https://eventos.utfpr.edu.br//sicite/sicite2020

Avaliação da ferramenta *Numba* no desenvolvimento de algoritmos de reconstrução de imagens

Evaluation of the *Numba* tool in the development of image reconstruction algorithms.

Lucas Henrique Kelniar
lucaskelniar@alunos.utfpr.edu.br
Universidade Tecnológica Federal do Paraná, Pato Branco, Paraná, Brasil

Giovanni Alfredo Guarneri
giovanni@utfpr.edu.br
Universidade Tecnológica Federal do Paraná, Pato Branco, Paraná, Brasil

RESUMO

Este trabalho tem o objetivo de avaliar o uso da ferramenta *Numba* na otimização do tempo de processamento de códigos *Python* utilizados para reconstrução de imagens com dados oriundos de inspeções não destrutivas por ultrassom. Foram avaliados somente os algoritmos de reconstrução de imagens baseados no princípio de atraso-e-soma e que são amplamente utilizados: *Synthetic Aperture Focusing Technique* (*SAFT*) e o *Total Focusing Method* (*TFM*). O *Numba* é um compilador *just in time* usado para melhorar o tempo de execução, principalmente em códigos que fazem uso extensivo de operações matemáticas, estruturas de repetição e usam recursos da biblioteca *NumPy*. Como os algoritmos de reconstrução de imagens usados atendem essas características, este trabalho propõe técnicas de adaptação dos códigos para utilizar o *Numba*, e a avaliação das implementações dessas técnicas pela medição e comparação dos tempos de execução. Foi observado uma melhora de aproximadamente 2 vezes para o SAFT e 6 vezes para o TFM nos tempos de execução com aplicação das técnicas, mostrando que o *Numba* é uma ótima alternativa para melhorar o desempenho na execução dos algoritmos de reconstrução de imagens.

PALAVRAS-CHAVE: Ensaios não destrutivos. Ultrassom. Reconstrução de imagens.

ABSTRACT

This work aims to evaluate the *Numba* tool's use to optimize the processing time of Python codes used to reconstruct images with data from non-destructive ultrasonic inspections. Only the widely used image reconstruction algorithms based on the principle of delay-and-sum were evaluated: Synthetic Aperture Focusing Technique (SAFT) and the Total Focusing Method (TFM). *Numba* is a just in time compiler used to improve execution time, mainly in codes that make extensive use of mathematical operations, repetition structures and use *NumPy* library resources. As the image reconstruction algorithms used meet these characteristics, this work proposes techniques to adapt the codes to use *Numba* and evaluate these techniques' implementations by measuring and comparing execution times. It was observed an improvement of approximately two times for SAFT and six times for TFM when applying some *Numba* techniques, showing that *Numba* is an excellent alternative to improve the performance in the execution of image reconstruction algorithms.

KEYWORDS: Non-destructive testing. Ultrasound. Image reconstruction.

Recebido: 19 ago. 2020.
Aprovado: 01 out. 2020.

INTRODUÇÃO

Em uma tentativa de garantir cada vez mais segurança, confiabilidade, utilidade e qualidade aos produtos e serviços para a sociedade, a humanidade está sempre em busca de técnicas e tecnologias que tornem isso possível. Os ensaios não destrutivos (ENDs), são técnicas que tornaram possível avaliar diversas características de um material: tamanho, composição, defeitos e suas dimensões, dentre outras; como forma de verificar características e condições que comprometam sua qualidade e sem corromper esse material de nenhuma forma (HELLIER, 2003). A aplicação dos ENDs trazem uma contribuição para a qualidade dos bens e serviços, redução de custos nos processos envolvidos e a preservação da vida e do meio ambiente onde são aplicados, é muito popular na indústria e tem uma vasta aplicação e inclusive na área médica (ABENDI, 2014).

O Brasil possui uma grande matriz de extração de petróleo em alto mar (ou *off shore*, então inspeções frequentes para dimensionar e monitorar falhas nessas tubulações são de extrema importância para garantir a integridade dos procedimentos em execução.

Este trabalho faz parte de um projeto de reconstrução de imagens em ensaios não-destrutivos por ultrassom que tem como objetivo principal desenvolver ferramentas de análise para auxiliar nas inspeções de tubulações submarinas.

