HPyC-FPGA - Integração de Aceleradores em FPGA de Alto Desempenho com Python para Jupyter Notebooks*

Lucas B. da Silva[1], **Jeronimo Costa Penha**[1,2], **Dener V. Ribeiro**[1]
Alysson Silva[1], **José Augusto M. Nacif**[1], **Ricardo Ferreira**[1]

[1]Universidade Federal de Viçosa (UFV)
[2]Centro Federal de Formação Tecnológica de Minas Gerais (CEFET-MG)
{lucas.bragança, jnacif, ricardo}@ufv.br

Resumo. *O desenvolvimento de aceleradores em FPGAs (Field Programmable Gates Arrays) ainda é um desafio. Recentemente, o ambiente PYNQ da Xilinx possibilitou a integração de código Python com aceleradores em FPGA. A maioria dos exemplos está voltada para placas de prototipação utilizadas no desenvolvimento de aplicações embarcadas. Este artigo apresenta o algoritmo K-means de aprendizado de máquina não supervisionado como estudo de caso. A principal contribuição deste trabalho é o encapsulamento de 3 aceleradores acoplados com PYNQ usando o ambiente Jupyter Notebook. A avaliação foi realizada em uma máquina de alto desempenho utilizando um FPGA Alveo U55C com memória HBM (High Bandwidth Memory). Os resultados são promissores, além de mostrar as facilidades de uso do FPGA de forma encapsulada, o ganho de desempenho foi de uma a duas ordens de grandezas em comparação a um sistema com dois processadores Xeon(R) Silver 4210R com 10 núcleos cada, executando a etapa de classificação do algoritmo K-means.*

1. Introdução

Os FPGAs (*Field Programmable Gates Arrays*) são plataformas de hardware reconfigurável que oferecem alto desempenho com baixo consumo de energia. A maior vantagem do FPGA é sua flexibilidade para se adaptar ao problema, gerando um acelerador capaz de explorar as operações com paralelismo espacial e temporal. Entretanto, as aplicações precisam ser mapeadas, o que envolve a geração da configuração para o FPGA conhecido pelo termo *bitstream*, que é o equivalente ao código de máquina de uma aplicação em software. Ademais, o mapeamento ou a síntese para geração do *bitstream* pode demorar de minutos até horas, desestimulando o uso dos FPGAs. Outro ponto é a complexidade das linguagens de desenvolvimento para descrição de hardware como Verilog e VHDL, que exige conhecimento do processo e da arquitetura do FPGA. Nos últimos anos, várias alternativas com código em alto nível vêm sendo propostas para simplificar esta etapa como, por exemplo, Java [Caldeira and et al. 2018], OpenCL e o uso de diretivas de compilação com a linguagem C++.

Atualmente, os FPGAs são compostos por milhões de células reconfiguráveis e uma malha de interconexão programável. Todos esses elementos são configurados em

*Agradecimentos: FAPEMIG, CNPq (PIBIC, 213.312/2020–6, 236.290/2018–9), NVIDIA, Xilinx, Funarbe. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) - Código de Financiamento 001.

nível de *bit*. Além da flexibilidade em nível de *bit*, os FPGAs oferecem recursos heterogêneos que possuem uma granularidade maior como os DSPs (*Digital Signal Processor*) utilizados para soma/multiplicação e módulos de memória. Por exemplo, o FPGA de alto desempenho Alveo U55C da Xilinx possui 9.024 DSPs, 2.016 blocos de RAM (*Random Access Memory*) de 36Kb (totalizando 71 Mb) e 960 módulos de ultraRAM de 288Kb (totalizando 270 Mb). Além dos recursos de alto nível, os novos FPGAs possuem facilidades para conexão com memória externa. A placa Alveo U55C possui 32 bancos de memória HBM (*High Bandwidth Memory*) de 512 MB, totalizando 16 GB com uma vazão aproximada de 460 GB/s. Estes recursos tornam os novos FPGAs competitivos com outros aceleradores como as GPUs (*Graphics Processing Units*) [He et al. 2021], abrindo novas possibilidades de aplicações em ambientes de alto desempenho.

Porém, ainda existem grandes desafios para simplificar a utilização e o acesso aos FPGAs [Silva et al. 2019]. O arcabouço PYNQ [Xilinx 2022] é um projeto de código aberto da Xilinx que oferece um pacote na linguagem *Python*. O PYNQ permite acoplar um ambiente de alto nível com aceleradores implementados em plataformas de FPGA. Embora o PYNQ simplifique a integração hardware/software, ainda impõe alguns obstáculos para os programadores que não possuem familiaridade com plataformas de hardware. Desse modo, alternativas para melhorar o ciclo de desenvolvimento de projeto em hardware têm sido propostas nos últimos anos. Essas alternativas são chamadas síntese de alto nível ou HLS (*High Level Synthesis*) que permitem a descrição de aceleradores com linguagens de alto nível C/C++ por meio de diretivas de compilação. A extensão HLS [Cong and et al. 2018] da Xilinx vem possibilitando a popularização do uso de FPGAs para descrever aceleradores sem a necessidade do uso de linguagens de descrição de hardware como VHDL e Verilog. Entretanto, as ferramentas HLS ainda exigem conhecimentos de hardware por parte dos programadores para escolha das diretivas. Ademais, para problemas específicos, o uso de geradores parametrizáveis de código Verilog/VHDL pode gerar ganhos [Wang and et al. 2021, Bragança and et al. 2021]. Portanto, o programador pode simplesmente importar uma biblioteca com a funcionalidade que deseja e acoplar com seu código de alto nível. Entretanto, a maioria dos trabalhos que usam PYNQ foram validados em placas de prototipagem. Apenas alguns trabalhos recentes avaliaram placas de alto desempenho, como a interligação entre várias placas pela rede proposta em [He et al. 2021].

