Uma implementação do algoritmo LCS em FPGA usando High-Level Synthesis

Carlos A. C. Jorge[1], **Alexandre S. Nery**[2], **Alba C. M. A. de Melo**[1]

[1]Universidade de Brasilia - Departamento de Ciência da Computação - Campus Darcy Ribeiro – CIC/EST

[2]Universidade de Brasilia - Departamento de Engenharia Elétrica - Campus Darcy Ribeiro – Faculdade de Tecnologia - ENE

Abstract. *This paper presents an implementation of the Longest Common Subsequence (LCS) algorithm for comparing two biological sequences using High Level Synthesis (HLS) for FPGAs. Results were obtained with a CPU Intel Core® i7-3770 CPU and a FPGA Xilinx® ADM-PCIE-KU3 that has a Xilinx Kintex® UltraScale XCKU060-2. Our experiments showed that the CPU implementation consumed 6.8x more energy to execute compared to FPGA.*

Resumo. *Este trabalho apresenta uma implementação do algoritmo* Longest Common Subsequence (LCS) *para comparação de duas sequências biológicas utilizando linguagem de alto nível* High Level Synthesis (HLS) *para FPGAs. Foram comparados resultados entre a execução em uma CPU Intel Core i7-3770 e uma FPGA Xilinx® ADM-PCIE-KU3 que possui uma Xilinx Kintex® UltraScale XCKU060-2. Os resultados mostraram que a implementação em CPU consumiu 6,8x mais energia em relação à FPGA.*

1. Introdução

A descoberta de padrões em sequências é um dos problemas mais desafiadores em biologia molecular e ciência da computação. Dado um conjunto de sequências, deve-se encontrar o padrão que ocorre com maior frequência. Em um processo de busca exata, a busca por um padrão de *m* letras pode ser resolvida por uma simples enumeração de todos os padrões de m letras que aparecem nas sequências. No entanto, quando se trabalha com sequências, realiza-se a busca aproximada pois os padrões incluem mutações, inserções ou remoções de nucleotídeos [Bucak and Uslan 2011].

O alinhamento de sequências expõe claramente os padrões que ocorrem com maior frequência, sendo útil para descobrir informação funcional, estrutural e evolucionária em sequências biológicas. Para tanto, é necessário descobrir o alinhamento ótimo, que maximiza a similaridade entre as sequências. Sequências muito parecidas (similares) provavelmente têm a mesma função e, se forem de organismos diferentes, são definidas como homólogas caso tenha existido uma sequência que seja ancestral de ambas [Mount 2001]. A similaridade de sequências pode ser um indício de várias possíveis relações de ancestralidade, inclusive a ausência de uma origem comum [Arslan 2004].

Na comparação de duas sequências biológicas calculam-se, a partir de métodos computacionais, métricas que ajudam a identificar o seu grau de relacionamento. Uma dessas métricas é o *escore*, que é atribuído a um alinhamento. Um alinhamento é definido como um pareamento, resíduo a resíduo, das sequências. Um par de resíduos das

duas sequências pode ser definido como *match*, quando os pares são iguais, *mismatch*, quando os pares são distintos ou um *gap*, quando o resíduo de uma sequência está alinhado com uma lacuna [Mount 2001]. Dentre os diversos algoritmos de comparação de sequências biológicas existentes na literatura, o algoritmo *Longest Common Subsequence (LCS)* [Wagner and Fischer 1974] é um dos mais utilizados.

A eficiência do alinhamento é obtida através de diversas técnicas, utilizadas para editar e calcular o *escore* de similaridade. As operações de edição consistem em inserir, remover e substituir. Para se calcular o *escore* de similaridade, existem diversas abordagens, porém, geralmente, os métodos de alinhamentos visam minimizar o número de *gaps* e *mismatches* penalizando-os no cálculo do *escore* final [Setubal and Meidanis 1997]. Se duas sequências possuem o mesmo ancestral, espera-se que elas possuam muitos símbolos em comum. Assim, o alinhamento busca parear os símbolos das sequências analisadas.

