Um Novo Algoritmo Compacto Genético em Hardware: emCGA-HW

Rafael R. da Silva[3], Carlos R. Erig Lima[1,2,3], Heitor S. Lopes[1,2,3]

1 – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI/UTFPR)
2 – Programa de Pós-Graduação em Computação Aplicada (PPGCA, DAINF - DAELN, UTFPR)
3 – Universidade Tecnológica Federal do Paraná (UTFPR) – Curitiba – PR, Brasil (41 33104761)

Resumo - O algoritmo compacto genético (CGA - Compact Genetic Algorithm) é um algoritmo genético especialmente projetado para atender requisitos de implementação em hardware. Este trabalho propõe uma nova extensão do CGA, denominada emCGA-HW. O emCGA-HW é comparado com o emCGA em software, já descrito em trabalhos anteriores. Em ambas as abordagens é introduzido um novo operador de mutação objetivando-se uma melhor performace geral do CGA. São apresentados resultados desta comparação, enfatizando-se a significativa redução do tempo execução da abordagem em hardware em comparação com a abordagem em software, sem perda de qualidade do resultado.

Palavras Chaves – FPGA; CGA; Genetic Algorithm; Reconfigurable Logic)

I. INTRODUCÃO

Algoritmos Genéticos (GAs - *Genetic Algorithms*) são eficientes para resolver problemas de otimização, sendo aplicados com sucesso em inúmeros problemas de engenharia. No entanto, para algumas aplicações, o custo computacional do GAs é muito alto, demandando um excessivo tempo de execução ou recursos de *hardware* proibitivos. Uma alternativa é explorar GAs com menor complexidade computacional, permitindo a execução em plataformas menos poderosas. Alternativamente, o GA pode ser executado em arquiteturas paralelas, utilizando-se dispositivos lógicos reconfiguráveis [1].

Em geral, um GA requer uma grande quantidade de memória para armazenar a população de indivíduos. Isso pode ser uma importante desvantagem, particularmente para projetos em *hardware*. Além disso, é importante observar que, tipicamente, a evolução da população do GA é executada de forma sequencial, apesar dos GA serem intrinsecamente algoritmos paralelos. Neste contexto, o CGA (Algoritmo Genético Compacto) pode obter um desempenho competitivo com um GA tradicional (SGA - Algoritmo Genético Simples), utilizando menos memória para armazenar a população. Isto é possível porque CGA opera com um vetor de probabilidade para representar a população. O CGA evolui este vetor de probabilidades, imitando o comportamento do SGA. Devido à simplicidade do CGA e dos seus baixos requisitos de memória, alguns trabalhos têm apresentado versões do CGA em *software* ou *hardware*, mostrando desempenhos interessantes.

Por outro lado, o objetivo do conceito de *hardware* reconfigurável é permitir uma fácil e rápida adaptação de um projeto a uma contínua evolução tecnológica. Pode-se citar o paralelismo real (em contraponto ao modelo de Von Newmann), a fácil hierarquização e modularização do projeto e a facilidade de atender diferentes requisitos de tempo real em sistemas complexos, como algumas das vantagens da utilização de *hardware* reconfigurável. Como motivação adicional, há a disponibilidade de uma grande gama de dispositivos comerciais disponíveis, com diversas ferramentas de suporte bem desenvolvidas. Por exemplo, algumas placas de desenvolvimento baseadas em FPGAs (*Field Programmable Gate Arrays*) presentes no mercado, permitem a utilização de mais de 100.000 LUTs (*Look-up tables*), várias memórias internas e externas, diversos módulos dedicados para utilização do usuário (multiplicadores de alta velocidade e PLLs) interfaces de comunicação padrões (incluindo ethernet), com custo orbitando a faixa de U$300,00.

Na próxima seção será apresentado, com mais detalhes, o GA em *hardware*. Na seção seguinte, será apresentada a metodologia realizada e, na sequência, os resultados obtidos nos experimentos e as conclusões finais.

