Análise de portabilidade e escalabilidade de algoritmo paralelo de sensoriamento espectral por características cicloestacionárias usando OpenMP

Caio Henrique Costa Souza
Arthur D. L. Lima
Luiz F. Q. Silveira

September 2, 2017

Resumo

A análise cicloestacionária destaca-se entre os métodos de sensoriamento espectral por se mostrar eficaz em cenários de baixa relação sinal ruído (SNR), o programa usado implementa um algoritmo de análise cicloestacionária, o Cyclic Periodogram Detection (CPD). Esse algoritmo permite detectar e classificar modulações através da estimação da função densidade espectral cíclica (SCD), obtendo-se uma superfície de características cicloestacionárias do sinal. Foram utilizadas duas plataformas distintas, a primeira consiste em um nó computacional com um coprocessador Intel Xeon Phi de primeira geração com 60 núcleos. A segunda foi um dispositivo móvel com um processador Qualcomm Snapdragon 415 com 8 núcleos. No Xeon Phi observou-se um speedup bastante próximo do linear, de 56,5 para 59 núcleos no código não otimizado. O código otimizado foi em todos os cenários ao menos 3 vezes mais rápido do que o não otimizado, apesar do speedup mais distante do linear. Já no Qualcomm Snapdragon, o speedup foi consideravelmente mais distante do linear em relação ao Xeon Phi.

Palavras chave

Cicloestacionariedade. Espectro. Paralelo. Escalabilidade. Speedup.

Portability and scalability analysis of a spectrum sensing parallel algorithm by cyclostationary features using OpenMP

Abstract

Cyclostationary analysis stands out among the spectral sensing methods for being effective in scenarios of low signal-to-noise ratio (SNR), the program used implements a cyclostatinary analysis algorithm, the Cyclic Periodogram Detection (CPD). This algorithm allows to detect and classify modulations through the estimation of the Cyclic

Spectral Density (SCD) function, obtaining a surface of cyclostationary features of the signal. Two distinct platforms were used, the first consists of a computational node with a first-generation Intel Xeon Phi coprocessor with 60 cores. The second was a mobile device with a Qualcomm processor Snapdragon 415 with 8 cores. It was observed on the Xeon Phi a speedup very close to the linear, of 56.5 using 59 cores with the non-optimized code. The optimized code was in all cases at least 3 times faster than the non-optimized, although it was more distant from the linear. The speedup was considerably more distant from the linear in the Qualcomm Snapdragon than in Xeon Phi.

Keywords

Cyclostationarity. Spectrum. Parallel. Scalability. Speedup.

1 Introdução

As faixas de frequência livres do espectro eletromagnético, em particular a radiofrequência que vai de $3kHz$ a $300GHz$, estão se tornando cada vez mais escassas devido ao uso em diversas aplicações como rádio e televisão.

Atualmente a política amplamente difundida de alocação de banda é a fixa, em que uma agência reguladora concede o direito de uso exclusivo, por tempo determinado, de faixas de frequência a usuários primários. Essa política acaba gerando um desperdício de banda durante os períodos em que quem possui o direito de uso de dada frequência não está transmitindo.

Uma solução para esse problema é a alocação dinâmica do espectro, onde existem usuários primários e secundários. Nesse cenário, os usuários primários podem transmitir em suas faixas de frequência a qualquer momento, enquanto os usuários secundários podem transmitir sempre que houver uma oportunidade de transmissão. Assim, usuários secundários devem ser capazes de monitorar constantemente o canal, tanto para aproveitarem as oportunidades de transmissão, quanto para interromper sua transmissão o mais rápido possível quando o usuário primário começar a usar o canal.

Os principais algoritmos para fazer o sensoriamento do espectro são a detecção por energia, por filtro casado e por características cicloestacionárias. Esse trabalho aborda a detecção por cicloestacionariedade, que mostra-se superior em cenários de baixa relação sinal ruído (SNR).

Um sinal aleatório $X(t)$ é dito cicloestacionário de segunda ordem quando apresenta média $E[X(t)]$ e autocorrelação $R_x(t,\tau) = E[X(t+\tau)X(t)]$ periódicas com período T_0 [1]:

$$E[X(t)] = E[X(t+T_0)],$$
$$R_x(t+T_0,\tau) = R_x(t,\tau),$$

para todo t e τ. Sendo periódica, a autocorrelação pode ser expressa como uma série de Fourier:

$$R_x(t,\tau) = \sum_{\alpha} R_x^{\alpha}(\tau) e^{j2\pi\alpha t},$$

em que $\alpha = \frac{i}{T_0}, i \in \mathbb{Z}$. Os coeficientes $R_x^\alpha(\tau)$ são chamados de função de autocorrelação cíclica (CAF), dados por:

$$R_x^\alpha(\tau) = \frac{1}{T_0} \int_{-T_0/2}^{T_0/2} R_x(t,\tau) e^{-j2\pi\alpha t} dt.$$

Já a função densidade espectral cíclica (SCD) $S_x^\alpha(f)$ pode ser obtida como uma generalização do Teorema de Wiener–Khinchin, fazendo o coeficiente $\alpha \neq 0$ e usando a transformada de Fourier na CAF:

$$S_x^\alpha(f) = \int_{-\infty}^{\infty} R_x^\alpha(\tau) e^{-j2\pi f\tau} d\tau.$$

Assim, a SCD é uma função de duas variáveis que forma uma superfície de características cicloestacionárias, onde f é a frequência espectral e α é chamado de frequência cíclica.