O trabalho proposto neste artigo, denominado *HPyC-FPGA*, busca contribuir na avaliação da integração de PYNQ com aceleradores em FPGA para algoritmos clássicos de aprendizado de máquina. Diferente dos trabalhos anteriores com PYNQ que focaram em placas de prototipagem, este trabalho investiga FPGAs de alto desempenho com memórias HBM e faz uma análise crítica do acoplamento e desempenho de aceleradores HLS e RTL (*Register Transfer Level*). O algoritmo K-means foi selecionado como estudo de caso. Para avaliar a versatilidade da abordagem, três aceleradores foram incorporados. Os dois primeiros foram codificados em HLS e proposto em Rodinia-HLS [Cong and et al. 2018]. O terceiro é um gerador parametrizado de código Verilog para K-means [Bragança and et al. 2021]. Apesar destes aceleradores terem sido desenvolvidos para interface DDR de memória externa, foi possível integrá-los com uma interface de memória HBM para validação nas placas Alveo de alto desempenho.

2. Fundamentos

2.1. K-means

O algoritmo K-means [Lloyd 1982] foi proposto há mais de seis décadas, porém se popularizou nas últimas duas décadas com a difusão do uso de técnicas de aprendizado de máquina e a grande disponibilidade de dados. O surgimento de plataformas de alto nível, como a biblioteca *Scikit-learn* e o ambiente *Jupyter notebook*, simplificou muito o uso do K-means na maioria dos conjuntos de dados.

O K-means é uma técnica de agrupamento não supervisionada que classifica as amostras em grupos. Cada grupo tem um ponto central denominado pelo termo centroide. O hiperparâmetro K determina o número de centroides ou classes para o agrupamento. Uma amostra é classificada no grupo que estiver mais próximo do centroide. Apesar de ser um problema NP-difícil gerar o agrupamento, várias heurísticas iterativas foram propostas [Lloyd 1982]. Supondo K centroides, a heurística percorre e classifica todas as amostras. Em um segundo passo, os K centroides são ajustados e o algoritmo é executado novamente até convergir. Este artigo avalia o passo de classificação do algoritmo implementada na biblioteca Rodinia em HLS [Cong and et al. 2018] e no gerador de código [Bragança and et al. 2021]. A complexidade deste passo é $O(K \times \mathrm{N} \times M)$ para M amostras com N atributos particionadas em K grupos.

2.2. PYNQ

PYNQ é o acrônimo para Python + ZYNQ, onde Python é usado como *front-end* nas placas de prototipagem da série ZYNQ Xilinx. Inicialmente o PYNQ foi proposto para o ambiente embarcado. Recentemente, a versão 2.5 incorporou placas de alto desempenho [Xilinx 2022], possibilitando que o desenvolvimento de *drivers* personalizados para acoplamento de código em Python com aceleradores em FPGAs disponíveis em *data centers*. Os *drivers* gerenciam de forma encapsulada os recursos de hardware como escrever e ler em registradores, memórias e outros componentes do FPGA.

Ferramentas como Jupyter Notebook [Rule et al. 2018] vêm ganhando cada vez mais popularidade, principalmente para ensino de aprendizado de máquinas e áreas correlatas da Inteligência Artificial [Canesche and et al. 2021]. O objetivo deste trabalho é utilizar a combinação Jupyter Notebook + PYNQ como um ambiente interativo em um navegador, mostrando como usar a computação com aceleradores em FPGAs com poucas linhas de código, abstraindo do usuário todo mecanismo complexo das ferramentas de projeto de hardware.

A Figura 1 apresenta um exemplo de código para uso do acelerador em FPGA na linguagem Python com PYNQ. Algumas linhas foram omitidas para destacar os pontos principais. Primeiro, pode-se observar que o código usando PYNQ é simples e direto. No exemplo foi utilizado um acelerador que executa uma soma vetorial. Portanto, possui dois vetores v_1 e v_2 como entradas e retorna a soma em um terceiro vetor. Além dos vetores é necessário especificar o tamanho do vetor. Na Figura 1(a), primeiro é necessário criar uma classe Python a partir da classe *pynq.DefaultIP*. Um acelerador simples requer apenas um método para realizar a computação, incluindo a alocação/transferência de memória. A classe implementa o *driver* que fará a interface com o acelerador. Essa classe é instanciada no momento que o FPGA for configurado com *bitstream* através do comando executado na Figura 1(b) linha 1. Os principais passos estão destacados na Figura 1(a), onde nas

linhas 11, 12 e 13 o método *pynq.allocate* aloca um espaço de memória contígua que será enviado para a memória global do FPGA. O próximo passo é copiar os dados para este espaço, que ainda está restrito a memória do processador, os dados só são enviados para o FPGA nas linhas 15 e 16 com a chamada do método *sync_to_device*. Finalmente, com os dados na memória do FPGA, o acelerador é chamado na linha 17 de forma síncrona. O processador aguarda o final da execução do acelerador, para então ler os dados de volta para a sua memória na linha 18 com o método *sync_from_device*. A Figura 1(c) mostra a movimentação entre as regiões de memórias.