II. IMPLEMENTAÇÃO DE GA EM HARDWARE

O CGA foi proposto por Harik, Lobo e Goldberg [2]. O CGA é um EDA que gera descendentes através de um modelo estatístico da população de pais, em vez de usar os tradicionais operadores de recombinação e mutação. Este modelo estatístico foi inspirado pelo modelo *random walk* apresentado em por Harik et al. [3]. Neste trabalho, os autores fizeram estimativa precisa do tempo de convergência para uma classe especial de problemas no GA.

Apesar do sucesso da utilização de GAs para solução de diversos problemas de otimização em engenharia, permanece o fato de ser um algoritmo que demanda plataformas de alto desempenho para serem executados de forma eficiente. Para

minimizar esta limitação, vários trabalhos propuseram diferentes versões em *hardware* dos GAs.

Por exemplo, o HGA (*Hardware Genetic Algorithm*), proposto por Scott e Seth [4], e o SPGA (*Splash ParallelGenetic Algorithm*), proposto por Graham e Nelson [5], implementaram um GA em *hardware* usando máquinas de estado finitas. Outro método utilizado foi a matriz sistólica, introduzido por Bland e Megson [6]. Técnicas de pipeline foram utilizadas no trabalho sobre SSGA (*Steady-State Genetic Algorithm*), apresentado por Shackleford et al. [7] [8], e no trabalho sobre PGA (*Pipelined Genetic Algorithm*), apresentado por Saenz et al. [9]. Pode-se citar ainda o algoritmo VGP-I, apresentado por Kavvadias, Giannakopoulou e Nikolaidis [10], o qual descreve uma arquitetura customizada para acelerar GAs. Em comum, embora essas abordagens tenham alcançado um melhor desempenho em velocidade, resultaram em projetos complexos, exigindo uma grande estrutura de *hardware* para sua realização.

As versões em *hardware* dos GA apresentam outro gargalo, além da complexidade computacional: a quantidade de memória para armazenar a população. Este parâmetro é crítico e pode demandar uma grande quantidade de recursos de *hardware* [11]. Em função disto, é possível identificar na literatura duas abordagens principais: obter uma aceleração do GA usando SGA ou SSGA (usando *hardware* dedicado ou processadores reconfiguráveis) ou obter extensões do CGA em *hardware*.

Considerando apenas a segunda abordagem (usada neste trabalho), outros trabalhos podem ser mencionados. A implementação de *hardware* do CGA, proposta por Aporntewan e Chongstitvatana [11], era 1000 vezes mais rápida do que a versão em *software* equivalente. Foi projetado em Verilog HDL (*Verilog Hardware Description Language*) e implementado usando um FPGA (*Field Programmable Gate Array*) Xilinx Virtex V1000FG680, rodando com um clock 20MHz e utilizando 15.210 elementos lógicos. Sua versão do *software* foi escrita em C e implementada usando em uma plataforma Ultra Sparc II 200 MHz. O problema utilizado em ambas as versões foi o OneMax, o mesmo proposto por [2]. O mCGA, uma extensão do trabalho prévio, foi apresentado por J. Gallagher, Vigraham e Kramer [12] para otimizar o problema OneMax, utilizando um cromossomo de 32 bits e uma população simulada de 255 indivíduos. O mCGA foi desenvolvido em VHDL e implementado em uma FPGA VirtexII xc2v1000. Foi comparado com o mesmo algoritmo em C, executado em uma plataforma de Ultra Sparc II 200MHz. O mCGA em *hardware* superou em 1000 vezes a velocidade de execução da versão em *software*. O mCGA demandou 18.732 elementos lógicos.