A proposta desse trabalho foi analisar a portabilidade e escalabilidade, isto é, como a adição de núcleos melhora o processamento de cargas de trabalho maiores, em outras plataformas do algoritmo de estimação da SCD, o Cyclic Periodogram Detection (CPD) [2, 3].

Esse algoritmo é divido em 5 passos:

1. Divide-se o sinal $x[n]$ em L blocos de N amostras:

$$x_l[n], 0 < l < L-1, 0 < n < N-1.$$

2. Usa-se a transformada discreta de Fourier em cada um dos L blocos, obtendo:

$$X_l[n] = \sum_{k=0}^{N-1} X_l[k] e^{-j\frac{2\pi}{N}nk}, 0 < n < N-1.$$

3. Calcula-se a expressão:

$$T_l^\alpha[n] = \frac{1}{N} X_l[k + \frac{\alpha}{2}] X_l^*[k - \frac{\alpha}{2}].$$

4. A média desses L blocos é calculada,

$$T^\alpha[n] = \frac{1}{L} \sum_{l=0}^{L-1} T_l^\alpha[n],$$

5. Por fim, realiza-se uma suavização na frequência:

$$S^\alpha[n] = \frac{1}{M} \sum_{m=0}^{M-1} T^\alpha[kM + m].$$

A implementação paralela do CPD original [2] foi escrita em C e utiliza o OpenMP - uma interface de programação paralela que através de diretivas indica ao compilador quais seções devem ser executadas paralelamente, o que dá uma grande portabilidade de código. A plataforma originalmente utilizada [2] foi um computador com dois processadores AMD Opteron 6172 de 12 núcleos, totalizando 24 núcleos.

Nesse trabalho, foram feitas modificações no programa fazendo com que seja capaz de funcionar em duas plataformas distintas:

- Coprocessador Intel Xeon Phi 5110P, com seus 60 núcleos físicos foi possível analisar a fundo a escalabilidade do CPD paralelo, através de medidas de speedup para várias configurações de carga de trabalho, ou seja, variações nos parâmetros L e N.

- Dispositivo móvel Lenovo Vibe K5 com processador Qualcomm Snapdragon 415, seus 8 núcleos permitiram além de analisar a escalabilidade em dispositivos móveis, realizar um estudo sobre a portabilidade do código vislumbrando aplicações práticas futuras na área de rádio cognitivo usando smartphones.

2 Metodologia

2.1 Intel Xeon Phi

Foram utilizados nós computacionais do NPAD que contém cada um 2 processadores Intel Xeon E5-2698v3, com 16 núcleos físicos de $2.30GHz$ e 32 threads, $40MB$ de cache e $9.6GT/s$ de largura de banda de barramento. E também um coprocessador Intel Xeon Phi 5110P da primeira geração de coprocessadores Xeon Phi, Knights Corner, com 60 núcleos físicos de $1.05GHz$ e 240 threads e $30MB$ de cache.

Existem 3 modos de executar um programa usando o coprocessador Xeon Phi: offload, nativo e simétrico. No modo nativo, são gerados binários que rodam nativamente no coprocessador. No modo offload, a aplicação começa rodando no processador (hospedeiro) da plataforma e seções específicas do código são transferidas para que rodem no coprocessador através de diretivas de programação. Por fim, no modo simétrico as aplicações rodam tanto no hospedeiro quanto no coprocessador.

O hospedeiro usa a distribuição GNU/Linux CentOS 6.5, kernel versão 2.6.32-431 .23.3.el6.x86_64. Para compilação, foi usado o Intel C++ Compiler (ICC) versão 16.0.1 20151021. O coprocessador usa o kernel Linux versão 2.6.38.8+mpss3.6.1.

Nesse trabalho o código original [2] foi modificado para que funcione em modo offload, fazendo com que o hospedeiro inicie algumas variáves e que todo o algoritmo do CPD rode no coprocessador. Apesar de ser possível forçar o uso de todos os 60 núcleos no modo offload através de variáveis de ambiente (KMP_PLACE_THREADS), foram usados no máximo 59, pois essa é a recomendação devido ao overhead do sistema operacional [10].

O código original [2] usa a biblioteca FFTW [4] para cálculo de transformadas de Fourier, mas ao invés de fazer uma compilação cruzada de modo que a FFTW funcione no coprocessador, a interface de suporte de FFTW da intel [5] foi utilizada. Essa interface consiste de wrappers, que oferecem a mesma interface de programação mas com uma implementação diferente, usando as funções de transformações da Intel Math Kernel Library (MKL), que é a biblioteca de matemática da Intel otimizada para seus produtos. Isso tornou o código mais fácil de portar, já que não foi preciso alterar as linhas de código que chamam a transformada de Fourier (com a interface da FFTW).

Por fim, foram feitas otimizações para plataforma como alinhamento de dados e vetorização. Para vetorização, o código foi compilado com a flag −axavx, que gera instruções Intel Advanced Vector Extensions (AVX) [7], que realizam operações Single Instruction Multiple Data (SIMD) de 256 bits para seus processadores.

Os relatórios de otimização do próprio compilador ICC foram usados, através de flags como: −qopt−report, que determina o nível de detalhes do relatório de otimização, −qopt−report−phase, que mostra o relatório de otimização apenas da fase desejada

(vetorização, no caso) e −qoverride−limits, que aumenta os limites de tempo que acabam atrapalhando na otimização.