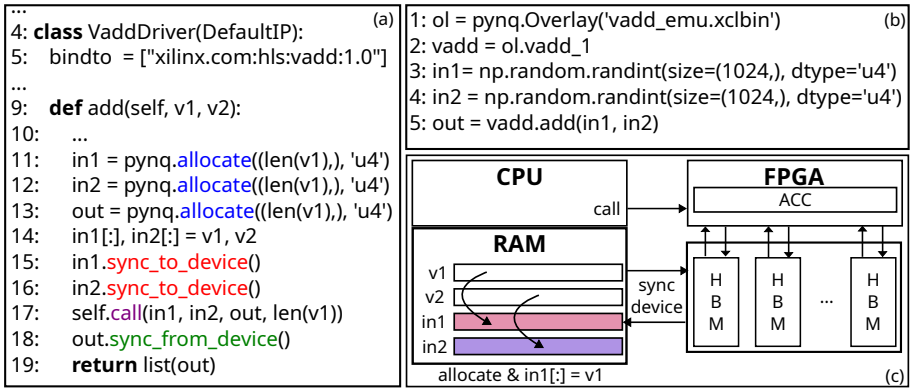

Figura 1. (a) *Driver* em PYNQ do acelerador para soma de vetores; (b) Chamada do acelerador; (c) Localização e transferência entre as regiões de memória

2.3. Aceleradores em HLS e RTL

Para reduzir a complexidade do desenvolvimento de projetos com FPGA, as abordagens com descrições em alto nível evoluíram significativamente na última década. Pode-se destacar o uso de ferramentas de síntese de alto nível (*High Level Synthesis* – HLS) como SmartHLS[1], Intel HLS e Xilinx HLS. Outra forma de reduzir a complexidade é o uso de geradores de códigos para domínios específicos [da Silva et al. 2017]. A vantagem é personalizar o código em função das especificidades da aplicação. A seguir são ilustradas duas abordagens para o caso de estudo do algoritmo K-means.

2.3.1. K-means Rodinia HLS

A Figura 2 ilustra a implementação em HLS C++ do código básico para o algoritmo K-means disponibilizado na base Rodinia-HLS [Cong and et al. 2018]. Pode-se observar que é um código em C++ do K-means com algumas diretivas. As diretivas ilustradas na Figura 2 definem três conexões *AXI* com a memória global que irão fornecer os três vetores: (a) *feature* com todas as amostras, (b) *membership* armazenará a classificação de cada amostra; (c) *cluster* com os centroides. São três laços aninhados no código, o laço externo varre todas as amostras, o segundo laço varre todos os centroides e o laço mais interno varre todos os atributos do ponto e do centroide em questão, calcula a distância euclidiana e seleciona o centroide mais próximo.

[1] https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler

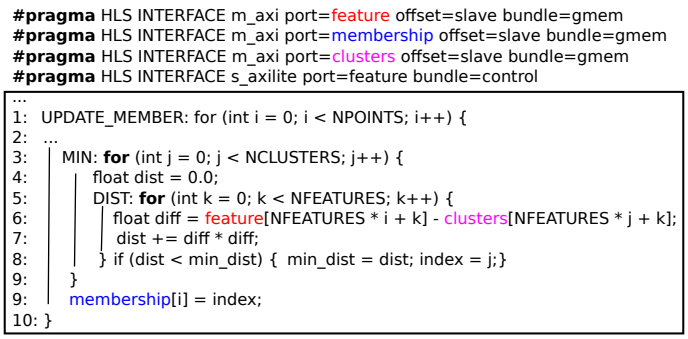

Figura 2. Código básico C++ HLS para K-means Rodinia.

Além da versão básica, cinco implementações otimizadas são disponibilizadas na base Rodinia-HLS [Cong and et al. 2018]. A versão com maior desempenho está disponível no repositório do GitHub[2]. Primeiro, o código tem 10× mais linhas, ou seja, a codificação não é trivial. Segundo, o maior impacto é no tratamento da memória. O código carrega os *clusters* e blocos das amostras (*tile*) para uma memória local e classificação também é gravada localmente. Estas operações podem ser vistas nos métodos *load_local_cluster*, *load_local_feature* e *store_local_membership* nas linhas 4, 16 e 60, respectivamente. O algoritmo de classificação é basicamente o mesmo e esta implementado no método *compute_local_membership* na linha 28 e possui três laços aninhados, porém, faz a expansão dos laços internos com o *pragma unroll*. Para o laço externo e para os acessos às memórias, o *pragma pipeline* é usado. Finalmente, no método *workload* na linha 72 é declarado as memórias locais e realizado as chamadas dos métodos por blocos. Resumindo, o código pode gerar desempenho, mas é necessário uma capacitação do programador para conhecer as diretivas e estruturas que serão geradas em *hardware*.