III. METODOLOGIA

O mCGA [11], mencionado anteriormente, utiliza um operador de mutação. Na verdade, este não é o único operador de mutação relatado na literatura. Outro operador de mutação, chamado MBBCGA, foi proposto por Jewajinda e Chongstivatana [13], melhorando o desempenho do CGA. Em ambos, MBBCGA e mCGA, o operador de mutação não se concentra no controle da diversidade populacional, mas na busca local. No trabalho MBBCGA a mutação é aplicada no indivíduo elite da geração corrente para gerar um novo indivíduo. Depois, o melhor desses indivíduos será a elite para a próxima geração. No mCGA a mutação é aplicada no primeiro indivíduo para substituir a segunda geração da CGA convencional.

O presente trabalho usa um novo operador de mutação proposto em originalmente em [14], que visa melhorar a qualidade das soluções, mantendo uma velocidade de convergência razoável. O novo operador proposto permite um controle mais eficiente da pressão de seleção, ajustando a diversidade da população como consequência da manipulação do vetor de probabilidade. Comparando-se o operador de mutação proposto com o apresentado pelo mCGA, observa-se a diminuição o número de torneios por geração e, consequentemente, o número total de avaliações por geração. A consequência é uma significativa melhoria na velocidade de convergência do algoritmo. O novo CGA proposto, resultante da utilização da mutação é denominado emCGA (*elitism with mutation CGA*). Basicamente, o operador de mutação proposto altera a fase de geração aleatória, mudando o cromossomo gerado através do vetor de probabilidade. A versão do emCGA em *hardware*, apresentada neste trabalho, é denominada emCGA-HW.

A comparação entre o emCGA e algoritmos similares, usando várias funções de Fitness, já foi feita por Silva, Lopes e Erig Lima [14]. Neste trabalho, só será apresentado uma função binária de Fitness, denominada Sum Function, descrita nas equações (1), (2) e (3), considerando um cromossomo de 8 genes, cada gene com 8 bits. Este é um problema de minimização e a melhor solução é obtida quando todos os elementos do vetor convergem para zero. Baseado nos resultados obtidos em [14], uma população de 256 indivíduos foi utilizada. O emCGA-HW inspira-se também no algoritmo CGA em *hardware* apresentado por Aporntewan e Chongstitvatana [11].

$$f(X) = \sum_{i=0}^{8} x_i \quad \forall x \in Z \tag{1}$$

$$f(\dot{X}) = \min(f(X)) = 0 \tag{2}$$

$$\dot{X} = [0,0,0,0,0,0,0,0] \tag{3}$$

Resumidamente, o emCGA usa os blocos lógicos descritos na Fig. 1: gerador de números pseudo randômicos (RNG-*Random Number Generator*), memória de probabilidade de mutação (MPM, *Mutation Probility Memory*) e máquina de estados finitos (FMS – *Finite State Machine*).

A implementação em *hardware* do emCGA apresenta algumas particularidades associadas a esta classe de projeto. Por exemplo, surge a possibilidade de explorar a execução paralela de alguns blocos funcionais. No emCGA-HW, como cada elemento do vetor de probabilidades é independente um

do outro, é possível gerar e atualizar os estados com blocos paralelos.

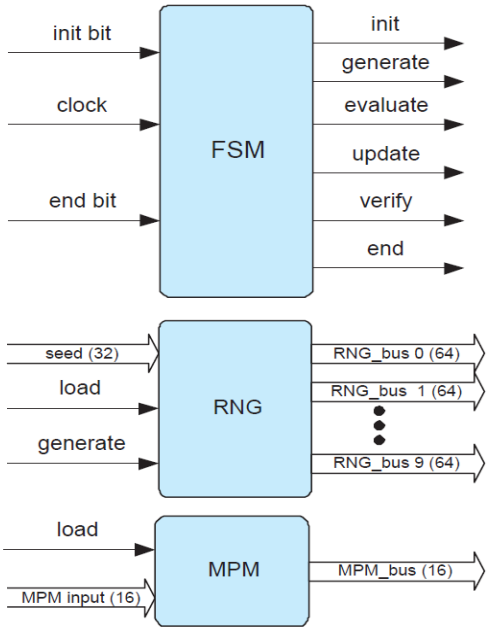

Fig. 1. Diagrama de blocos do emCGA-HW representado a máquina de estados (FSM), o gerador de números pseudorandomicos e mémoria de probabilidade de mutação.