A otimização foi feita a partir do alinhamento módulo 64 Bytes nas variáveis. Por exemplo, para alocação estática basta reescrever float A[10] como float A[10] __attribute__ ((aligned(64))). Já para dinâmica, usa-se _mm_malloc(A,64) ao invés do malloc convencional. Após fazer o alinhamento é necessário informar o compilador, como __assume _aligned(A,64);.

Tendo feito as alterações e otimizações no código, a vetorização foi feita através de flags como −axavx durante a compilação com o ICC.

2.2 Qualcomm Snapdragon 415

A outra plataforma usada nesse projeto foi um smartphone Lenovo Vibe K5, com um processador Qualcomm Snapdragon 415, com 8 núcleos ARM Cortex A53 de $1.4Ghz$, rodando na custom rom aokp_A6020_nougat_nightly_2017-08-02_0522, usando o Android versão 7.1.2 e kernel versão 3.10.49-perf-g1e56051.

O código foi modificado em um computador com a distribuição GNU/Linux Ubuntu versão 16.04.1, usando o Android Native Development Kit (NDK) versão r15b-linux-x86_64, que permite a compilação de bibliotecas e programas escritos em C/C++ para as arquiteturas comuns de processadores que suportam o Android.

Todas as manipulações do código entre o computador e smartphone, como transferências e executar o código, foram feitas usando acesso remoto através do aplicativo SSHDroid.

Primeiramente, para fazer a compilação cruzada da biblioteca FFTW, o texto publicado por Divick Kishore [8] e o stackoverflow [9] foram usadas como referência.

Após fazer o download da FFTW versão 3.3.6 e do Android NDK, foi necessário gerar o arquivo config.h que contém constantes usadas para portabilidade. Esse arquivo é gerado através do comando configure que faz parte do GNU Build System (Autotools) - um conjunto de ferramentas que visa tornar código fonte portável entre sistemas unix-like. Uma das dificuldades foi descobrir como gerar esse arquivo com suporte para OpenMP. Após uma análise e estudo nos relatórios de compilação (config.log), concluiu-se que as flags necessárias eram −lgomp e −lgcc. Com isso, o arquivo config.h correto foi gerado. O próximo passo na compilação cruzada foi escrever os arquivos Android.mk e Application.mk, que são usados pelo Android Build System durante a compilação. No arquivo Android.mk deve-se indicar o caminho para todos os arquivos contendo código fonte usados pela aplicação. Como a FFTW é bastante complexa e possui vários desses arquivos, ao invés de escrever manualmente o Android.mk, o processo foi automatizado através de shell script. Outro ponto importante é que existem vários arquivos que não foram usados dentro da biblioteca FFTW, como suporte para MPI, threads ou otimizações SSE, AVX etc. A inclusão de qualquer um desses arquivos indesejados impossibilita a compilação, gerando mensagens de erro. Ainda no arquivo Android.mk foi incluido suporte ao OpenMP através das flags −fopenmp. Já no arquivo Application.mk foi indicado que o binário deveria usar a arquitetura arm64-v8a e plataforma android-24. Com isso, finalmente foi gerado o arquivo binário da biblioteca FFTW através do comando ndk−build.

A próxima etapa foi compilar o CPD, houveram erros de compilação em conversões entre tipos float complex da biblioteca complex.h e o tipo complexo usado na FFTW, o fftw_complex. A solução foi abandonar o uso da biblioteca complex.h e reescrever as partes

necessárias usando apenas o fftw_complex. No arquivo Android.mk do CPD, o caminho para a biblioteca compilada anteriormente da FFTW foi indicado. O Application.mk escrito antes foi reutilizado. Finalmente, o binário foi gerado de forma semelhante com o comando ndk−build. A Figura 10 resume esse procedimento.

3 Resultados e discussão

Todos os resultados obtidos no Xeon Phi utilizaram de 1 a 59 núcleos, já os do Lenovo Vibe K5 utilizaram de 1 a 8 núcleos. No Xeon Phi, todas as medições dos gráficos foram feitas como a mediana de 200 amostras. No Lenovo Vibe K5 também utilizou-se a mediana de 200 execuções, mas como aplicativos rodando de fundo poderiam influenciar os resultados, modificou-se o código para que todas as 200 amostras fossem no máximo 20% maiores do que a mediana e no mínimo 20% menores, caso contrário, as 200 amostras eram obtidas novamente. Desse modo, aplicações de fundo fariam com que o tempo de algumas amostras tivessem um grande desvio da mediana e com que todas as amostras fossem descartadas e obtidas novamente.

Foram gerados no coprocessador Xeon Phi resultados usando dois binários diferentes, um sem otimizações e outro otimizado com alinhamento de dados e vetorização (AVX). Os resultados otimizados são mostrados nas Figuras 1, 2 e 3. Já os resultados não otimizados correspondem às Figuras 4, 5 e 6. Por fim, as Figuras 7 e 8 comparam o otimizado e o não otimizado.

Os resultados obtidos no smartphone da Lenovo Vibe K5 com um Qualcomm Snapdragon 415 são mostrados na Figura 9.

3.1 Xeon Phi

Uma das vantagens dos coprocessadores Intel Xeon Phi em relação aos seus concorrentes, as unidades de processamento gráfico (GPU), é a portabilidade e facilidade de programação [11]. Os resultados dessa pesquisa corroboram essa afirmação, pois através de simples diretivas de programação, como #pragma offload target(mic) inout (dados:length(tamanho)), que serve para rodar seções de código e transferir variáveis da memória do processador para o coprocessador e vice versa, não houve necessidades de realizar grandes mudanças no código. As mudanças necessárias, além da otimização, foram devido a impossibilidade de fazer transferência de variáveis com as diretivas citadas que contenham estruturas de dados complexas (ponteiros de ponteiros ou structs com ponteiros). O suporte da Intel à interface da biblioteca FFTW através de Wrappers com a Intel MKL também contribuiu na portabilidade, fazendo com que todas as linhas de código envolvendo manipulações da transformada de Fourier não fossem reescritas. Para otimizar não foi necessário alterar muitas linhas do código, o alinhamento de dados e vetorização foram feitos como descrito anteriormente.