Apesar da relativa simplicidade do HLS, o código do Rodinia [Cong and et al. 2018] define de forma estática o número de amostras, centroides e atributos. Ou seja, em tempo de compilação para otimizar a geração de código RTL para FPGA. Esta opção de projeto reduz a flexibilidade, pois até para alterar o número de amostras é necessário recompilar, o que pode demorar horas.

2.3.2. K-means RTL

Uma vez que o desenvolvimento em Verilog também não é trivial para a comunidade de software, outra alternativa é usar um gerador de código de domínio específico para um nicho de aceleradores. A Figura 3 apresenta o gerador proposto por [Bragança and et al. 2021]. Este trabalho dá suporte para a geração de várias cópias de aceleradores que acessam a memória externa de forma independente como mostrado na Figura 3(a). Cada amostra carregada da memória pode conter vários pontos, os quais podem ser computados em paralelo conforme mostrado na Figura 3(b). Cada ponto então é enviado para um grafo de fluxo de dados que faz o cálculo das distâncias para todos os centroides, seguido da redução para classificação, como ilustrado na Figura 3(c). Esta etapa equivale aos dois laços internos do código HLS do Rodinia. O gerador é pa-

[2]https://github.com/SFU-HiAccel/rodinia-hls/blob/master/Benchmarks/kmeans/kmeans_6_multiddr/src/kmeans.cpp

rametrizado em função do número de centroides e dimensões. A implementação proposta em [Bragança and et al. 2021] faz uma interface com código OpenCL para chamada. Na Seção 3.2 é descrita a adaptação para a interface PYNQ. Em comparação com a implementação HLS, a versão RTL permite a configuração dinâmica do número de amostras. Apenas o número de centroides e atributos são definidos estaticamente.

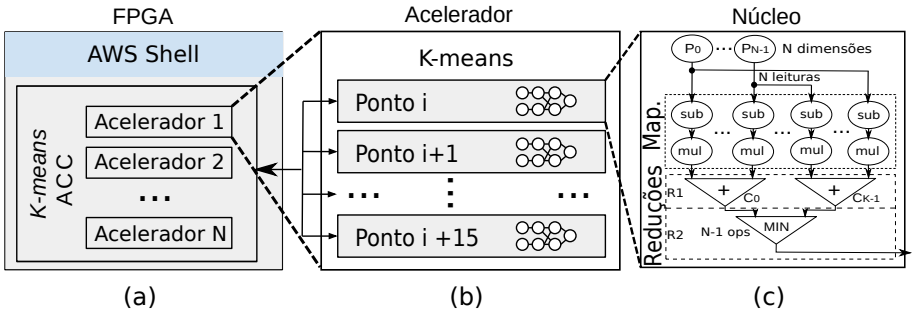

Figura 3. K-means RTL [Bragança and et al. 2021]. (a) Cópias dos aceleradores. (b) Computação dos atributos. (c) Fluxo de dados para classificação das amostras.

3. Integração Python com os Aceleradores

3.1. K-means Rodinia HLS

Neste estudo de caso, o acelerador K-means em HLS requer três vetores: os centroides, os pontos de entrada e um vetor de saída com o resultado da classificação de todos os pontos. O PYNQ permite a passagem de listas/vetores *numpy* como entrada, porém um espaço de memória contígua deve ser alocado com o método *pynq.allocate* como mostrado nas linhas 6, 7, 8 da Figura 4. K é o número de centroides e N é número de atributos (ou pontos). Este método retorna um objeto da classe *pynq.Buffer* que herda da classe *numpy.array*, desse modo, é possível usar todas as operações implementadas no pacote *numpy*. Os principais métodos utilizados nesta função são *sync_to_device* e *sync_from_device* responsáveis por enviar e receber os dados para/da memória global do FPGA, respectivamente. O método *call* dispara a execução no acelerador.

(a)

```
1:class KmeansHLS(pynq.DefaultIP):
2:    ...
3:    def __init__(self,description):
4:       ...
5:    def allocate(self, k, n, num_points, dtype,ol):
6:       self.cluster_buffer = pynq.allocate((k*n,), dtype=dtype)
7:       self.input_buffer = pynq.allocate((num_points*n,), dtype=dtype)
8:       self.output_buffer = pynq.allocate((num_points,), dtype=np.uint32)
9:    def classify(self, k, n, clusters, data,ol):
10:      ...
11:      self.allocate(k, n, len(data),data.dtype,ol)
12:      ...
13:      self.input_buffer.sync_to_device()
14:      self.cluster_buffer.sync_to_device()
15:      self.call(self.cluster_buffer,self.input_buffer, self.output_buffer)
16:      self.output_buffer.sync_from_device()
17:      return np.array(self.output_buffer, dtype=np.uint32)
```

(b)

```
1: from kmeans import KMeansFPGA

2: N,K =8,4
3: df = pd.read_csv('USCensus1990.data.txt',
          header=None, nrows=2000000,
          dtype=np.ushort,
          usecols=list(range(1,N+1)))
4: km = KMeansFPGA(K,N,1,xclbin[0]).fit(df)
5: p = km.predict(df)
```

(c)

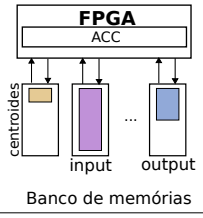

Figura 4. (a) *Drive* para K-means HLS; (b) Exemplo de código para chamada do K-means; (c) Distribuição dos vetores HLS nos bancos HBM.