Apesar de sua longa história, a pesquisa em alinhamento de sequências continua a florescer. O alinhamento de sequências na biologia computacional moderna é a base de muitos estudos de bioinformática e os avanços na metodologia de alinhamento podem conferir benefícios abrangentes em uma ampla variedade de domínios de aplicação. Embora muitas dessas abordagens dependam dos mesmos princípios básicos, os detalhes das implementações podem ter grandes efeitos sobre o desempenho, tanto em termos de precisão quanto de velocidade.

Os algoritmos que produzem resultados ótimos são executados em tempo quadrático ($O(n^2)$) onde n é o tamanho das sequências. Por essa razão, seu tempo de execução é muito grande caso as sequências comparadas sejam longas. Logo, arquiteturas paralelas dedicadas em hardware (como *Field Programmable Gate Arrays (FPGAs)*) podem ser empregadas para acelerar a execução de algoritmos de alinhamento de sequência. Tais sistemas de hardware são projetados usando Linguagens Descritivas de Hardware (HDL), como Verilog, System Verilog e VHDL, que são consideradas como linguagens de baixo nível. Como o nome sugere, pode-se descrever o hardware (sistemas digitais) usando HDLs. Mas projetar hardware em FPGAs usando HDLs requer conhecimento em eletrônica digital, além de ser mais custoso em relação ao tempo de implementação e também em relação ao custo de projeto.

Síntese em Alto Nível, ou *High Level Synthesis (HLS)*, é o processo que interpreta um sistema descrito funcionalmente em uma linguagem de alto nível (normalmente C, SystemC, C++ ou Matlab), e que produz uma arquitetura RTL (*Register Transfer Level*) correspondente para implementação em um dispositivo alvo (*e.g.* FPGA). Tal arquitetura RTL é especificada pela síntese de alto nível por meio de uma linguagem de descrição de hardware como VHDL ou Verilog. O HLS foi introduzido para facilitar a especificação e a implementação de arquiteturas RTL a partir de códigos em alto nível [Xilinx Corporation 2019], reduzindo significativamente o tempo de projeto de sistemas complexos. Ele também facilita o projeto destes sistemas quando se trata de alcançar um determinado modelo exigido pelo hardware sem se preocupar muito com os componentes eletrônicos do circuito, o que é especialmente importante para desenvolvedores de software que, em geral, não possuem formação adequada para especificar sistemas digitais. Por fim, o HLS também contribui para a portabilidade de código, uma vez que o sistema descrito em linguagem de alto nível pode ser compilado e executado tanto em arquiteturas Von-Neumann como também pode ser recompilado pelo HLS para implementação RTL

em outros dispositivos alvo.

Neste sentido, no presente trabalho, propõe-se uma implementação do algoritmo *Longest Common Subsequence (LCS)* em HLS para comparação de duas sequências de DNA de tamanho médio (até 50000 resíduos), analisando os recursos utilizados, tempo de execução e consumo de energia do *design* proposto. O objetivo da nossa proposta é fornecer uma implementação eficiente, onde o consumo de energia seja reduzido. Para tanto, utilizamos as *Block RAMs* da FPGA para armazenar dados (ao invés da memória DDR presente na FPGA) e preenchemos a matriz diagonal por diagonal. Os resultados experimentais mostram que o consumo de energia da nossa solução é significativamente menor que aquele da solução em CPU.

2. Longest Common Subsequence

A essência do problema *Longest Common Subsequence (LCS)* entre os elementos de um conjunto de sequências de símbolos de qualquer natureza é determinar o grau de similaridade existente entre as sequências que fazem parte do conjunto.

Define-se como sequência o encadeamento ordenado de objetos naturais representados por símbolos, comumente caracteres alfa-numéricos. Como exemplo, as letras A, C, G e T representam os quatro nucleotídeos de uma cadeia de DNA (i.e. as bases adenina, citosina, guanina e timina), e sequências como ACCCGGTTT representam uma sequência de DNA. Uma subsequência qualquer pode ser obtida extraindo-se zero ou mais caracteres da sequência original, mantendo seu ordenamento. Por exemplo, as sequências ACCCGGTTT, ACC, AGTT e ACGT são todas subsequências de ACCCGGTTT. Por extensão, a sequência vazia é subsequência de todas as sequências da natureza, e toda sequência é subsequência de si própria [Wagner and Fischer 1974].