Os resultados das Figuras 1, 2 e 3 são referentes ao código otimizado, com os parâmetros N igual a 512, 1024 e 2048, respectivamente e $L \in \{60; 120\}$ nas três. Percebe-se que o aumento de L não teve grande influência até aproximadamente 35, 40 e 45 núcleos; que as curvas começaram a se distanciar de um speedup linear a partir de 5, 10 e 20 núcleos; e que para 59 núcleos e $L \in \{60; 120\}$ o speedup foi de $\{34,57; 38,38\}$, $\{42,43; 46,2\}$ e $\{45,76; 49,33\}$, para todos os valores de forma respectiva.

Já os resultados das Figuras 4, 5 e 6 são referentes ao código não otimizado, com os parâmetros N igual a 512, 1024 e 2048, respectivamente e $L \in \{60; 120\}$ nas três. Percebe-

se que o aumento de L não influêncou significativamente até aproximadamente 45, 54 e 55 núcleos, e mesmo depois desses números de núcleos a influência foi pequena; que as curvas começaram a se distanciar de um speedup linear a partir de 20, 25 e 35 núcleos; e que para 59 núcleos e $L \in \{60; 120\}$ o speedup foi de $\{50,63; 52,59\}$, $\{53,67; 54,80\}$ e $\{55,33; 56,50\}$, para todos os valores de forma respectiva.

A Figura 7 mostra o ganho de velocidade do código otimizado em relação ao não otimizado. O eixo horizontal é o número de núcleos, de 1 até 59. O eixo vertical mostra, para cada número de núcleos, o valor $\frac{t_o}{t_n}$, onde t_o é o tempo de execução do código otimizado e t_n o tempo de execução do código não otimizado. Os parêtros foram N=512 e L=60, sendo o cenário com menor carga de trabalho utilizada. Analisando essa figura, observa-se que o ganho foi diminuindo com o aumento de núcleos, começando como 4.4 para 1 núcleo e indo até 3 para 59 núcleos.

Na Figura 8 os eixos representam as mesmas quantidades de forma semelhante. Os parâmetros foram N=2048 e L=120, sendo o cenário com a maior carga de trabalho utilizada. Para 1 núcleo o ganho foi próximo de 3.95 e caiu para 3.44 com 59 núcleos.

Dessa forma constatou-se que o speedup no código não otimizado foi mais suscetível às variações de carga de trabalho do que o código não otimizado, que a curva do código não otimizado foi bastante próxima do speedup linear, principalmente no cenário de maior carga de trabalho estudada e que o código otimizado foi sempre mais rápido no mínimo 3 vezes em relação ao não otimizado.

3.2 Qualcomm Snapdragon 415 - Lenovo Vibe K5

O resultado principal obtido nesse dispositivo móvel foi o estudo da portabilidade, vislumbrando aplicações práticas usando smartphones para realizar um sensoriamento do espectro com rádio cognitivo, visando agregar resoluções à problemática discutida da ineficiência da política fixa de alocação do espectro.

No geral, portar o código para essa plataforma foi simples, a maior dificuldade encontrada foi com a biblioteca FFTW, pois ela utiliza o GNU Build System, enquanto que os programas do Android usam o Android Build System. Esse problema foi resolvido através de análise de erros, mas no pior dos casos, seria possível reescrever o código usando outras bibliotecas de transformadas de Fourier. O Android NDK torna simples o processo de compilar códigos em C e C++ para o Android.

A escalabilidade no processador Qualcomm Snapdragon 415 também foi estudada, como mostra a Figura 9. Ela consiste de 4 curvas com configurações de carga de trabalho diferentes, além da curva de speedup linear. Os parâmetros foram $N \in \{512; 1024\}$ e $L \in \{16; 32\}$. Percebe-se que todas as curvas possuem a mesma tendência, distanciando-se consideravelmente a partir de 2 núcleos do speedup linear. Para 8 núcleos e $\{N; L\} \in \{\{512; 16\}; \{512; 32\}; \{1024; 16\}; \{1024; 32\}\}$, o speedup foi de {4,11;4,24;4,44;4,38}.

4 Conclusão

A partir dos resultados obtidos no coprocessador Xeon Phi, constatou-se que o algoritmo de sensoriamento espectral, o CPD paralelo, possui grande escalabilidade. Evidenciou-se um speedup com tendência linear ao verificar que o código não otimizado executando em 59 núcleos foi 56,5 vezes mais rápido do que com 1 núcleo na maior configuração de carga de trabalho (N igual a 2048 e L igual a 120). Nitidamente, a curva de speedup do código otimizado foi mais distante do caso linear e o aumento na carga de trabalho teve

mais influência (melhorando o speedup) do que no código não otimizado. Apesar disso, o código otimizado foi notavelmente superior em termos de velocidade, executando sempre com velocidades superiores a 3 vezes em relação ao não otimizado. Já no dispositivo móvel com o Qualcomm Snapdragon 415, ficou claro que as 4 curvas de speedup com configurações distintas de carga de trabalho possuíram uma mesma tendência, com um speedup bastante inferior ao linear, sendo pouco superior a 4 em todas as curvas usando 8 núcleos. Conclui-se também que o código possui grande portabilidade, requerendo poucas alterações para funcionar nas duas plataformas.

