PERFILAGEM DO PROBLEMA DE RESOLUÇÃO DA EQUAÇÃO DA ONDA POR DIFERENÇAS FINITAS EM COPROCESSADOR XEON PHI

Raphael Fernandes Vilela

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientadores: Jorge Lopes de Souza Leão
Alvaro Luiz Gayoso de Azeredo Coutinho

Rio de Janeiro
Março de 2017

PERFILAGEM DO PROBLEMA DE RESOLUÇÃO DA EQUAÇÃO DA ONDA POR DIFERENÇAS FINITAS EM COPROCESSADOR XEON PHI

Raphael Fernandes Vilela

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

Prof. Jorge Lopes de Souza Leão, Dr.Ing.

Prof. Alvaro Luiz Gayoso de Azeredo Coutinho, D.Sc.

Prof. José Jeronimo Camata, D.Sc.

Prof. Claúdio Márcio do Nascimento Abreu Pereira, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2017

Vilela, Raphael Fernandes

Perfilagem do Problema de Resolução da Equação da Onda por Diferenças Finitas em Coprocessador Xeon Phi/Raphael Fernandes Vilela. – Rio de Janeiro: UFRJ/COPPE, 2017.

XV, 76 p.: il.; 29, 7cm.

Orientadores: Jorge Lopes de Souza Leão

Alvaro Luiz Gayoso de Azeredo Coutinho

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2017.

Referências Bibliográficas: p. 69 – 76.

1. Arquitetura de Computadores. 2. Otimização. 3. Paralelismo. I. Leão, Jorge Lopes de Souza *et al.*. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

À minha avó (in memoriam) e à minha mãe.

Agradecimentos

Agradeço aos colegas da *Petroleum Research and Technology* (PETREC) pelo apoio, especialmente ao Luciano; ao Núcleo de Atendimento a Computação de Alto Desempenho (NACAD) e à Intel pelo suporte técnico e de *hardware*; ao Danilo e aos professores Leão e Álvaro, pelo acolhimento e orientação; à Coordenação de Aperfeiçoamento de Pessoal de Ensino Superior (CAPES) pelo suporte financeiro; à Karen pelo apoio fundamental na reta final.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PERFILAGEM DO PROBLEMA DE RESOLUÇÃO DA EQUAÇÃO DA ONDA POR DIFERENÇAS FINITAS EM COPROCESSADOR XEON PHI

Raphael Fernandes Vilela

Março/2017

Orientadores: Jorge Lopes de Souza Leão
Alvaro Luiz Gayoso de Azeredo Coutinho

Programa: Engenharia Elétrica

A computação científica, que envolve cálculos que duram de várias horas a dias, tem na otimização de código um meio de vital importância. A proposta deste trabalho é validar, através de métricas de performance, a otimização de uma implementação da equação da onda nos meios isotrópicos (ISO) em duas dimensões (2D) e com Isotropia Transversalmente Inclinada (TTI) em três dimensões (3D). Pretende-se medir a quantidade de operações em ponto flutuante por segundo (FLOP/s) e a intensidade aritmética (IA) do código desenvolvido utilizando a tecnologia de *Many Integrated Cores* (MIC), apontando seus limites operacionais e o quão próximo destes limites este código ficou.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PROFILING OF THE FINITE DIFFERENCES WAVE EQUATION RESOLUTION PROBLEM IN XEON PHI COPROCESSOR

Raphael Fernandes Vilela

March/2017

Advisors: Jorge Lopes de Souza Leão
Alvaro Luiz Gayoso de Azeredo Coutinho

Department: Electrical Engineering

Software Optimization is a promising technique to scientific computing applications, which can have calculations that may require several hours or even days. The idea of this work is to validate, using performance metrics, the optimization of a finite differences code that solve the wave equation in isotropic media (ISO) in two dimensional (2D) and also, transverse tilted isotropic (TTI) three dimensional (3D) media. The objective is to measure the number of floating point operations per second (FLOP/s) and the arithmetic intensity of the code developed with the use of the Many Integrated Cores (MIC) technology, pointing its operational limites and how near of these limits is this code.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