Define-se subsequência comum a duas sequências como uma sequência que é subsequência de ambas as sequências. Uma subsequência comum máxima é aquela que, dentre todas as subsequências comuns, tem o maior comprimento. Para efeito do estudo da LCS, podemos abstrair completamente o significado de cada símbolo em uma sequência qualquer e manter o foco somente em encontrar a subsequência máxima entre sequências de caracteres. Por esta razão, é comum definir o LCS como o problema de determinar a maior subsequência de caracteres comuns a duas sequências de caracteres, também chamadas de *strings*.

O problema da LCS é um problema com complexidade de tempo quadrática, por essa razão, este problema possui dificuldade crescente de realização prática com o crescimento do tamanho das *strings* comparadas.

Dadas duas sequências A e B de comprimento m e n, respectivamente, onde $A = A_1, A_2, ..., A_m$ e $B = B_1, B_2, ..., B_n$ e $D(i,j) = \delta(A(i), B(j)), 0 \leq i \leq |A|, 0 \leq j \leq |B|$. Definimos $D(i,j)$ como a máxima subsequência comum entre A e B. A equação de recorrência para o cálculo pode ser descrita da seguinte forma [Wagner and Fischer 1974]:

$$D(i,j) = max \begin{cases} D(i-1, j-1) + \gamma(A(i) \rightarrow B(j)) \\ D(i-1, j) + \gamma(A(i) \rightarrow \Lambda) \\ D(i, j-1) + \gamma(\Lambda \rightarrow B(j)) \end{cases} \quad (1)$$

Na Equação 1 o primeiro elemento verifica *match* ou *mismatch* entre dois caracteres das sequências, o segundo elemento verifica *gap* na primeira sequência e o terceiro verifica *gap* na segunda sequência.

3. Trabalhos Relacionados

Al Junid et al. (2010) apresentam o projeto e desenvolvimento da técnica de aceleração e otimização de alto desempenho para alinhar sequências de DNA com o algoritmo Smith-Waterman (SW) [Smith and Waterman 1981]. O artigo tem seu foco na otimização de memória e velocidade, otimizando e mapeando os dados das sequências de DNA antes do alinhamento. Essa técnica de otimização é projetada com base na técnica de compactação de dados para reduzir o número de dados transmitidos do computador para o acelerador, que neste caso é a FPGA. A técnica proposta foi projetada e implementada em uma FPGA Altera Cyclone II 2C70. O código foi escrito em HDL Verilog. Como resultado, a análise teórica, a simulação e o resultado da implementação baseados no desenvolvimento e implementação do projeto proposto no FPGA foi de um speedup de 1,75x em relação à CPU com sequências de até 1024 caracteres.

Chen et al. (2011) propõem uma arquitetura sistólica reconfigurável para o problema de alinhamento de sequências. Inicialmente, utilizam o algoritmo Needleman-Wunsch (NW) [Needleman 1970] empregando a técnica de "dividir e conquistar", e obtém um alinhamento de seqüência com a melhor pontuação. Utilizam uma FPGA Altera Cyclone II EP2C35 com programação em VHDL para sequências de DNA.

Mousavi et al. (2012) propõem um novo algoritmo baseado no método de busca construtiva de feixes. Criaram uma nova heurística, inspirada na teoria da probabilidade, destinada a domínios em que as sequências de entrada são consideradas independentes. Estruturas de dados especiais e métodos de programação dinâmica são desenvolvidos para reduzir a complexidade de tempo do algoritmo. O algoritmo proposto é comparado com o estado da arte em vários benchmarks padrão, incluindo sequências biológicas aleatórias e reais. Utilizaram como plataforma CPU Intel i7 2770 com programação JAVA com sequências de tamanho 100 e obtiveram speedup de 1,37x quando comparado com o trabalho de [Blum et al. 2009].