Como o código no celular não foi otimizado, seria interessante futuramente analisar como otimizações influenciam na escalabilidade e que posteriormente sejam desenvolvidas aplicações que atendam a problemática da ineficiência de alocação do espectro. Para isso, é necessário escrever um programa que capture as amostras dos sinais em smartphones e use essas amostras como entrada do CPD paralelo.

References

[1] Gardner, William A, *Cyclostationarity in communications and signal processing*, STATISTICAL SIGNAL PROCESSING INC YOUNTVILLE CA 1994.

[2] Lima, Arthur DD, et al, *Parallel cyclostationarity-exploiting algorithm for energy-efficient spectrum sensing*, IEICE Transactions on Communications 97.2 (2014): 326-333.

[3] Zhang, Zaichen, and Xiaodan Xu. *Implementation of cyclic periodogram detection on vee for cognitive radio*, Global Mobile Congress (GMC'2007). 2007.

[4] Matteo Frigo and Steven G. Johnson, *The design and implementation of FFTW3*, Proc. IEEE 93 (2), 216–231 (2005).

[5] *FFTW3 Interface to Intel® Math Kernel Library*, https://software.intel.com/en-us/mkl-developer-reference-c-fftw3-interface-to-intel-math-kernel-library, acesso em 19 agos. 2017.

[6] *Data Alignment to Assist Vectorization*, https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization, acesso em 19 agos. 2017.

[7] *Introduction to Intel® Advanced Vector Extensions*, https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions, acesso em 19 agos. 2017.

[8] *Building Open Source libraries with Android NDK*, https://warpedtimes.wordpress.com/2010/02/03/building-open-source-libraries-with-android-ndk/, acesso em 19 agos. 2017.

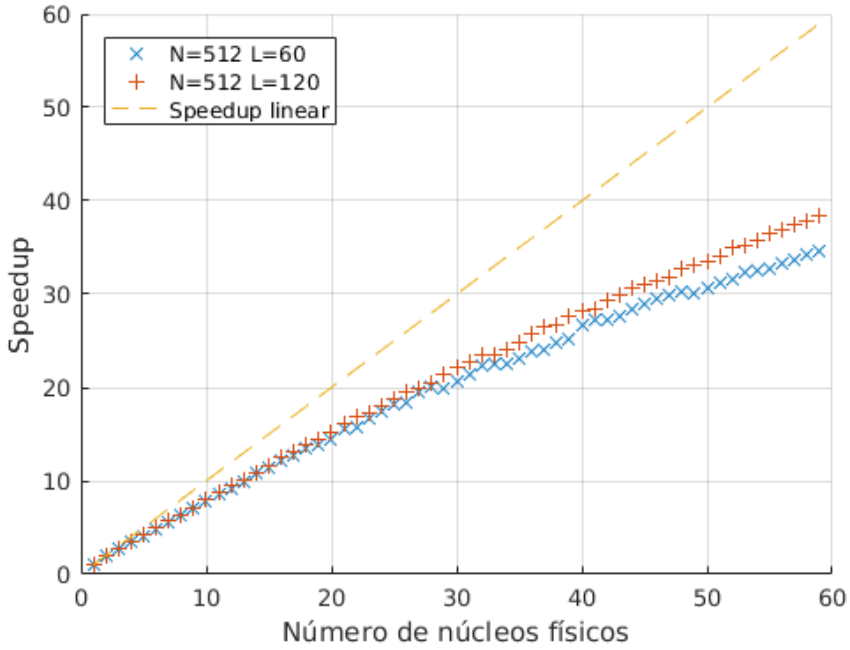
[9] *compiling fftw3 in android ndk*, https://stackoverflow.com/questions/4201911/compiling-fftw3-in-android-ndk, acesso em 19 agos. 2017.

[10] *OpenMP* Thread Affinity Control* https://software.intel.com/en-us/articles/openmp-thread-affinity-control, acesso em 19 agos. 2017.

[11] Reinders, James, *An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors*, Intel Corporation, Santa Clara (2012).

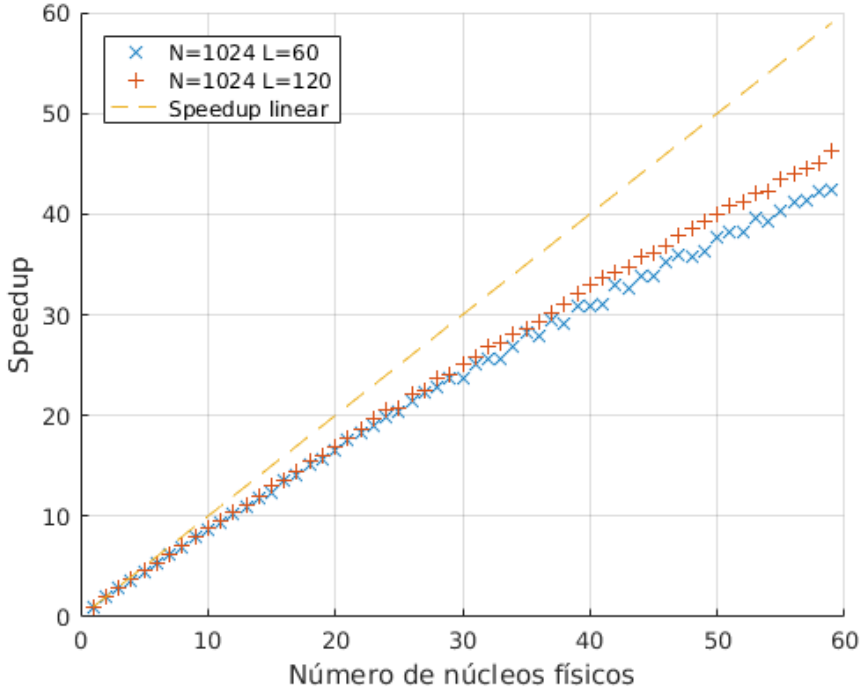
Anexos

Figure 1: Código otimizado, N=512, L=60 e 120, mediana de 200 amostras.