O projeto tem diversas frentes de trabalho, e em uma dessas frentes foi desenvolvida uma ferramenta que disponibiliza uma interface gráfica para a leitura, manipulação e o processamento de dados de inspeções por ultrassom. Atualmente, essa ferramenta atende diversas funcionalidades, distribuídas nas etapas de leitura e conversão dos dados provenientes de diversas fontes, sejam elas transdutores comerciais ou simuladores, pré-processamento desses dados, algoritmos de reconstrução de imagens e pós-processamento das imagens.

Os algoritmos de reconstrução de imagens foram desenvolvidos na linguagem de programação *Python*, que é bastante usada em aplicações científicas (MAROWKA, 2018), e recomendada em aplicações que precisam de desenvolvimento rápido, implantação de produção (ou quando uma aplicação fica disponível para o usuário final) e sistemas escaláveis (GORELICK; OZSVALD, 2014). Mas, em aplicações que demandam grande quantidade de operações computacionais, como é o caso dos algoritmos de reconstrução de imagem tem um desempenho inferior quando comparado com linguagens compiladas como *C, C++* e *FORTRAN* (LANARO, 2017). Por outro lado, existem diversas ferramentas disponíveis na linguagem Python para melhorar o desempenho do processamento, dentre elas, o *Numba*.

Desta forma, este trabalho tem como proposta utilizar o *Numba* para a melhoria do desempenho no tempo de processamento em algoritmos de reconstrução de imagem baseados no método atraso-e-soma (do inglês, *delay-and-sum*) (BESSON *et al.*, 2016): *SAFT* (*Synthetic Aperture Focusing Technique*, tradução para: Técnica de Focalização pela técnica de abertura sintética), *TFM (Total Focusing Method,* tradução para Método de Focalização Total*).*

METODOLOGIA

Para melhorar o tempo de execução dos algoritmos de reconstrução de imagem foi utilizado a ferramenta *Numba* (NUMBA, 2020). Porém, para que isso ocorra efetivamente, o código ao qual serão aplicados precisam seguir uma estrutura específica.

No caso do *Numba,* a maneira mais comum, e que também é usada neste trabalho, é aplicá-lo em funções que se baseiam no uso de estruturas de repetição do *Python (*ex.: *for..in, while,* etc.*),* funções que contenham funcionalidades otimizáveis pelo *Numba* na biblioteca *NumPy,* ou ainda uma combinação desses. Além disso, existem duas variações de otimização principais: a compilação no tempo de execução usando o decorador *@jit (just in time,* ou em tempo de execução*),* ou o decorador *@njit (*em tempo de execução, mas sem modo objeto). O *Numba* analisa as funções em *Python,* implementa algoritmos inteligentes para identificar os tipos das variáveis, e as compila em uma representação intermediária de nível mais baixo, para uma execução mais rápida (LANARO, 2017).

No *Python* existe a representação Objeto *Python* (ou *PO, Python Object*) em que uma variável pode assumir diferentes tipos de valores. Quando se está otimizando, isso pode apresentar obstáculos para a melhoria do desempenho, então foram utilizados ambos os decoradores nos algoritmos, o *@jit* e o *@njit.* Usando o *@njit* e respeitando a condição de não usar variáveis do tipo *PO,* esperava-se um tempo de execução dos algoritmos igual ou inferior aos quais se aplica o *@jit*. Também foi usado o artifício de paralelismo, em que o compilador do *Numba* é habilitado para usar várias unidades de processamento da unidade de processamento central (ou a *CPU, Central Processing Unit*) do computador para executar as operações. Neste trabalho foi utilizado esse recurso para possibilitar que várias instâncias de uma estrutura de repetição, em uma função que esteja aplicada o *Numba,* fossem processadas simultaneamente. Isso tende a melhorar o tempo de execução das funções que possuam iterações, e gerar resultados efetivos desde de que uma instância de uma estrutura de repetição seja descorrelacionada de outra, ou seja, que qualquer instância possa ser executada a qualquer momento, não dependendo da ordem.

Foi utilizado o *Numba* na versão 0.45.1 nos algoritmos *SAFT* e *TFM,* em que foi analisado o tempo de execução desses algoritmos com a função *time* da biblioteca *time,* e usado um conjunto de dados de inspeção simulados em comum para os dois algoritmos.