Ozsoy et al. (2013) descrevem uma nova técnica para otimizar o algoritmo *Longest Common Subsequence (LCS)* com várias GPUs, transformando a computação em operações *bit-wise* e executando uma etapa de pós-processamento. Utilizaram como plataforma NVIDIA M2090 Fermi com programação em CUDA e obtiveram um speedup 8,3x em relação à CPU com sequências de DNA com tamanho até 4000.

Cinti et al. (2018) apresentam um novo algoritmo para correspondência de cadeia aproximada on-line (OASM) capaz de filtrar *shadow hits* em tempo real, de acordo com regras de prioridade de finalidade geral que atribuem prioridades a ocorrências sobrepostas. Uma implementação em FPGA do OASM é proposta e comparada com uma versão de software serial. Mesmo quando implementado em FPGAs de nível básico, o procedimento proposto pode alcançar um alto grau de paralelismo e desempenho superior no tempo em comparação com a implementação de software, ao mesmo tempo em que mantém baixo o uso de elementos lógicos. Utilizaram uma CPU Intel® i7 4700MQ e uma FPGA Altera Cyclone® IV E com programação em C++ para a CPU e VHDL para a FPGA com sequências sintéticas com tamanho de 3104.

Tabela 1. Tabela comparativa entre os trabalhos relacionados.

Artigo	Ano	Técnica	Plataforma	Programação	SpeedUp	Tamanho	Energia	Tipo
Junid et al.	2010	Compressão de Dados + SW	Altera Cyclone II 2C70	Verilog	1,75x	1024	ND	DNA
Chen et al.	2011	Dividir para Conquistar + Algoritmo Sistólico NW	Altera Cyclone II EP2C35	VHDL	ND	ND	ND	DNA
Mousavi et al.	2012	Constructive Beam Search Method	Intel i7 2770	Java	1,37x	100	ND	DNA
Ozsoy et al.	2013	LCS	NVIDIA M2090 Fermi	CUDA	8,3x	4000	ND	DNA
Cinti et al.	2018	OASM SW-OASM e HW-OASM	Intel i7 4700MQ + Altera Cyclone IV E	C++ (CPU) + VHDL (FPGA)	ND	3104	ND	ND
Alser et al.	2019	Shouji	Intel i7-3820 + Xilinx Virtex®-7 VC709	C (CPU) + Verilog (FPGA)	1,07x	250	ND	DNA

E por último, temos Alser et al. (2019) apresentando um algoritmo chamado Shouji, um filtro de realinhamento altamente paralelo e altamente preciso, que usa uma abordagem de janela de busca deslizante para identificar rapidamente sequências diferentes, sem a necessidade de algoritmos de alinhamento computacionalmente caros. Shouji baseia-se em um novo algoritmo de filtragem que reduz a necessidade de alinhamento ótimo excluindo rapidamente sequências diferentes do cálculo de alinhamento ótimo e faz uso melhor da arquitetura de paralelismo dos FPGAs modernos para acelerar este novo algoritmo de filtragem. Utilizaram CPU Intel® i7-3820 e FPGA Xilinx Virtex®-7 VC709 com programação em C para CPU e Verilog para a FPGA. O tamanho das maiores sequências de testes é de 250 pares de bases e obtiveram como resultado um speedup de 1,07x em relação à CPU.

Pela Tabela 1 pode-se notar que todos utilizaram sequências pequenas de DNA (até 4000 caracteres). Além disso, nenhum dos trabalhos relacionados fez-se medição de consumo de energia. Ainda pode-se notar que a grande maioria dos trabalhos que utilizaram FPGAs, utilizaram linguagem de programação HDL.

4. A Arquitetura Proposta

Neste trabalho, utiliza-se a ferramenta *Vivado® HLS* desenvolvida pela *Xilinx* com o objetivo de fornecer suporte para o desenvolvimento de circuitos em FPGAs Xilinx. A ferramenta permite que a especificação funcional de um sistema em alto nível (C/C++) seja usada para a produção de um circuito em nível *Register Transfer Level (RTL)*, sem a necessidade de fazê-lo manualmente [Xilinx 2016]. Além disso, ela fornece algumas di-