A representação de cada um dos elementos do vetor de probabilidades é feita utilizando-se um número inteiro de $\log_2(n)$. Uma população com 256 indivíduos, um cromossomo de 10 bits e uma taxa de mutação de 16 bits são adotados. O problema utilizado calcula o Fitness com precisão de 6 bits. A probabilidade 0 ou 1 é representada pelos valores 0 ou 255, respectivamente, evitando-se as operações em ponto flutuante em *hardware*, muito custosas em termos de recursos.

As Fig. 2 e Fig. 3 complementam o diagrama de blocos da Fig. 1. O emCGA-HW é desenvolvido utilizando o kit Altera UP3, o qual permite a utilização de uma interface para comunicação com um computador. Através desta interface, o usuário carrega a semente do bloco RNG e a taxa de mutação associada ao bloco MPM. Além disto, a interface com o usuário permite iniciar a execução do algoritmo através de um bit de controle (*Init Bit*). Esta interface com o usuário foi desenvolvida em ANSI C e Matlab GUI (*Graphical User Interface*).

Durante o estado de inicialização, é possível carregar os parâmetros de configuração do emCGA-HW, inicializar os elementos do vetor de probabilidades com a probabilidade inicial (0,5) e gerar individuo elite inicial. Os parâmetros de configuração são: tamanho da população, comprimento cromossomo, taxa de mutação e a semente utilizada pelo RGN (sinal "load"). Uma vez iniciado, o algoritmo entra num ciclo: geração, avaliação, atualização e verificação até a convergência da população, isto é, até que todos os elementos do vetor de probabilidades alcançarem as probabilidades 0 ou 1. Durante o estado de geração, um dos indivíduos do torneio futuro é obtido utilizando o operador de mutação. No estado de avaliação, o indivíduo vencedor (elite) é definido de acordo com a função de Fitness. O estado de atualização é responsável pela atualização do vetor de probabilidades na direção do indivíduo vencedor, obtido no estado anterior. Se após isso a atualização do vetor converge, o algoritmo irá parar (vai para o estado final, setando o bit "end bit"). Caso contrário, um novo indivíduo irá ser gerado, ou seja, retorna ao estado de geração.

A Fig. 3 descreve o funcionamento paralelo do de emCGA: 10 blocos BG (*Bit Generator*) são usados para a geração de um novo cromossomo. A avaliação deste novo cromossomo é feita pelo bloco FEV (*Fitness Evaluator*), gerando como saída uma palavra de 6 bits que será comparada com a palavra armazenada no bloco EFM. Durante o estado de inicialização, o bloco EFM é carregado com o pior valor de *fitness* para o problema. Estas entradas permitem ao bloco CMP gerar o sinal "result", o qual será usado por todos os blocos BG no estado de atualização do cromossomo. O bloco MO (*memory output*) contém os bits de saída de BG para processamento subsequente. Finalmente, o CV bloco (C*onvergence Verify*) verifica a convergência do vetor de probabilidades, indicando eventualmente o fim de execução do algoritmo.