Ao instanciar o objeto *pynq.Overlay*, o FPGA será configurado e o programador passa a ter acesso ao acelerador através dos métodos implementados. No exemplo, o método *classify* foi implementado, recebe os vetores para o kernel HLS e retorna uma lista com a classificação das amostras. A Figura 4(c) mostra como os vetores de centroides, dados e classificação estão distribuídos nos bancos HBM.

3.2. K-means RTL

Para o K-means RTL foi implementado um *driver* semelhante ao *driver* do K-means HLS. Porém, existe uma diferença na forma de tratar os dados de entrada e algumas modificações foram feitas. A primeira diferença é que esse acelerador possui apenas dois vetores passados como parâmetros, um para entrada e outro para saída dos dados. O vetor de saída possui a mesma lógica que o K-means HLS. Porém, o vetor de entrada tem dois campos de informação. O primeiro campo tem a configuração que permite enviar os valores dos centroides para o acelerador. O segundo campo são os dados de entradas. Apesar de pequenas diferenças, a interface do *driver* é a mesma do K-means HLS. A implementação é mostrada na Figura 5(a), onde nas linhas de 36, 37 e 38, os campos do vetor de entrada são preenchidos com os valores dos centroides e dos dados, respectivamente. Apesar de estarem no mesmo vetor, como ilustra a Figura5(b), os centroides são carregados no grafo de fluxo de dados e apenas as amostras são lidas durante a execução, gerando um fluxo de saída com a classificação.

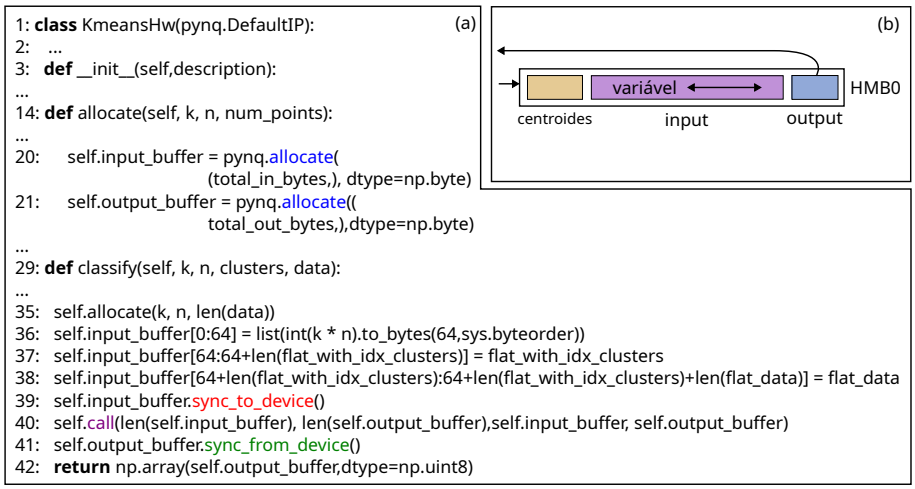

Figura 5. *Driver* para K-means RTL; (b) Três campos em um banco: centroides, dados e classificação.

3.3. Instanciação e chamada do Acelerador

Conforme descrito na Seção 2.2 o *driver* dos aceleradores é automaticamente instanciado pelo PYNQ no momento do carregamento do *bitstream*. Uma vez feito o carregamento é possível acessar o *kernel* e executar o método implementado no *driver*. A Figura 4(b) apresenta um exemplo de chamada do acelerador de K-means em uma célula do *Jupyter Notebook*. Na linha 2, o arquivo *CSV* com os dados é carregado. Na linha 4, o acelerador é instanciado e seu *bitstream* é carregado, além da passagem dos parâmetros de K, N. Finalmente, na linha 5, a função para classificação é chamada e executada.

4. Resultados

Esta Seção apresenta a validação do uso do PYNQ integrado a chamadas de código em Jupyter Notebook com acesso ao FPGA Alveo U55C de forma transparente para uso da fase de classificação do algoritmo K-means. A Tabela 1 mostra uma comparação de desempenho da implementação do Scikit-learn executando em um sistema com dois processadores Intel(R) Xeon(R) Silver 4210R com um total de 20 núcleos, 14.080K de cache L3 e 2,40GHz de *clock*. As medidas de tempo foram executadas usando a função *time.time()* do Python3. O código foi executado usando como entradas um conjunto de 2 milhões de amostras da base de dados *US Census 1990* [Dheeru and Karra Taniskidou 2017]. A classificação foi realizada usando 8 atributos ou dimensões, ou seja, $N = 8$. Para cada teste foram realizadas três execuções com 4, 8 e 16 centroides.

Tabela 1. Comparação do tempo de execução do Scikit-learn com os aceleradores: RTL, RodiniaHLS básico e otimizado. Os tempos foram medidos com *time.time()* do Python3 em milisegundos para 2 milhões de amostras.