L1, L2, L3 Níveis (1, 2 e 3) de Memória *Cache*

Lista de Abreviaturas

CUDA	*Compute Unified Device Architecture*
BW	*Bandwidth* (Largura de Banda)
DP	*Double Precision* (Precisão Simples)
EDP	Equação Diferencial Parcial
FMA	*Fused Multiply-Add* (Operação multiplicação seguida de adição)
FLOP/s	*Floating-Point Operations Per Second* (Número de operações de ponto flutuante por segundo)
HPC	*High-Performance Computing* (Computação de Alto Desempenho)
IA	Intensidade Aritmética
LLC	*Last Level Cache* (*Cache* de último nível)
MDF	Método das Diferenças Finitas
MIC	*Many Integrated Cores* (Núcleos Densamente Integrados)
MPI	*Message Passing Interface* (Interface de Transferência de Mensagens)
NUMA	*Non-Uniform Memory Access* (Acesso não-uniforme à Memória)
RTM	*Reverse Time Migration* (Migração Reversa no Tempo)
SP	*Single Precision* (Precisão Simples)
SIMD	*Single Instruction, Multiple Data* (Instrução Simples, Múltiplos Dados)

Capítulo 1

Introdução

Uma das indústrias que mais movimenta a pesquisa dentro da UFRJ e do estado do Rio de Janeiro é a indústria do petróleo, que demanda um grande aparato tecnológico e mão de obra qualificada. Dentro das diversas áreas de estudo, o segmento do imageamento sísmico permite que os riscos associados à exploração sejam reduzidos, entretanto, como as estruturas são complexas e extensas, o tempo de computação é grande. Como muito desses cálculos são etapas repetitivas de diferentes regiões, a otimização do chamado *core* do código irá, devido ao número de repetições, trazer reduções de horas ou até mesmo dias na execução do programa completo.

A Geofísica é uma ciência multidisciplinar, que faz uso intensivo da Computação de Alto Desempenho, pois os algoritmos da Sísmica, exigem cálculos intensivos e longos. Neste contexto, a otimização dos códigos é uma tarefa importante para reduzir o tempo de obtenção de resultados e, junto a isto, a diminuição dos gastos com o equipamento do processamento, pois diminui os riscos de erros, além de, ao otimizar um código que será executado por menos tempo, permitir que o tempo de aluguel de estrutura computacional seja menor.

A otimização de desempenho exige conhecimentos de programação, dos compiladores e da arquitetura dos processadores e coprocessadores, além do uso de ferramentas de análise de desempenho para identificar gargalos no código. Este estudo permite delimitar os limites da otimização, isto é, até onde é possível diminuir o tempo de execução. Também é interessante que se analisem os custos energéticos, pois, atualmente, representam o maior empecilho ao desenvolvimento da Computação de

Alto Desempenho, devido aos alto valores de frequência de *clock*, que geram mais calor e maior necessidade de refrigeração. Um dos interesses na pesquisa consiste na análise da relação entre desempenho e gasto de energia, para um modelo específico de propagação de onda.

Os estudos realizados neste trabalho concentram-se no trecho correspondente ao cálculo da equação da onda de um programa de migração desenvolvido pelo Laboratório Multidisciplinar de Modelagem (Lab2M) e pela PETREC (*Petroleum Research and Technology*), que corresponde à etapa mais lenta da execução [1]. Os testes foram realizados nas máquinas fornecidas pelo Núcleo Avançado de Computação de Alto Desempenho (NACAD), utilizando a seguinte estrutura : uma *workstation* Xeon E5 codinome *Sandy Bridge* com acelerador Intel® Xeon Phi™e e o Intel® Parallel Studio XE (incluindo o compilador icc e o perfilador VTune™). [1]