As técnicas de adaptação dos códigos dos algoritmos usadas nesse trabalho para o uso do *Numba* foram*:*

a) Colocar o procedimento de cálculo correspondente à técnica de reconstrução usada em forma de função do *Python* (esse é o quesito essencial para o uso do *Numba)*;

b) Usar laços de iteração para executar o procedimento respectivo ao procedimento de cálculo usado;

c) Usar variáveis com tipos conhecidos pelo *Numba* na função que se espera otimizar;

d) Adaptar todas funções do *NumPy* não reconhecidas pelo *Numba*;

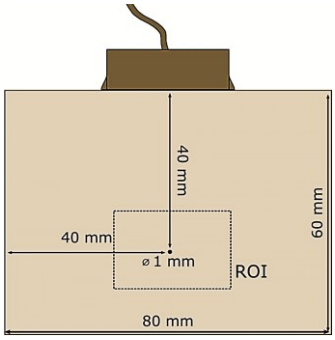
e) Para informar, referir ou calcular um vetor ou matriz, fazer isso de forma particularizada: cada iteração das estruturas de repetição feitas ponto a ponto, ou seja, fazer as iterações percorrendo cada uma das posições;

f) Tornar cada iteração de um laço descorrelacionada de outra, para possibilitar o paralelismo das iterações.

RESULTADOS E DISCUSSÕES

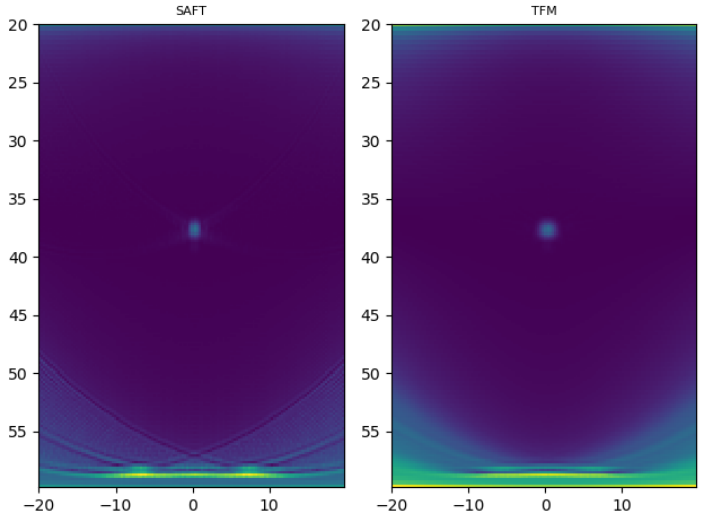
Todas as versões foram executadas neste trabalho de forma a atender a reconstrução de uma imagem de 200 pixels de altura por 130 pixels de largura (ou 26 mil pixels, cada imagem), representando uma ROI (Region of Interest, tradução para: Região de Interesse) de 40 mm de altura por 40 mm de largura. O canto dessa ROI fica 20 mm abaixo da borda superior e 20 mm à direita da borda esquerda, conforme Figura 1, de uma peça simulada pelo CIVA, que é um software que permite a simulação de ENDs (CALMON *et al.*, 2006; CALMON, 2012; RAILLON *et al.*, 2012). A partir dos dados dessa simulação é feita a reconstrução das imagens usando cada um dos algoritmos como mostra a Figura 2.

Figura 1 –Peça simulada que possui uma descontinuidade do tipo SDH (*Side Drilled Hole* – Furo Lateral Passante) com 1 mm de diâmetro na posição indicada.



Fonte: Autoria Própria (2020).

Figura 2 – À esquerda é mostrada a imagem que é gerada pelos algoritmos que utilizam o *SAFT*. À direita, a imagem gerada pelos algoritmos que usam o *TFM*.



Fonte: Autoria Própria (2020).

O tempo de execução de cada algoritmo foi medido repetidamente por 10 vezes e feito a média e o desvio padrão para amostras, em cada caso. No caso dos algoritmos usando o *Numba*, foi medido o tempo repetidamente por 11 vezes e descartado o primeiro resultado. Isso porque na primeira chamada da função que se está otimizando, o *Numba* compila a função com os tipos de argumentos fornecidos nela e gera o código de referência de nível mais baixo, para que seu compilador execute. Durante esse processo, o *Numba* salva os tipos dos argumentos usados em uma memória *cache*, de forma que se aquela função for chamada novamente ou, nesse caso, repetidas vezes, com os mesmos tipos, esse tempo de compilação será encurtado, pois esse processo já vai ter sido feito (NUMBA, 2020). Descartando o primeiro resultado fez-se o cálculo da média e desvio para como nos casos convencionais.

