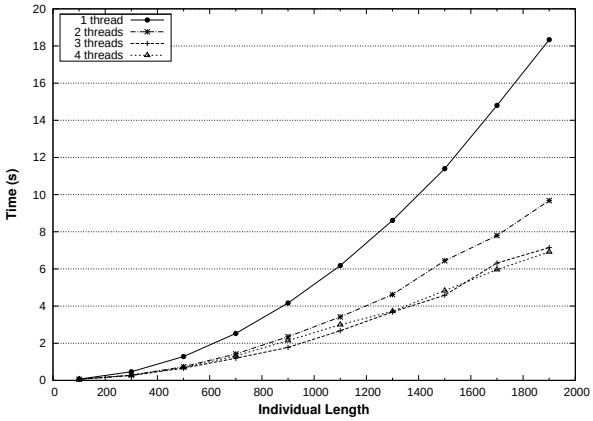

Figure 7. Execution times for the vectorized version on the Quad-Core Xeon.

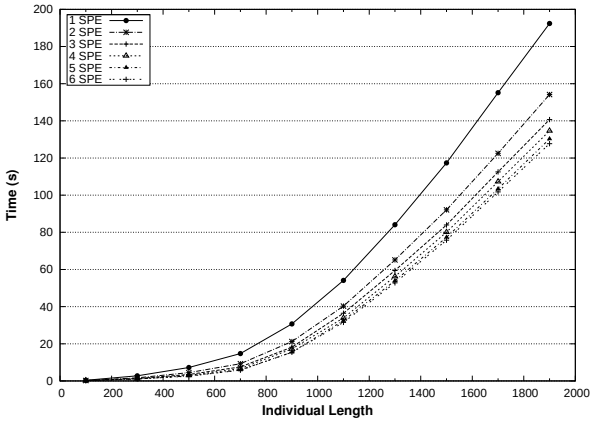

Figure 8. Execution times for the vectorized version on the Cell.

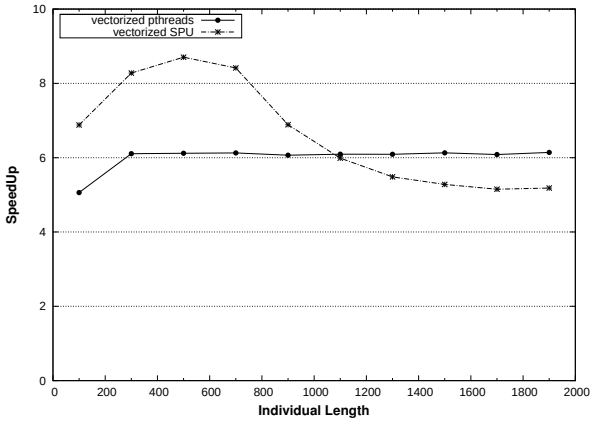

Figure 9. Impact of vectorization in the Xeon Quad-Core and the Cell.

using different repertoires of intrinsics. The vectorized versions of the programs perform exceedingly well, providing additional speedups on the 5-9 range (depending on the platform and the problem size).

This experience with the Cell has shown us the difficulty of develop a solution for this architecture, but also its capacities when addressing an EDA. Our opinion about this subject is that, even if the architecture is hard to master, once the developer has the knowledge about the platform and having a suitable problem, the task is more affordable, giving us a powerful machine for a budget.

As a future work, it will be interesting to use different tools, su h as the IBM CBE Simulator included in the IBM's SDK in order to compare and analyze in detail the results presented in this paper. The use of the simulator can be very helpful to better understand some particularities of this architecture, such as, for example, the interconnection network.

Acknowledgements

This work has been supported by the Ministry of Education and Science (Spain), grant TIN2007-68023-C02-02, by Saiotek and Research Groups 2007-2012 (IT-242-07) programs from the Basque Government, TIN2008-06815-C02-01 and Consolider Ingenio 2010 - CSD2007-00018 projects (Spanish Ministry of Science and Innovation) and COMBIOMED network in computational biomedicine (Carlos III Health Institute).