A motivação do trabalho reside na investigação do efeito do uso de coprocessadores em cálculos de alto desempenho, que permite à comunidade de computação científica traçar alternativas para a escolha da infraestrutura, levando em conta a expectativa de performance, o custo de equipamento e o custo energético. Além disso, é uma forma de comparar com outras estratégias a capacidade de se obter uma maior relação entre performance e custo energético e vencer a barreira atual dos ExaFLOP/s, isto é, 10^{18} operações de ponto flutuante por segundo, por meio do crescimento do número de processadores por núcleo. [2]

O objetivo do trabalho é estudar o uso de técnicas de otimização, analisando aquelas mais adequadas ao uso de um coprocessador, e medir parâmetros de desempenho para analisar o impacto dessas técnicas no aproveitamento dos recursos da arquitetura Intel Phi. Vale notar que os códigos utilizados neste trabalho foram desenvolvidos pela equipe envolvida no projeto, incluindo o autor desta dissertação. Porém, não é o objetivo do trabalho aprimorar os códigos desenvolvidos, usando-os apenas como objeto de estudo para realizar as perfilagens.

[1] Intel®, VTune™e Xeon Phi™são marcas registradas.

A sistemática desenvolvida neste trabalho procura, através de ferramentas de análise de performance, mais comumente chamadas de perfilagem, determinar o gargalo de programas complexos e entender onde o programa está escrito de forma ineficiente. Estas ferramentas também permitem medir o gasto energético, através de cálculos que envolvem registradores do processador. O modelo de *roofline*[3] é apresentado como forma de indicar os limites da máquina, permitindo comparar, de forma justa, com os resultados em outras arquiteturas.

A dissertação é dividida da seguinte forma: No capítulo 2, é apresentado o problema da modelagem sísmica, além das soluções para a equação da onda estudadas pelo Método das Diferenças Finitas (MDF) e dos parâmetros derivados destas soluções. No capítulo 3, é feita uma revisão bibliográfica, mostrando um breve histórico da computação de alto desempenho, uma lista de técnicas de otimização, uma motivação para o uso do modelo de *roofline*, uma introdução à análise de consumo de energia, apresentando, por fim, uma lista de ferramentas disponíveis para mostrar o comportamento energético dos processadores. Também são apresentadas as arquiteturas disponíveis para trabalho, uma análise da relação entre o algoritmo e o tamanho das memórias *cache* e os limites de desempenho das arquiteturas apresentadas. O capítulo 4 apresenta o uso das ferramentas de perfilagem para obter os dados de número de operações e de potência gasta. O capítulo 5 apresenta os resultados obtidos e como o problema se comporta com o aumento do seu tamanho. Por fim, o capítulo 6 mostra as conclusões deste trabalho. As referências bibliográficas são encontradas ao final desta dissertação.

Capítulo 2

Imageamento Sísmico

2.1 Migração Reversa no Tempo

O mapeamento das camadas do subsolo é feito, principalmente, de duas maneiras: através de métodos baseados em forças eletromagnéticas naturais, e usando fontes de energia artificiais, onde se situam os métodos sísmicos. Eles consistem em inserir uma perturbação no subsolo, conhecida como tiro, a partir do qual uma onda se propaga, sendo difratada e refletida em cada camada onde houver diferença de meio. As frentes de onda refletidas são registradas por diversos geofones, compondo o Sismograma. Os tiros são efetuados tomando como fonte perturbadora uma explosão com dinamite ou vibração com um caminhão *vibroseis*, no caso terrestre, ou canhões de ar, no caso de exploração marítima, conhecida como *offshore*. [1][4][5].

A Migração Reversa no Tempo ou RTM (da sigla em inglês, *Reverse Time Migration*) é uma formulação do comportamento das frentes de onda a partir de um modelo inicial do meio, resolvendo uma equação diferencial parcial de propagação de onda em duas etapas: 1 - resolvendo a propagação direta, a partir da fonte do tiro e 2 - a propagação reversa, a partir dos receptores, usando os valores registrados por eles e calculando em sentido inverso, daí seu nome, "reversa". A equação de onda utilizada varia conforme a exigência da modelagem: inicialmente, pode-se assumir o meio como isotrópico, isto é, sem uma direção preferencial para a propagação. Se for necessário um aumento na complexidade, pode-se adotar o meio verticalmente transverso (VTI, do inglês, *Vertical Transverse Isotropy*) ou transverso incli-