Código	K	N	Tempo de alocação	CPU→FPGA	FPGA→CPU	Tempo Kernel	Acel. Kernel	Tempo Total	Acel. Total
Scikit-learn parallel	4	8	-	-	-	-	1	303,61	1
	8	8	-	-	-	-	1	316,42	1
	16	8	-	-	-	-	1	468,24	1
Rodinia HLS básico	4	8	33,97	6,37	0,95	26,29	11,55	67,58	4,50
	8	8	33,90	6,66	1,11	80,06	3,95	121,73	2,60
	16	8	32,73	6,59	1,15	1.200,77	0,39	1.241,24	0,38
Rodinia HLS Otimizado	4	8	25,35	5,89	0,84	7,20	42,17	39,28	7,73
	8	8	33,41	6,26	0,92	7,20	43,95	47,79	6,62
	16	8	33,93	7,27	1,05	7,36	63,62	49,61	9,44
RTL	4	8	13,17	2,93	0,37	3,27	92,85	19,74	15,38
	8	8	12,98	2,84	0,33	3,44	91,98	19,59	16,15
	16	8	12,83	2,85	0,37	3,30	141,89	19,35	24,20
RTL 2 cópias	8	8	14,65	3,07	0,44	1,69	187,23	19,85	15,94
4 cópias	8	8	26,69	4,27	0,41	1,04	304,25	32,41	9,76

As três primeiras linhas mostram os valores do tempo de execução para a chamada do Scikit-learn para classificar as 2 milhões de amostras. Como os dados já estão na memória do processador, apenas o valor do tempo total de execução é mostrado na penúltima coluna. Sabendo que o número de operações cresce com $O(3K)$, pode-se observar que mesmo com 4 vezes mais centroides ($K = 16$), o tempo de execução é apenas 1,5$\times$ maior, pois a implementação explora o paralelismo com os múltiplos núcleos. Além disso, para valores maiores de K tem-se mais reuso dos dados, aumentando a intensidade aritmética. Nas colunas que apresentam a aceleração do kernel da implementação e do tempo total (incluindo transferências de dados), todos os valores são iguais a 1 para o Scikit-learn, que é a base para comparação.

O tempo de alocação das HBM foi basicamente o mesmo para as seis execuções da versão HLS do K-means do Rodinia [Cong and et al. 2018]. Comparando com o tempo de alocação da versão RTL, pode-se observar uma diferença de 3 vezes. Este valor se justifica, pois a versão RTL aloca um banco de memória HBM, enquanto que a versão HLS distribui o conjunto de dados, o conjunto de centroides e a classificação de saída em três bancos distintos de HBM. O mesmo efeito acontece nos tempos de transferência entre CPU→FPGA, sendo a versão RTL 2$\times$ mais rápida por usar apenas 1 banco.

Considerando a coluna do tempo de execução do Kernel do algoritmo para a versão HLS básica, pode-se observar que o tempo para 8 centroides é 4× mais lento que o tempo com 4 centroides. O laço da varredura dos centroides é o mais interno, como ilustrado na Figura 2. Porém, a execução com 16 centroides é 16× mais lenta, pois a leitura da memória não é agrupada. A implementação HLS mais otimizada do trabalho [Cong and et al. 2018] apresenta o mesmo tempo de execução independente do número de centroides, impactando em um ganho de 42,17 a 63,62× mais rápido que a execução da versão Scikit-learn. Ao considerar o tempo total que contabiliza as transferências de memória, a versão HLS ainda é 6,6 a 9,4× mais rápida. O tempo de execução do *kernel* não muda, pois não depende do valor de K e está limitado pela memória. Os laços internos do código do K-means foram expandidos gerando um paralelismo espacial e temporal com a diretiva de pipeline. Entretanto, como um banco de memória fornece dados com a vazão aproximada 10 GB/s e a versão otimizada está personalizada para inteiros de 32 bits, O tempo para leitura dos dados será no mínimo $\frac{10GB/s}{(2M\times8\times4)} = 6,4\ ms$. O tempo medido na execução foi em torno de 7 ms.

A versão RTL executa 2× mais rápido que a implementação otimizada em HLS, apesar de ter 4× mais recursos no grafo de fluxo de dados para explorar o paralelismo espacial e temporal. O motivo é que o gerador RTL proposto em [Bragança and et al. 2021] tem uma interface de memória de 512 *bits* e a saída do banco HBM é de 256 bits, sendo necessário 2 ciclos para enviar um pacote de dados. Portanto, a vazão de dados cai para 5 GB/s. A versão RTL pode ser até 141,9× mais rápida que a implementação Scikit-learn, considerando a execução isolada do kernel. Mesmo considerando o tempo total com as sobrecargas das chamadas dentro do Jupyter Notebook, as transferências de dados e alocação de memória, a execução com acelerador integrado é até 24,2× mais rápida que o Scikit-learn que executa em 2 processadores de alto desempenho totalizando 20 núcleos. Por fim, o FPGA Alveo U55C tem 32 bancos de memória. Se os dados forem distribuídos nos bancos, é possível disparar vários aceleradores em paralelo.

As duas últimas linhas da Tabela 1 mostram o tempo de execução para 2 e 4 cópias do acelerador RTL para $k = 8$ alocando 2 e 4 bancos de HBM, respectivamente. Apesar do ganho de desempenho do kernel, o tempo de alocação e o tempo de transferência degradam o tempo total ao manipular bancos de HBM distintos.