Como mencionado na metodologia os resultados foram obtidos para os algoritmos *SAFT* e *TFM*. Espera-se que aplicando as técnicas mencionadas, consiga-se reduzir o tempo de execução atual desses algoritmos independente do conjunto de dados de inspeção usados. Com a metodologia proposta, esperava-se que a cada versão nova de um algoritmo mais técnicas de adaptação fossem aderidas e melhor seria o ganho de desempenho. Contudo isso nem sempre ocorreu. Muitas vezes a adição de técnicas não impactou em melhoras de desempenho como está apresentado na Tabela 1.

No caso do *SAFT*, o melhor resultado foi a versão *SAFT3* com o *@jit*, mostrando que houve uma evolução positiva com a aplicação das técnicas. E no caso do *TFM* foi a versão *TFM1* com uso de paralelismo de processamento com o *@njit*. Esperava-se que as versões com paralelismo sempre tivessem um desempenho melhor. No entanto, isso nem sempre ocorreu por uma série de fatores. Dentre eles, o fato de que muitas vezes usando uma função pronta da biblioteca *NumPy* é muito mais eficaz do que criar e adaptar uma função nova no Python para aplicar o *Numba*. E mesmo que se use paralelismo, a função pronta ainda mostra esse diferencial. Nesse sentido, não é possível prever com antecedência a melhor forma.

Outra questão é que se esperava que algoritmos usando @njit tivessem um desempenho superior aos seus pares. Isso nem sempre ocorre porque o processo de geração do código de referência é muito sensível aos tipos, e mesmo sem sair do modo “sem objeto Python” – em que é feita a otimização de forma mais eficaz – pode estar ocorrendo alguma troca de atributo que reduz o desempenho relativo daquele caso.

Apesar de exigir um esforço de adaptar os códigos dos algoritmos, a metodologia adotada foi a maneira mais essencial de aproveitar o *Numba*. Com isso já foi possível atingir o objetivo esperado de melhorar o tempo de execução desses algoritmos. Porém, o *Numba* fornece diversas outras formas de melhorar ainda mais o desempenho sem que mais técnicas de adaptação sejam implementadas, entre elas: reduzir a precisão dos valores calculados e habilitar que uma memória salve tipos repetidos em novas compilações. Além dessas, outras técnicas mais avançadas estão disponíveis: usar multiprocessamento gráfico, compilar tipos dos argumentos fora do período de execução, entre outras. A comunidade do *Numba* é ativa e frequentemente novas atualizações são propostas com novidades que permitem novos tipos de otimização.

Tabela 1 – Ganho de velocidade de processamento dos algoritmos usando as otimizações propostas. Em fundo azul e fonte branca os resultados dos algoritmos originais, usados para comparação. E em negrito os melhores resultados entre cada algoritmo usando *SAFT* e *TFM*, respectivamente.

Algoritmo	Otimização	Tempo (s)	Ganho
Implementações seriais:			
SAFT1	**Convencional**	**0,0323±0,003**	**1,000 x**
	@jit	*	*
	@njit	*	*
SAFT2	Convencional	0,621±0,0309	0,052 x
	@jit	0,0302±0,0065	1,070 x
	@njit	0,0267±0,0046	1,210 x
SAFT3	Convencional	2,3025±0,0318	0,014 x
	@jit	**0,0158±0,0016**	**2,044 x**
	@njit	0,02±0,0029	1,615 x
TFM1	**Convencional**	**0,1886±0,0165**	**1,000 x**
	@jit	0,0728±0,0067	2,591 x
	@njit	0,0803±0,0065	2,349 x
TFM2	Convencional	48,2588±1,2266	0,004 x
	@jit	0,1343±0,0183	1,404 x
	@njit	0,1354±0,0122	1,393 x
TFM3	Convencional	48,7147±0,3439	0,004 x
	@jit	0,1455±0,0121	1,296 x
	@njit	0,1601±0,0171	1,178 x
Implementações usando paralelismo:			
SAFT1	@jit	*	*
	@njit	*	*
SAFT2	@jit	0,0268±0,0057	1,205 x
	@njit	0,0248±0,0028	1,302 x
SAFT3	@jit	0,0178±0,0029	1,815 x
	@njit	0,0167±0,0023	1,934 x
TFM1	@jit	0,0325±0,0045	5,803 x
	@njit	**0,0311±0,0039**	**6,06 x**
TFM2	@jit	0,041±0,0027	4,60 x
	@njit	0,0416±0,0031	4,534 x
TFM3	@jit	0,1075±0,002	1,754 x
	@njit	0,1081±0,0035	1,745 x

(*) não é aplicável

Fonte: Autoria Própria (2020).