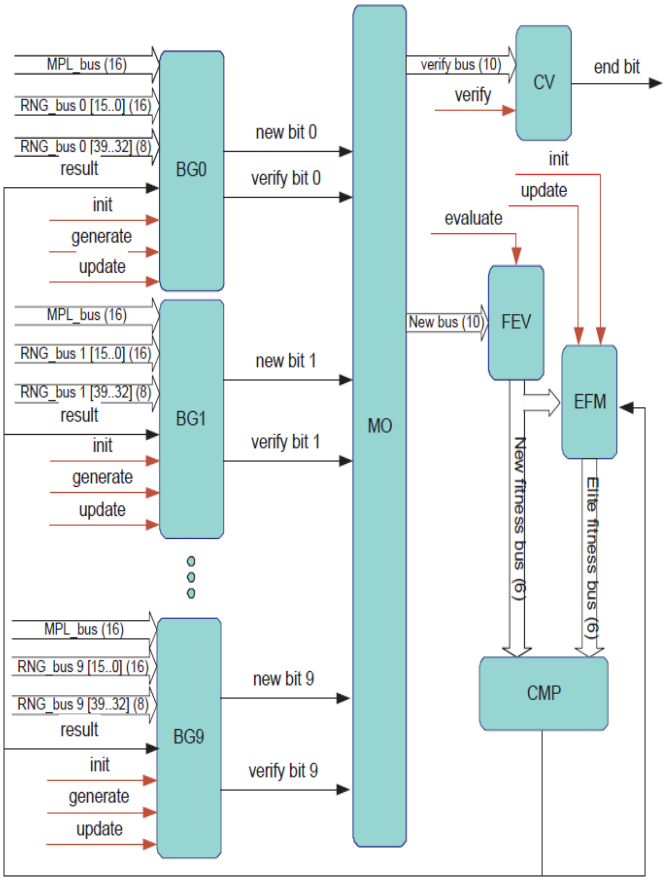

Fig. 2. Diagrama de blocos representado a operação paralela em *hardware* do emCGA.

O detalhamento do bloco BG, apresentado na Fig.3, mostra como é gerado um novo indivíduo e é feita atualização do

vetor de probabilidades. Os blocos CMP (*Comparator*) são comparadores de números inteiros e são associoados com o operador XOR (ou exclusivo) para gerar um bit do novo cromossomo. É importante observar que o operador de mutação é inserido nos blocos BG através do bloco PRM0. No bloco BM, através do sinal "result", a técnica do elitismo é aplicada: o melhor indivíduo é mantido ao longo das gerações.

Algumas observações sobre o emCGA-HW:

- O número de blocos BG é diretamente associado com o tamanho do cromossomo. O número de blocos BG não depende do tamanho da população e sim da natureza do problema (maximização ou minimização).
- O parametro de mutação pode utilizar até 32 bits, porém para o problema apresentado, 16 bits são suficientes.
- O barramento de cada elemento do vetor de probabilidades é relacionado com o tamanho da população. Considera-se uma população de 256 indivíduos, ou seja, barramentos de 8 bits.
- Os barramentos "New fitness" e "Elite fitness" são dependentes do problema. No caso, para o problema OneMax de 10 bits, o valor de fitness será representado entre 0 e 10 (decimal), sendo representado por 4 bits.
- O bloco CM associoado com os blocos FEV e EFM são também dependentes da natureza do problema.

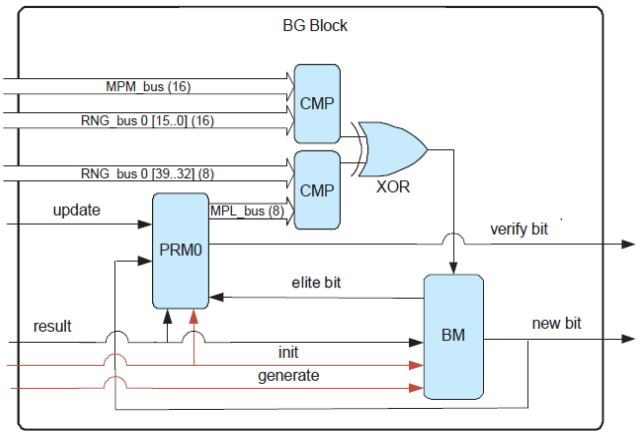

Fig. 3. Detalhamento do bloco de geração de bits os cromossomos BG (*Bit Generator*).