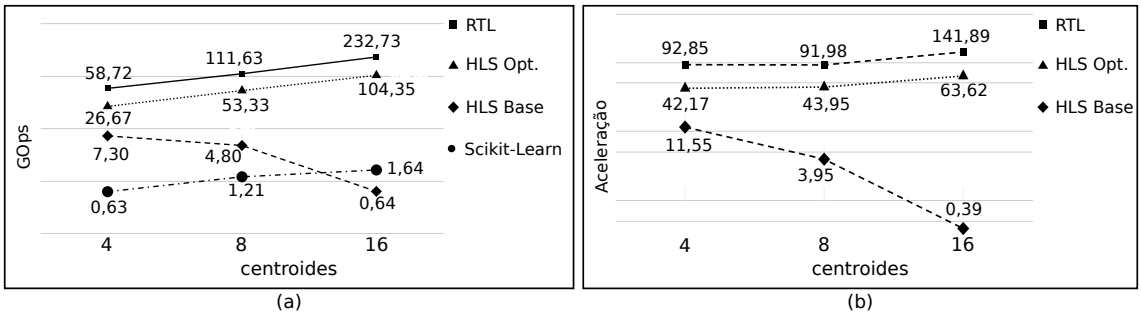

Figura 6. (a) Desempenho em Gops/s para Scikit-learn, HLS Rodinia Base e Otimizado e RTL; (b) Aceleração das implementações em hardware em comparação com Scikit-learn.

A Figura 6 apresenta o desempenho em Gops/s para todas as implementações. O classificador do K-means executa $3 \times K \times N \times M$ operações, onde K é o número de

centroides, N é o número de atributos e M é o número de amostras (2 milhões para os testes). Observe que quanto maior for o valor de K maior será o desempenho, pois o tempo de execução está limitado pela memória. Como a quantidade de dados que é a mesma em todos os experimentos, o tempo não muda. O desempenho da implementação Scikit-learn no processador de alto desempenho usando múltiplos núcleos é de no máximo de 1,6 Gops/s comparado com o desempenho de 232,3 Gops/s da versão RTL para $K = 16$, considerando que os dados já estão na memória do FPGA.

Tabela 2. FPGA Alveo U55C e utilização de recursos

Acelerador	K	LUT	LUTasMEM	REG	BRAM	DSP	Compilação
Rodinia Base	4	4.421	1.045	7.588	15	32	02h 44m 01s
	8	4.991	1.029	8.000	15	64	02h 44m 01s
	16	3.886	710	8.095	15	8	02h 38m 50s
Rodinia Otimizado	4	8.241	1.581	14.152	119	96	03h 05m 45s
	8	10.501	1.467	16.637	119	192	03h 10m 21s
	16	16.579	2.253	22.617	119	384	03h 11m 10s
RTL	4	9.350	306	11381	7	384	02h 44m 25s
	8	17.632	318	21.768	7	768	02h 56m 28s
	16	34.361	402	40.041	7	1.536	03h 23m 53s
RTL 2 cópias	8	35.286	636	42.421	15	1.536	03h 09m 04s
4 cópias	8	70.378	1.272	84.750	30	3.072	04h 05m 54s

A Tabela 2 apresenta a utilização de recursos para os 11 projetos de aceleradores. Primeiro, pode-se observar que o tempo de compilação é o maior gargalo, sendo, em média, 2 horas e 57 minutos para criar o acelerador. Segundo, para a versão simples em HLS para $K = 16$, o compilador não faz a expansão dos laços e aloca apenas 8 unidades DSP de cálculo, justificando o baixo desempenho. A versão otimizada gasta em média $2\times$ mais recursos de LUTs e REGs em comparação com a base. O consumo maior de módulos BRAM se justifica, pois os dados são alocados em memórias locais dentro do FPGA. Em relação aos DSPs para $K = 16$ com 384 unidades a 300 MHz, tem-se 115 Gops/s que é o desempenho verificado na execução, como ilustrado na Figura 6(a). A versão do acelerador RTL utiliza $4\times$mais DSPs. Poderia gerar um desempenho $4\times$ maior, mas como o acelerador usado [Bragança and et al. 2021] tinha sido projetado para uma interface de 512 *bits* e foi acoplado em uma interface com 256 *bits*, o desempenho foi reduzido por um fator $2\times$. Uma solução seria modificar o gerador para uma interface de 256 *bits*. Outra opção é aumentar o número de aceleradores usando 2 ou 4 cópias. As duas últimas linhas mostram que o uso de recursos para $k = 8$, escala com o número de cópias. Porém, a frequência de relógio é reduzida de 300 MHz para 265 e 241 MHz, respectivamente, degradando ganho de desempenho.

Finalmente, é observado várias vantagens do uso do PYNQ. O objetivo deste trabalho não é otimizar ao máximo o caso de estudo, mas ilustrar que podemos adaptar soluções existentes para ter desempenho, quais os novos gargalos e desafios podem surgir com a integração e encontrar onde existe espaço para melhorias.