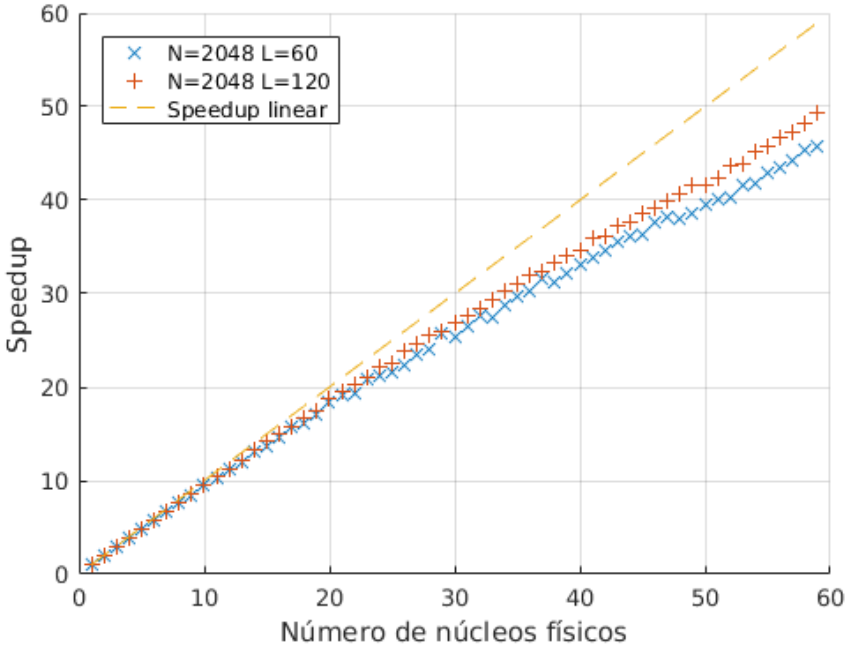
Fonte: Autoria própria, 2017.

Figure 2: Código otimizado, N=1024, L=60 e 120, mediana de 200 amostras.



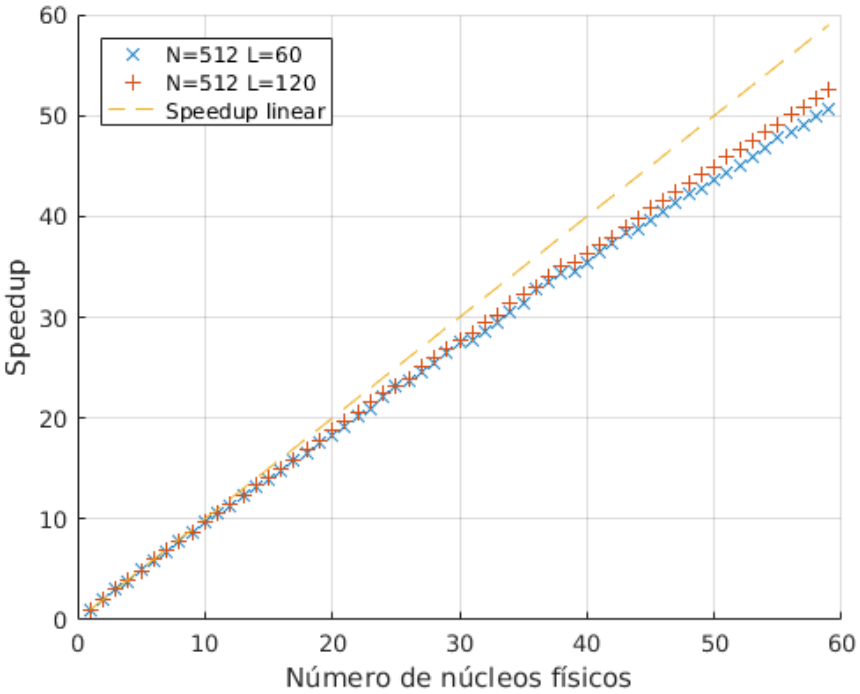
Fonte: Autoria própria, 2017.

Figure 3: Código otimizado, N=2048, L=60 e 120, mediana de 200 amostras.