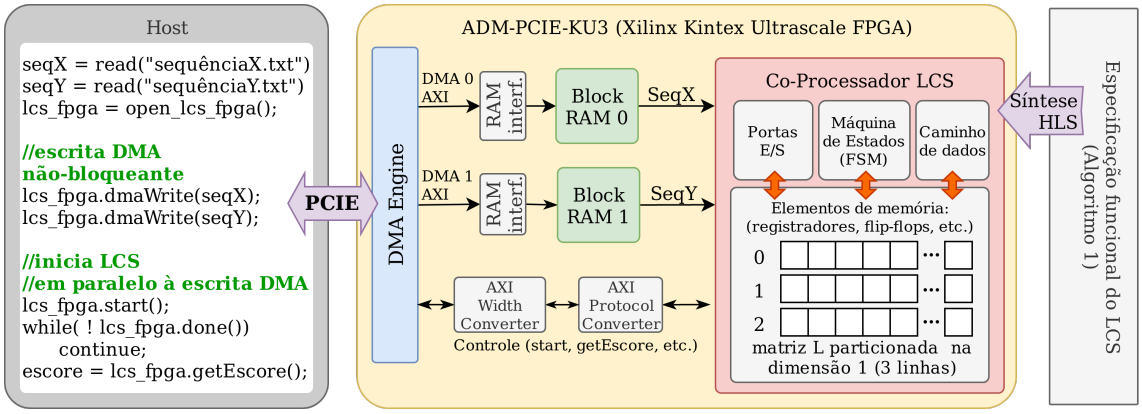

Figura 1. Arquitetura da implementação proposta em FPGA.

retivas de compilação para otimização da arquitetura RTL produzida, como por exemplo: *loop unrolling*, *pipeline*, *array partition*, etc.

Na Figura 1 observa-se (da direita para esquerda) que a síntese HLS traduz a especificação funcional do LCS em uma arquitetura RTL correspondente com quatro principais componentes: Portas de E/S, Máquina de Estados, Caminho de Dados e Elementos de Memória, que se comunicam através de barramentos ou linhas de dados específicas. Tais componentes são criados a partir da análise da especificação funcional e da extração do seu fluxo de controle e operações artiméticas/lógicas. Este processo não é exclusivo do LCS e, em geral, ocorre para toda e qualquer especificação funcional dada ao HLS. A Figura 1 também apresenta a arquitetura geral da solução proposta mostrando a interação entre uma CPU (Host) e a FPGA, via PCI-Express. A CPU transfere para a FPGA (*Block RAMs*) as duas sequências a serem comparadas e o circuito gerado pelo HLS (Co-Processador LCS) já na FPGA calcula a matriz de programação dinâmica (usando apenas 3 linhas) e retorna para a CPU o escore máximo entre as duas sequências.

A especificação do algoritmo implementado no Co-Processador LCS é uma adaptação do algoritmo LCS tradicional, calculando a matriz de programação dinâmica entre duas sequências na diagonal, e usando uma matriz de apenas 3 linhas. A Figura 2 exemplifica o fluxo de trabalho do algoritmo para duas sequências de tamanho 5 (ACGTA e CCGTT). O cálculo da matriz de programação dinâmica é feito na diagonal utilizando-se de três vetores que armazenam os resultados do cálculo e à medida que progride dentro da matriz, o vetor com a diagonal mais antiga é sobreposto pelo cálculo da diagonal corrente.

O trecho de código principal que especifica o LCS usando Vivado HLS é apresentado no Algoritmo 1. Inicialmente é definida uma matriz estática de três linhas L[3][MAX_N], onde serão armazenados os resultados intermediários a medida que a execução do algoritmo LCS progride. O tipo *u16* representa dados inteiros de 16-bits, sem sinal. Observe que a linha 7 especifica uma otimização de particionamento de array, indicando que a matriz L deve ser particionada em 3 blocos na dimensão 1, ou seja, 3 vetores. Isto é importante para que o sintetizador HLS possa instanciar elementos de memória com endereços de leitura/escrita separados para cada linha da matriz, permitindo o acesso em paralelo a qualquer uma das 3 linhas. Do contrário, a matriz inteira poderia ser mapeada em um conjunto de BlockRAMs com um único endereço de leitura/escrita,