5. Trabalhos Relacionados

Estão presentes na literatura alguns trabalhos que fazem o uso do PYNQ para realizar o *offload* de computação para aceleradores em FPGA. Kachris e colaborado-

res desenvolveram *SPynq* um ambiente de desenvolvimento que permite o uso do framework *Spark* que visa a programação distribuída em ambientes paralelos em uma série de placas de prototipagem de FPGAs [Kachris and et al. 2017]. Os FPGAs usados foram da série Zynq *all-programmable* SoC, onde um cluster com 4 placas foi configurado para acelerar aplicações de aprendizado de máquina. Posteriormente, Kachris e colaboradores apresentaram um estudo de caso do algoritmo K-means utilizando o ambiente de desenvolvimento Spynq. O experimento foi executado em um *cluster* com 4 placas Zynq, o ganho de desempenho foi de 2× em comparação com um processador Xeon E5-2658 [Kachris and et al. 2018].

Yuke Wang e colaboradores apresentaram KPynq, uma ferramenta para execução do K-means nas placas de prototipagem ZYNQ *all-programmable* SoC [Wang and et al. 2019]. A validação foi realizada usando o MNIST para reconhecimento de dígitos, mostrando uma aceleração de 2,95× comparado ao processador ARM Cortex-A9 com 650 MHz. Seguindo a mesma linha de experimentação com PYNQ, Cérin e colaboradores utilizaram a placa PYNQ-Z2 e o processador ARM para classificar vários conjuntos de dados com o K-means e obteve um ganho de desempenho 2× sobre a solução com processador [Cérin and et al. 2019]. Um gerador otimizado para K-means foi apresentado recentemente em [Wang and et al. 2021] e avaliado em um FPGA ZUC102 de alto desempenho, mas sem memórias HBM e sem acoplamento com PYNQ. Os ganhos de aceleração foram da ordem de 5-10× em comparação com um processador Xeon, porém usando valores altos de K que é vantajoso para o FPGA.

Entretanto, nenhum destes trabalhos avaliou o K-means em FPGAs de alto desempenho com memórias HBM comparando com implementações paralelas em processadores de múltiplos núcleos considerando a integração com o ambiente PYNQ.

6. Conclusão

Novas ferramentas estão surgindo permitindo a integração de aplicações em Python com aceleradores em FPGA. Entretanto, a maioria dos trabalhos presentes na literatura são avaliados em placas de desenvolvimento para embarcados. Este trabalho avaliou o K-means em um ambiente de alto desempenho com o FPGA Alveo U55C conectado a memórias HBM. Os resultados mostraram que é possível encapsular aceleradores desenvolvidos em HLS ou RTL e ainda obter desempenho de até 24x em comparação a um sistema com duas CPUs Xeon(R) Silver 4210R com 10 núcleos cada, mesmo considerando a transferência de dados. Os artefatos produzidos neste trabalho estão disponíveis no Github[3].

Referências

Bragança, L. and et al. (2021). An open source custom k-means generator for aws cloud fpga accelerators. In *Brazilian Symp on Computing Systems Engineering (SBESC)*.

Caldeira, P. and et al. (2018). From java to fpga: An experience with the intel harp system. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 17–24.

Canesche, M. and et al. (2021). Google colab cad4u: Hands-on cloud laboratories for digital design. In *International Symposium on Circuits and Systems (ISCAS)*.

[3]`https://github.com/lesc-ufv/artefatos_wscad2022`

Cérin, C. and et al. (2019). Where are the optimization potential of machine learning kernels. In *International Conference on Big Data Intelligence and Computing*.

Cong, J. and et al. (2018). Understanding performance differences of fpgas and gpus. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96. IEEE.

da Silva, L. B., Almeida, D., Nacif, J. A. M., Sánchez-Osorio, I., Hernández-Martínez, C. A., and Ferreira, R. (2017). Exploring the dynamics of large-scale gene regulatory networks using hardware acceleration on a heterogeneous cpu-fpga platform. In *IEEE Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*.

Dheeru, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository http://archive.ics.uci.edu/ml. In *arXiv preprint arXiv:1710.11342*.

He, Z., Parravicini, D., Petrica, L., O'Brien, K., Alonso, G., and Blott, M. (2021). Accl: Fpga-accelerated collectives over 100 gbps tcp-ip. In *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*.

Kachris, C. and et al. (2017). Spynq: Acceleration of machine learning applications over spark on pynq. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 70–77. IEEE.

Kachris, C. and et al. (2018). Seamless fpga deployment over spark in cloud computing: A use case on machine learning hardware acceleration. In *International Symposium on Applied Reconfigurable Computing*, pages 673–684. Springer.

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137.

Rule, A., Tabard, A., and Hollan, J. D. (2018). Exploration and explanation in computational notebooks. In *Conference on Human Factors in Computing Systems*.

Silva, L. B. D., Ferreira, R., Canesche, M., Menezes, M. M., Vieira, M. D., Penha, J., Jamieson, P., and Nacif, J. A. M. (2019). Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–20.

Wang, Y. and et al. (2019). Kpynq: A work-efficient triangle-inequality based k-means on fpga. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 320–320. IEEE.

Wang, Y. and et al. (2021). Tiacc: Triangle-inequality based hardware accelerator for k-means on fpgas. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 133–142. IEEE.

Xilinx (2022). Python productivity for zynq. `http://www.pynq.io/`.