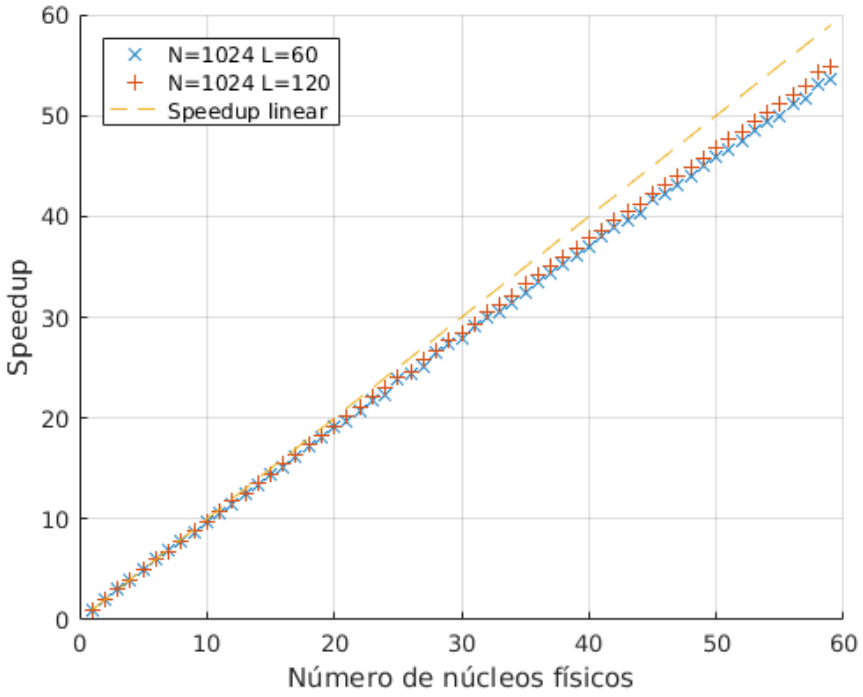
Fonte: Autoria própria, 2017.

Figure 4: Código não otimizado, N=512, L=60 e 120, mediana de 200 amostras.



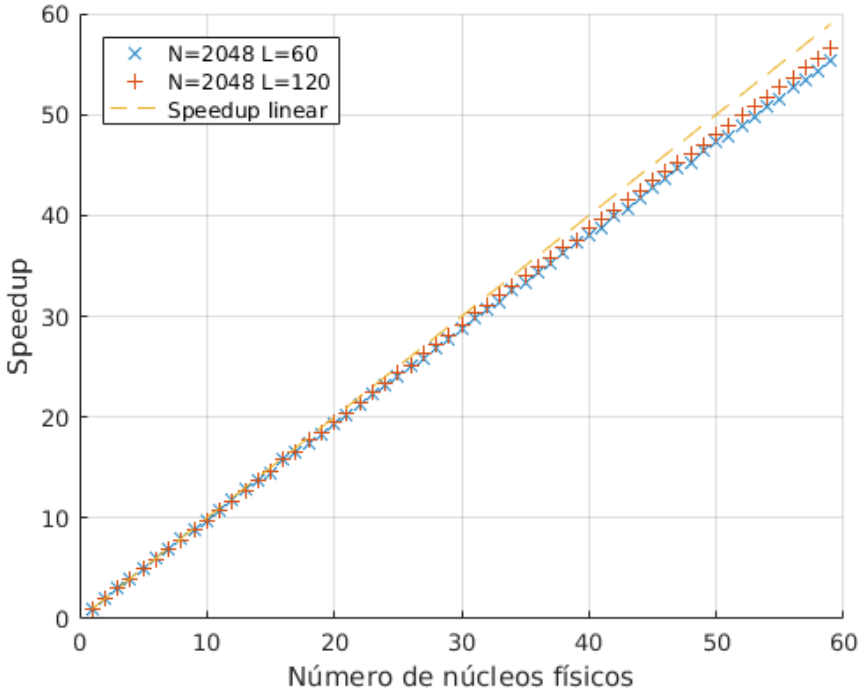
Fonte: Autoria própria, 2017.

Figure 5: Código não otimizado, N=1024, L=60 e 120, mediana de 200 amostras.



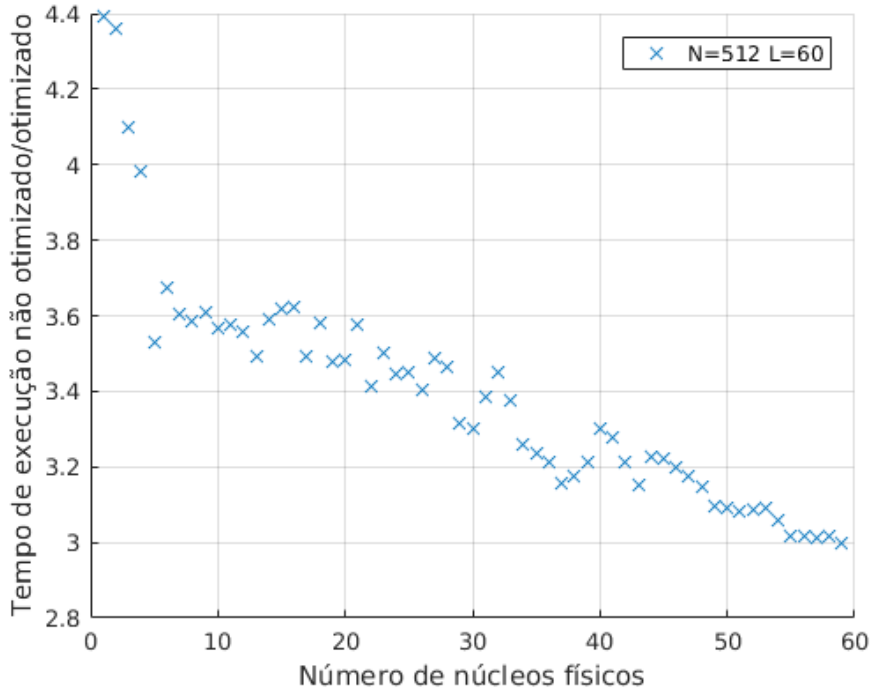
Fonte: Autoria própria, 2017.

Figure 6: Código não otimizado, N=2048, L=60 e 120, mediana de 200 amostras.