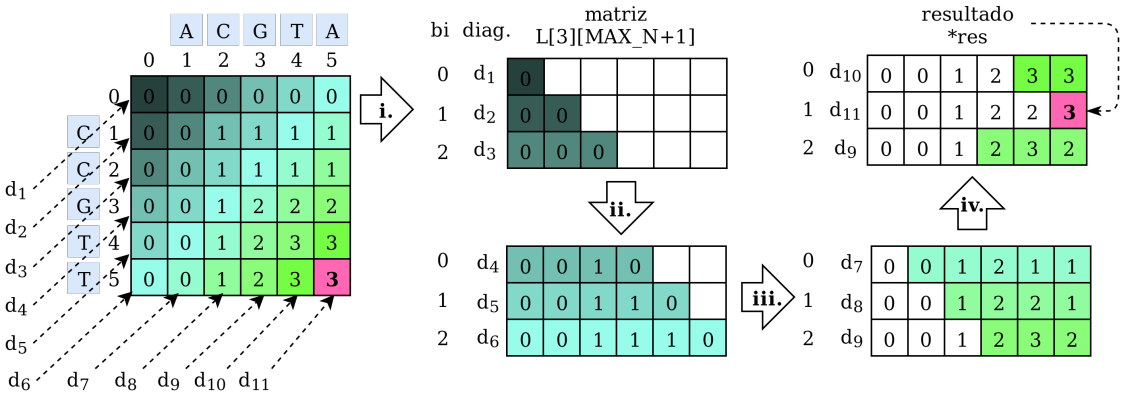

Figura 2. Fluxo do cálculo da matriz de programação dinâmica.

o que pode gerar um gargalo no sistema.

No Algoritmo 1 observa-se a síntese de interfaces na linha 6, que especifica que os argumentos m,n e $*res$ obedecerão ao protocolo *AXI-Lite slave*, e que ainda serão agrupados em uma porta Ctrl (Controle). Logo, cada um destes argumentos serão registradores que poderão ser acessado por mapeamento de memória através de uma máquina host. Em seguida, a variável *bi* indica o índice do vetor (linha da matriz) que representa a diagonal a ser acessada e modificada. O tipo *u2* representa dados inteiros de 2-bits, sem sinal, pois a variável *u2* só pode indexar uma das 3 diagonais.

Algoritmo 1. Trecho da especificação funcional do LCS usando Vivado HLS.

```
1  #define MAX_N 50000
2  static u16 L[3][MAX_N + 1]; //matriz de apenas 3 linhas para otimizar uso de BRAM
3
4  void lcs(volatile uchar X[MAX_N],volatile uchar Y[MAX_N],int m,int n,int *res) {
5
6    #pragma HLS INTERFACE s_axilite port=m,n,res,return bundle=Ctrl
7    #pragma HLS ARRAY_PARTITION variable=L block factor=3 dim=1
8
9    u2 bi = 0;
10   for (int line=1; line <= ((m + 1) + (n + 1) - 1); line++) {
11     int start_col = max(0, line - (m + 1));
12     int count = min3(line, ((n + 1) - start_col), (m + 1));
13
14     if (bi > 2) bi = 0; //alterna entre uma das 3 linhas de L
15
16     for (int k = 0; k < count; k++) { #pragma HLS PIPELINE
17       int i = (min((m + 1), line) - k - 1);
18       int j = (start_col + k);
19
20       if (i == 0 || j == 0) { L[bi][j] = 0; }
21       else if (X[i - 1] == Y[j - 1]) {
22         if (bi == 0)       L[bi][j] = L[1][j - 1] + 1;
23         else if (bi == 1)  L[bi][j] = L[2][j - 1] + 1;
24         else               L[bi][j] = L[0][j - 1] + 1;
25       } else {
26         if (bi == 0)       L[bi][j] = max(L[2][j - 1], L[2][j]);
27         else if (bi == 1)  L[bi][j] = max(L[0][j - 1], L[0][j]);
28         else               L[bi][j] = max(L[1][j - 1], L[1][j]);
29       }
30     }
31     if (start_col < m) bi ++; //incrementa se existirem diagonais a processar
32   }
33   *res = L[bi][n];
34 }
```