IV. Experimentos e Resultados

Os resultados do emCGA-HW são divididos em 3 experimentos baseados no problema Discrete1, em 2 ou 8 dimensões, usando variáveis de 8 bits.

A função *Discrete1* é definida na equação [4]. A representação numérica das variáveis pertence aos números inteiros.

$$\min\ f_c(x) = \sum_{i=1}^{n} |x_i| \qquad (4)$$

Onde *n* é a dimensionalidade do problema.

A síntese do emCGA-HW foi feita em um dispositivo Altera Cyclone II EP1C6Q240C8, operando com o clock de 48 MHz. O projeto foi desenvolvido na ferramenta Altera Quartus II, utilizando linguagem VHDL. Para fins de comparação com o *software*, foram desenvolvidos algoritmos equivalentes em um PC 2.8GHz, usando ANSI C e Matlab.

A. *Similaridade entre emCGA e emCGA-HW*

A implementação do emCGA-HW deve ter o mesmo resultado que a implementação equivalente em *software*, se ambos os algoritmos têm os mesmos parâmetros de entrada e o mesmo problema a ser tratado. O primeiro experimento usa parâmetros idênticos para as duas abordagens: características do problema, o tamanho da população, taxa de mutação, semente do gerador de números pseudo-aleatórios e tamanho do cromossomo. Para este experimento, a dimensionalidade do problema Discrete1 foi definida como 2 e as variáveis foram definidas como números inteiros de 8 bits e sem sinal. Para estes parâmetros, o tamanho do cromossomo foi de 16 bits e, como consequência, a taxa de mutação foi de 6,25%, isto é, o inverso do tamanho do cromossomo. O tamanho da população foi definido para 256 indivíduos.

Os indivíduos novos e indivíduos elite foram salvos (para posterior comparação) em cada geração, nas abordagens de *software* e de *hardware*. Os indivíduos assim armazenados são comparados usando uma função de similaridade. À medida que cada indivíduo é representado por duas variáveis inteiro sem sinal, denominado x e y, a similaridade entre estes indivíduos é definida pela distância Euclidiana, de acordo com a equação (5). Ao longo do experimento, esta distância manteve-se sempre igual zero, demonstrando que a evolução em *software* e a evolução em *hardware* foram iguais.

$$d(I_{HW}, I_{SW}) = d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \qquad (5)$$

Onde:

I_{hw} – indivíduo gerado pela solução de *hardware*.

I_{sw} – indivíduo gerado para solução de *software*.

x,y – variáveis do cromossomo individual.

d – distância Euclidiana.

B. *Demanda por recursos de hardware do emCGA-HW*

O segundo experimento tem como objetivo comparar a demanda de recursos de *hardware* do emCGA-HW com a demanda de recursos de *hardware* do CGA descrito por Aporntewan e Chongstitvatana [11], ambos usando o problema Discrete1.

Em ambos os algoritmos, foi usada dimensionalidade do problema igual a 8 e variáveis de 8 bits, inteiras e sem sinal. Desta forma, o comprimento do cromossomo foi de 64 bits e a probabilidade de mutação foi de 1,6%, o inverso do tamanho

do cromossomo. O tamanho da população foi definido como 256 indivíduos.

Ambos os algoritmos foram desenvolvidos e executados na mesma FPGA. O emCGA demandou 4531 elementos lógicos, enquanto a CGA comparado demandou 4212 elementos lógicos. É possível observar que o elitismo com extensão mutação implementado emCGA não aumentou de forma significativa a demanda de recursos de *hardware*.

C. *Comparação de desempenho*

O objetivo deste experimento foi comparar o desempenho do emCGA e do emCGA-HW. O ganho de desempenho foi calculado pela relação entre o tempo de execução do emCGA e o tempo de execução do emCGA-HW. O ganho de desempenho foi determinado pela razão tempo de execução em *software* pelo tempo de execução em *hardware*, segundo a equação (6).

$$g = \frac{t_{sw}}{t_{hw}} \tag{6}$$

Onde:

g: ganho de desempenho do emCGA em *hardware* com relação ao *software*.

tsw: tempo de execução do emCGA em *software*.

thw: tempo de execução do emCGA em *hardware*.