CONCLUSÕES

O objetivo principal deste trabalho era melhorar o desempenho do tempo de execução dos algoritmos de reconstrução de imagens, baseados no método *delay-and-sum*, usando a ferramenta *Numba*. Aplicando o *Numba* nos algoritmos *SAFT* e

TFM seguindo a metodologia proposta é possível concluir que as técnicas de adaptação propostas são um caminho para implementar melhorias de desempenho, mas elas não são definitivas e nem garantem que a melhoria sempre irá ocorrer, principalmente por conta da técnica (d). A biblioteca *NumPy* foi pensada com o objetivo de fornecer ferramentas computacionalmente eficientes e dificilmente uma função implementada em *Python* terá um tempo de execução inferior do que uma função do *NumPy*, mesmo usando o *Numba*. O ideal é combinar as duas para haver um ganho significativo.

É natural concluir que o esforço de adaptação dos códigos pode ser poupado se desde o desenvolvimento inicial seja considerado um formato adequado para a aplicação do *Numba* em um código. Porém, ele não deve ser usado dessa maneira. E sim para melhorias pontuais de eficiência de processamento que impactam toda a execução. Já era bem sabido nesse projeto que as etapas que mais consumiam tempo e processamento eram os códigos que efetuavam os cálculos pelos métodos propostos, tanto que isso motivou o presente trabalho. Além disso, todos os algoritmos originais atendiam vários requisitos para o uso do *Numba*: código puramente em *Python*, uso da biblioteca *NumPy*, alto consumo de tempo e processamento em funções de cálculos por iteração.

É importante ressaltar também que algumas vezes é necessário checar se o tempo necessário para adaptar um código para o uso do *Numba* é viável para o curso da aplicação final ou o projeto que está inserido. É comum pensar que tudo que mostra a oportunidade de reduzir o tempo de execução de uma aplicação é muito bem-vindo, independentemente da situação. Porém, muitas vezes isso é acompanhado de um gasto de tempo que poderia ser melhor investido em outras aplicações. Todavia, é uma ferramenta recomendada para melhoria de desempenho quando os fatores necessários para sua aplicação são satisfeitos.

Para trabalhos futuros recomenda-se a utilização do *Numba* em outro algoritmo *delay-and-sum*, o CPWC. E também implementar algo semelhante com a ferramenta *Cython*. Isso servirá como um complemento e comparação com o trabalho realizado aqui.

REFERÊNCIAS

ABENDI. **Ensaios não destrutivos e inspeção.** 09 2014. Disponível em: http://www.abendi.org.br. Acesso em: 17 abr. 2020

BESSON, A. *et al.* Compressed delay-and-sum beamforming for ultrafast ultrasound imaging. In: **2016 IEEE International Conference on Image Processing (ICIP)**. Ieee, 2016. p. 2509-2513.

CALMON, P. *et al.* CIVA: an expertise platform for simulation and processing NDT data. **Ultrasonics**, v. 44, n. sup., p. 975-979, jun. 2006.

CALMON, P. Trends and stakes of NDT simulation. **Journal of Nondestructive evaluation**, v. 31, n. 4, p. 339-341, 2012.

GORELICK, M.; OZSVALD, I. **High performance python:** practical performant programming for humans. Sebastopol, USA: O'Reilly Media, 2014. 370 p.

HELLIER, C. **Handbook of nondestructive evaluation**. New York, NY, USA: Mcgraw-Hill, 2003. 594 p.

LANARO, G. **Python high performance**. Birmingham, UK: Packt Publishing Ltd, 2017. 264 p.

MAROWKA, A. Python accelerators for high-performance computing. **The Journal of Supercomputing**, v. 74, n. 4, p. 1449-1460, 2018.

NUMBA. **User Manual.** Disponível em: https://numba.readthedocs.io/en/latest/user/index.html. Acesso em: 18 ago. 2020

RAILLON, R. et al. Validation of CIVA ultrasonic simulation in canonical configurations. In: WORLD CONFERENCE ON NON-DESTRUCTIVE TESTING, 18., 2012, Durban, South Africa. **Anais...** Durban: NDT, 2012.