A partir da linha 10 temos as iterações no cálculo da matriz de programação dinâmica por diagonal (indicada pela variável $line$) e armazenando os valores na matriz L particionada em três vetores das diagonais. Note que o particionamento é automático,

ou seja, a forma de acesso à matriz L não precisa ser modificada para indicar o acesso a cada um dos vetores em particular. Na linha 16 do algoritmo também foi inserida uma otimização do HLS conhecida como PIPELINE. O pragma PIPELINE gera na arquitetura RTL um pipeline com um determinado intervalo de iniciação para uma função ou emphloop, permitindo a execução simultânea de operações em diferentes estágios do pipeline, diminuindo o tempo de execução entre cada uma das iterações. Durante a síntese, o intervalo de iniciação (*Initiation Interval*, II) é configurado em 1, indicando que a cada ciclo uma nova operação da função ou emphloop pode ser iniciada. Caso este valor inicial seja proibitivo para produção do circuito dentro das especificações de tempo fornecidas pelo desenvolvedor, o valor é incrementado até atingir um intervalo de iniciação aceitável para o circuito proposto operar conforme as especificações desejadas. Por fim, o resultado (escore máximo identificado entre as duas sequências de entrada) estará no último elemento da matriz L de programação dinâmica indicado pelas variáveis *bi* e *n*.

5. Experimentos

O ambiente de testes utilizado para esses experimentos foi uma máquina portando uma CPU Intel Core i7-3770 e uma placa Alpha-Data ADM-PCIE-KU3 que possui uma FPGA Xilinx Kintex® UltraScale XCKU060. O circuito gerado pela implementação possui frequência de 250MHz e utiliza os recursos da placa como mostra a Tabela 2.

Tabela 2. Tabela de recursos utilizados pelo circuito na FPGA.

Tipo	Usado	Disponível	Utilizado(%)
LUTS	26129	331680	7,88
Registradores	42208	663360	6,36
Blockram	223	1080	20,65
DSPs	64	2760	2,32
IOB	53	520	10,19
IO	49	104	47,12

Os dados utilizados para os testes realizados foram as sequências de DNA com identificadores de acesso NC_024791 e KM224878 ambas obtidas pelo NCBI [NCBI 2019] e truncadas em 50000. As mesmas sequências foram utilizadas para os testes de 20K e 10K. No caso da solução proposta o intervalo de iniciação do *pipeline* foi configurado automaticamente em 4 pela ferramenta, ou seja, a cada 4 ciclos ele inicia a computação de um novo elemento da diagonal, e menos que isso não foi possível por conta das dependências de dados nos circuitos que foram gerados com intervalo de iniciação II = 1,2 e 3.

A energia elétrica *power* foi medida através dos sensores existentes na FPGA (que medem amperagem e voltagem dos dois principais *power rails* que alimentam a placa). A ferramenta *sysmon* faz parte das referências de projetos da *Alpha-Data*, montadora da placa, em que é possível visualizar a amperagem e voltagem registrados pelos sensores durante o tempo de execução do algoritmo. O tempo de transferência entre a CPU e FPGA não produziu impacto significativo e, portanto, foi desprezado.

A energia elétrica é a mesma para todos os testes, pois foi implementado um único circuito com capacidade de armazenar sequências de até 50000 caracteres. O tempo de

execução foi medido através do *host* em que contabiliza o tempo de transferência dos dados para a FPGA até o retorno do escore para o programa *host*. Para a CPU foi utilizado a ferramenta *powerstat* [Canonical 2015] que mede a energia elétrica do processador quando o algoritmo está em execução.

A Tabela 3 apresenta os resultados obtidos para cada teste realizado na FPGA. Nessa tabela, a energia (*energy*) foi calculada multiplicando-se a energia elétrica *power* pelo tempo de execução.