Neste experimento foram utilizados parâmetros idênticos de tamanho da população, taxa de mutação, semente do gerador de números pseudo-aleatórios e tamanho do cromossomo para as abordagens de *software* e *hardware*. Os valores dos parâmetros utilizados foram os mesmos do primeiro experimento. A Tabela 1 apresenta o tempo de execução total, o Fitness médio, o número médio de avaliações de Fitness para 500 rodadas independentes, bem como o tempo de execução médio para cada rodada.

TABELA 1 RESULTADOS OBTIDOS NOS EXPERIMENTOS

	Fitness médio	**Avaliações de Fitness**	**Tempo total (ms)**	**Tempo médio (us)**
emCGA	0.82	3300	17781	35522
emCGA-HW	0.82	3300	173	206

A aceleração em termos de tempo de execução foi de 173 vezes.

V. CONCLUSÃO

Muitos dos problemas encontrados em engenharia representam grandes desafios computacionais. Mesmo sendo exitosos em reduzir a complexidade computacional através da introdução de restrições e simplificações, frequentemente, alguns problemas exigem um tempo de processamento inaceitável ou demandam recursos computacionais proibitivos. A utilização de abordagens baseadas em *hardware*, como a proposta neste trabalho, contorna algumas das deficiências encontradas em computação convencional.

O objetivo deste trabalho é validar uma nova versão de um algoritmo compacto genético implementado em *hardware*. O algoritmo desenvolvido no emCGA-HW baseou-se em um trabalho prévio dos autores, no qual o emCGA foi validando totalmente em *software* [14]. A versão do emCGA-HW apresenta a mesma geração de novos indivíduos e indivíduos elite observados na versão em *software*, considerando os mesmos parâmetros de entrada e o mesmo problema objetivo.

Por outro lado, em comparação com trabalho similar em *hardware* proposto por Aporntewan e Chongstitvatana [11], observou-se um aumento de 7,6% dos recursos de *hardware*. Estes recursos adicionais não são significativos, particularmente em função das vantagens da adição das características de elitismo e mutação feitas no emCGA-HW. O pequeno aumento observado deve-se ao bloco de mutação (demanda comparadores de números inteiros), ao bloco elitismo (demanda uma memória para armazenar seu cromossomo) e ao bloco RNG, pois o emCGA requer o dobro de números pseudo-aleatórios por geração. Contudo, neste aspecto, o emCGA apresenta uma vantagem: a utilização de somente um avaliador de *fitness* (FEV). Assim, em aplicações onde a avaliação de *fitness* demande muitos recursos de *hardware*, a aplicação completa com o emCGA pode ser ainda mais compacta que a do CGA comparado.

Um resultado importante deste trabalho é a velocidade de processamento do emCGA-HW. Utilizando-se os mesmos parâmetros de entrada, o mesmo problema objetivo e comparando-se o emCGA (sendo executado em um PC padrão) com o emCGA-HW, observa-se uma redução do tempo total de processamento de 173 vezes. Isto se deve ao processamento paralelo observado nos diferentes blocos do emCGA-HW, bem como ao reduzido tempo de execução destes blocos. No entanto, essa comparação deve ser interpretada com cautela. De fato, as arquiteturas de ambos os sistemas são radicalmente diferentes, não só em como os dados de entrada e de saída são tratados, mas, principalmente, como os dados são processados. Por conseguinte, o processamento de alta velocidade de operações em *hardware* e o paralelismo (geralmente em mais de um nível) permite que, de modo geral, sistemas construídos em *hardware* alcancem um desempenho claramente inatingível por sistemas baseados em *software* comuns, rodando em máquinas regulares baseadas em arquitetura de Von Neumann.