References

[1] Fixstars Corp. home page.
[2] IBM Cell SDK Example Library API Reference.
[3] Intel(R) C++ Compiler Intrinsics Reference.
[4] PPU and SPU C/C++ Language Extension Specification.
[5] W. Bossert. Mathematical optimization: Are there abstract limits on natural selection? In P. S. Moorehead and M. M. Kaplan, editors, *Mathematical Challenges to the Neo-Darwinian Interpretation of Evolution*, pages 35–46. The Wistar Institute Press, Philadelphia, PA, 1967.
[6] D. R. Butenhof. *Programming with POSIX Threads*. Addison–Wesley Professional Computing Series, 1997.
[7] J. M.-A. C. Perez-Miguel and A. Mendiburu. An analysis of iterated density estimation and sampling in the UMDAc algorithm. In *GECCO '09: Proceedings of the 2009 conference on Genetic and evolutionary computation*, pages 2491–2498, New York, NY, USA, 2009. ACM Press.
[8] E. Cantú-Paz. *Efficient and accurate parallel genetic algorithms*. Kluwer Academic Publishers, 2000.
[9] L. J. Fogel. Autonomous automata. *Industrial Research*, 4:14–19, 1962.
[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison–Wesley, Reading MA, 1989.

[11] J. J. Grefenstette. Optimization of control parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

[12] M. Henrion. Propagating uncertainty in Bayesian networks by probabilistic logic sampling. In R. D. Shachter, T. S. Levitt, L. N. Kanal, and J. F. Lemmer, editors, *UAI*, pages 149–163. North-Holland, 1988.

[13] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1992.

[14] IBM. *Software Development Kit for Multicore Acceleration. Programming Tutorial. Version 3.1*. 2008.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. J. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005.

[16] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[17] P. Larrañaga, R. Etxeberria, J. Lozano, and J. Peña. Optimization by learning and simulation of Bayesian and Gaussian networks. Technical Report KZZA-IK-4-99, Department of Computer Science and Artificial Intelligence, University of the Basque Country, 1999.

[18] P. Larrañaga, R. Etxeberria, J. Lozano, and J. Peña. Optimization in continuous domains by learning and simulation of Gaussian networks. In L. D. Whitley, D. E. Goldberg, E. Cantú-Paz, L. Spector, I. C. Parmee, and H. G. Beyer, editors, *GECCO*, pages 201–204. Morgan Kaufmann, 2000.

[19] P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, 2002.

[20] J. A. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea, editors. *Towards a New Evolutionary Computation. Advances on Estimation of Distribution Algorithms*, volume 192 of *Studies in Fuzzines and Soft Computing*. Springer, 2005.

[21] A. Mendiburu, J. A. Lozano, and J. Miguel-Alonso. Parallel implementation of EDAs based on probabilistic graphical models. *IEEE Transactions on Evolutionary Computation*, 9(4):406–423, 2005.

[22] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. In H. M. Voigt, W. Ebeling, I. Rechenberger, and H. P. Schwefel, editors, *PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 178–187. Springer, 1996.

[23] J. H. O. Garnica, J.L. Risco-Martín and J. Lanchares. Speeding-up resolution of deceptive problems by a parallel gpu-cpu architecture. *WPABA08 (PACT08)*, 2008.

[24] J. Ocenasek and J. Schwarz. The parallel Bayesian optimization algorithm. In *Proceedings of the European Symposium on Computational Intelligence*, pages 61–67, 2000.

[25] J. Ocenasek and J. Schwarz. The distributed Bayesian optimization algorithm for combinatorial optimization. In *EUROGEN - Evolutionary Methods for Design, Optimisation and Control, CIMNE*, pages 115–120, 2001.

[26] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Commun. ACM*, 31(10):1192–1201, 1988.

[27] M. Pelikan, D. E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002.

[28] M. Pelikan, K. Sastry, and E. Cantú-Paz. *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications (Studies in Computational Intelligence)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[29] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann–Holzboog, Stuttgart, 1973.

[30] B. D. Ripley. *Stochastic Simulation*. John Wiley and Sons, 1987.

[31] J. Rudin. Accelerating persistent surveillance radar with the cell broadband engine. *Embedded Technology Journal*, 2008.

[32] A. A. Törn and A. Zilinskas. *Global Optimization*, volume 350 of *Lecture Notes in Computer Science*. Springer, 1989.

[33] A. A. Zhigljavsky. *Theory of Global Random Search*. Kluwer Academic Publishers, 1991.