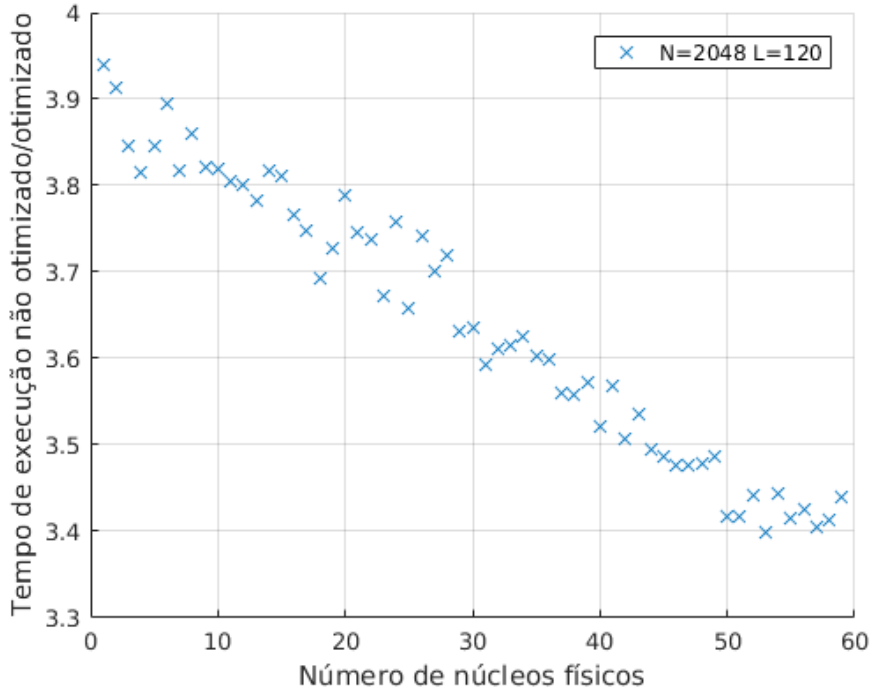
Fonte: Autoria própria, 2017.

Figure 7: Código otimizado/não otimizado, N=512, L=60, mediana de 200 amostras.



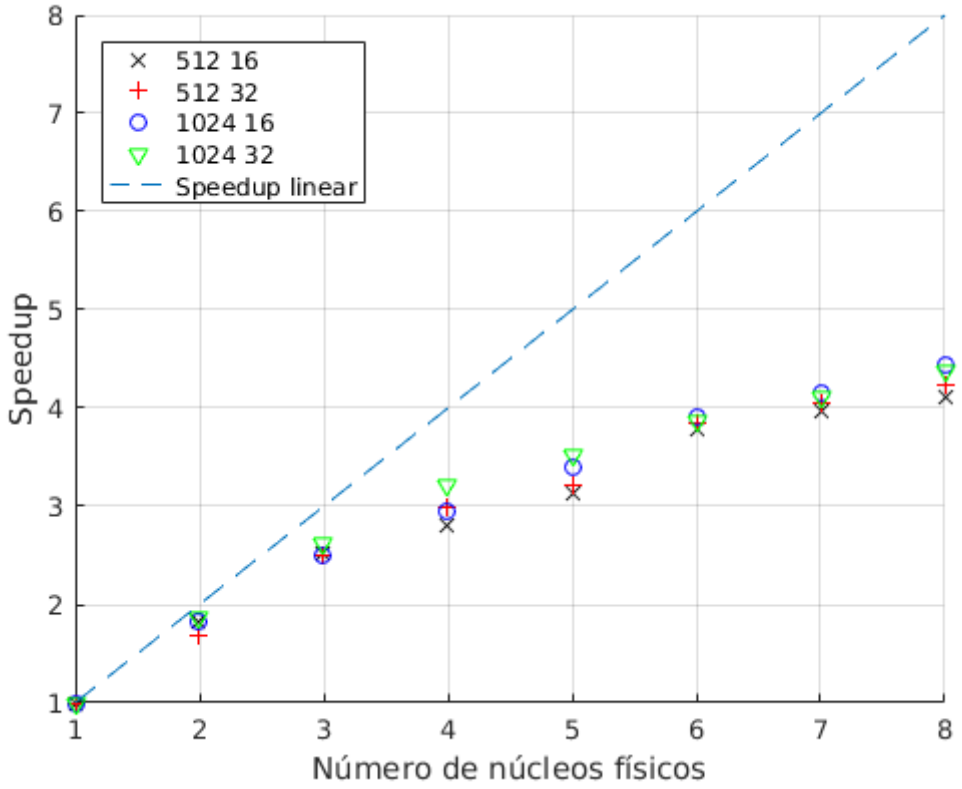
Fonte: Autoria própria, 2017.

Figure 8: Código otimizado/não otimizado, N=2048, L=120, mediana de 200 amostras.



Fonte: Autoria própria, 2017.

Figure 9: Código não otimizado, $N \in \{512; 1024\}$, $L \in \{60; 120\}$, mediana de 200 amostras.



Fonte: Autoria própria, 2017.

Figure 10: Procedimento utilizado.

1	Compilação cruzada da FFTW
1.1	Gerar o config.h com suporte para OpenMP
1.2	Escrever os arquivos Android.mk e Application.mk
2	Compilação cruzada do CPD paralelo
2.1	Reescrever os tipos float complex para fftw_complex
2.2	Escrever o arquivo Android.mk e utilizar o Application.mk em 1.2

Fonte: Autoria própria, 2017.