Como pode ser visto na Tabela 3, o tempo de execução das três comparações (10K, 20K e 50K) é menor na CPU, sendo em torno de 25% mais rápido quando comparado com o nosso projeto em FPGA. No entanto, quando consideramos a energia gasta na execução nas duas plataformas, a nossa solução em FPGA consome 15% da energia gasta em CPU. Por exemplo, temos 17,02 Joules em FPGA e 108,40 Joules em CPU para a comparação de 10K.

Tabela 3. Tabela com resultados experimentais.

	Tamanho	**Tempo (s)**	**Escore**	**Potência Elétrica (W)**	**Energia (J)**
FPGA	10K	01,60	8672	10,64	17,02
CPU	10K	01,22	8672	88,86	108,40
FPGA	20K	06,40	17141	10,64	68,09
CPU	20K	04,80	17141	90,80	435,84
FPGA	50K	40,00	41497	10,64	425,60
CPU	50K	30,00	41497	93,96	2907,90

6. Conclusão e Trabalhos Futuros

O presente artigo propôs e avaliou uma solução HLS em FPGA para comparação de sequências biológicas com o algoritmo LCS. A nossa solução utilizou *Block RAMs* para armazenar as sequências, que são acessadas através dos barramentos DMA da FPGA, otimizou as iterações do cálculo da matriz de programação dinâmica utilizando pipeline, reduziu o custo de espaço em memória utilizando somente 3 vetores para armazenar os valores calculados e diminuiu o tempo de execução calculando-se na diagonal.

Os resultados experimentais mostraram que a solução proposta em FPGA é capaz de consumir muito menos energia que a solução em CPU. Na comparação de duas sequências de 50000 caracteres, a energia gasta foi 425,60 Joules em FPGA enquanto a mesma comparação gastou 2907,90 Joules em CPU, com isso nota-se que a CPU consome 6.8x mais energia para executar a implementação em relação à FPGA. Nota-se também que o tempo de execução em FPGA aumenta cerca de 25% em relação à CPU. Com a nossa solução, o consumo de energia foi significativamente menor na FPGA.

Cabe ainda melhorias na implementação do algoritmo LCS tanto na parte do *host* quanto no *design* do HLS, como o projeto de transferência assíncrona entre o *host* e a FPGA. Ainda há espaço para melhorias no código HLS, como otimizações avançadas, tais como *HLS Interface* (melhoria nas portas de entrada e saída da FPGA), *HLS array_map*(otimização de Block RAMs utilizadas), *HLS array_partition* (otimização de leitura e escrita em memória), *HLS loop_flatten* (otimização de *loops*) e *HLS dataflow*

e stream (otimizações de instruções no pipeline) tanto no próprio código quanto nas configurações da ferramenta Vivado®, a fim de otimizar o circuito a ser gerado.

Finalmente, pretendemos fazer um estudo complementar para saber o impacto do consumo de recursos da placa, bem como se comporta o circuito gerado pela ferramenta, aumentando o tamanho das sequências a serem comparadas.

7. Agradecimentos

A presente pesquisa é parcialmente financiada pelo projeto Capes/PROCAD 183794.

Referências

Arslan, A. (2004). *Sequence Alignment*. Biyoinformatik-II.

Blum, C., Blesa, M. J., and López-Ibáñez, M. (2009). Beam search for the longest common subsequence problem. *Comput. Oper. Res.*, 36(12):3178–3186.

Bucak, I. O. and Uslan, V. (2011). Sequence alignment from the perspective of stochastic optimization: a survey. *Turkish Journal of Electrical Engineering and Computer Sciences*, 19:157–173.

Canonical (2015). *Ubuntu Manpage: powerstat - a tool to measure power consumption*. http://manpages.ubuntu.com/manpages/xenial/man8/powerstat.8.html.

Mount, D. W. (2001). *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press.

NCBI (2019). National center for biotechnology information. *National Center for Biotechnology Information*. Disponível em https://www.ncbi.nlm.nih.gov/.

Needleman, S. B.; Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*.

Setubal, J. C. and Meidanis, J. (1997). *Introduction to Computational Molecular Biology*. PWS.

Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. pages 195–197.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173.

Xilinx (2016). Vivado high-level synthesis. *Xilinx*. Disponível em https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

Xilinx Corporation (2019). Ultrafast high-level productivity design methodology guide.