Os inconvenientes atuais do emCGA-HW são: a topologia é dependente do problema, a capacidade de expansão do sistema é limitada (pode ser facilmente resolvido com dispositivos mais poderosos) e as funções de *Fitness* complexas são de difícil realização.

Como trabalhos futuros propõe-se uma análise do desempenho do emCGA-HW para diferentes problemas, inclusive verificando-se o comportamento do emCGA-HW quanto a escalabilidade. Propõe-se ainda uma melhor abordagem para geração de funções *Fitness* complexas e para utilização de variáveis em ponto flutuante.

REFERENCIAS

[1] E.P. Ferlin, H.S. Lopes, C.R.E. Lima and E. Chichaczewski, "Recongurable parallel architecture for genetic algorithms: application to the synthesis of digital circuits", Lect.Notes Comput. Sc. 4419, pp. 326-336, 2007.

[2] G. Harik, F. Lobo and D.E. Goldberg, "The compact genetic algorithm", IEEE T. Evolut. Comput. pp. 287-297, 1999.

[3] G. Harik, E. Cantu-paz, D.E Goldberg, B. L. Miller. "The gambler's ruin problem, Genetic Algorithms, and the sizing of populations", IEEE Transactions on Evolutionary Computation, v. 7, pp. 231–253, 1999.

[4] S. Scott and A. Seth, "HGA: a hardware-based genetic algorithm", Proc. ACM/SIGDA, Third Int. Symp. on Field-Programmable Gate Arrays, pp. 53-59, 1995.

[5] P. Graham and B. Nelson, "A hardware genetic algorithm for the traveling salesman problem on SPLASH 2", Proc. 5th Int. Workshop on Field Programmable Logic and Applications, pp. 352-361.1995.

[6] I.M. Bland and G.M. Megson, "The systolic array genetic algorithm, systolic arrays as a recongurable design methodology", Proc. FPGAs for Custom Computing Machines. Pp. 260-261, 1998.

[7] B. Shackleford, E. Okushi, M. Yasuda, H. Koizumi, K. Seo, T. Iwamoto and H. Yasuura, "An FPGA-based genetic algorithm machine", Proc. ACM/SIGDA Eighth Int. Symp. on Field-Programmable Gate Arrays, pp. 218, 2000.

[8] B. Shackleford, G. Snider, R.J. Carter, E. Okushi, M. Yasuda, K. Seo and H. Yasuura, "A high-performance, pipelined, FPGA-based genetic algorithm machine", Genet.Program. Evolvable Mach.. pp. 33-60, 2001.

[9] F. Saenz, A. Ibarra, J. Lanchares and J.I. Hidalgo, "Pipelined genetic architecture with fitness on the fly", Proc. Euromicro Symp. on Digital Systems Design DSD2001, pp. 382-385, 2001.

[10] N. Kavvadias, V. Giannakopoulou, S. Nikolaidis, "Development of customized processor architecture for accelerating genetic algorithms", Microprocess. Microsy. 31:5, pp.347-359, 2007.

[11] C. Aporntewan and P. Chongstitvatana, "A hardware implementation of the compact genetic algorithm", Proc. 2001 IEEE Congress Evolutionary Computation 1, pp. 624-629, 2001.

[12] J. Gallagher, S. Vigraham and G. Kramer, "A family of compact genetic algorithms for intrinsic evolvable hardware", IEEE T. Evolut. Comput. 8:2 , pp. 111-126, 2004.

[13] Y. Jewajinda and P. Chongstivatana, "A cooperative approach to compact genetic algorithm for evolvable hardware", Proc. of IEEE Congr. on Evolutionary Computation , pp. 2779-2786, 2006.

[14] R. R. da Silva, H. S. Lopes and C. R. Erig Lima, "A new mutation operator for the elitism-based compact genetic algorithm", Proc. of Adaptative and Natural Computing Algorithms - LNCS 49, pp. 159-166, 2007.