nado (TTI, do inglês, *Tilted Transverse Isotropy*). Apesar da maior complexidade, adota-se com frequência o meio isotrópico como padrão na indústria[4]. Uma extensa revisão bibliográfica sobre as equações anisotrópicas pode ser vista em [5] e uma descrição dos parâmetros da equação de onda VTI e TTI pode ser encontrada em [4]. O objetivo inicial da modelagem do meio é resolver a equação de movimento de cada partícula do meio, dada pela equação (2.1), onde ρ é a massa específica em função da posição; σ representa o tensor de tensões internas, onde os elementos da diagonal principal são as tensões normais e os demais, as tensões cisalhantes; u_i é o vetor de deslocamento das partículas e f_i é a densidade de força do corpo.

$$\rho(\mathbf{x})\frac{\partial^2 u_i}{\partial t^2}(\mathbf{x},t) = \frac{\partial \sigma_{ij}}{\partial x_j}(\mathbf{x},t) + f_i(\mathbf{x},t) \tag{2.1}$$

Como exposto em [4], a grandeza σ é da forma da equação (2.2):

$$\sigma_{ij}(\mathbf{x},t) = C_{ijkl}(\mathbf{x},t)\epsilon_{ij}(\mathbf{x},t) \tag{2.2}$$

onde C_{ijkl} é um tensor de quarta ordem simétrico e positivo definido, que expressa a relação entre as tensões e as deformações no meio elástico. O tensor é apresentado na equação (2.3):

$$C_{ijkl} = C_{\alpha\beta} = \begin{bmatrix} C_{11} & C_{21} & C_{31} & C_{41} & C_{51} & C_{61} \\ C_{21} & C_{22} & C_{32} & C_{42} & C_{52} & C_{62} \\ C_{31} & C_{32} & C_{33} & C_{43} & C_{53} & C_{63} \\ C_{41} & C_{42} & C_{43} & C_{44} & C_{54} & C_{64} \\ C_{51} & C_{52} & C_{53} & C_{54} & C_{55} & C_{65} \\ C_{61} & C_{62} & C_{63} & C_{64} & C_{65} & C_{66} \end{bmatrix} \tag{2.3}$$

Já o tensor deformação ϵ_{kl} é definido pela equação (2.4):

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.4}$$

Combinando as equações (2.1) e (2.4), tem-se a equação da onda para deslocamento em meio elástico, anisotrópico e não-homogêneo:

$$\rho \frac{\partial^2 u_i}{\partial t^2}(\mathbf{x}, t) = \frac{\partial}{\partial x_j} C_{ijkl}(\mathbf{x}, t) \frac{\partial u_k}{\partial x_l}(\mathbf{x}, t) + f_i(\mathbf{x}, t) \tag{2.5}$$

Para isotropia transversa, os 21 parâmetros elásticos são reduzidos a 5 propriedades independentes dadas pela equação (2.6):

$$C_{\alpha\beta} = \begin{bmatrix} C_{11} & C_{11} - 2C_{66} & C_{31} & 0 & 0 & 0 \\ & C_{11} & C_{31} & 0 & 0 & 0 \\ & & C_{33} & 0 & 0 & 0 \\ & & & C_{44} & 0 & 0 \\ & & & & C_{44} & 0 \\ & & & & & C_{66} \end{bmatrix} \tag{2.6}$$

Utilizando os parâmetros de Thomson[6], definem-se:

$$V_p \equiv \sqrt{\frac{C_{33}}{\rho}}$$

$$V_s \equiv \sqrt{\frac{C_{55}}{\rho}}$$

$$\epsilon \equiv \frac{C_{11} - C_{33}}{2C_{33}}$$

$$\delta \equiv \frac{(C_{13} + C_{55})^2 - (C_{33} - C_{55})^2}{2C_{33}(C_{33} - C_{55})}$$

$$\gamma \equiv \frac{(C_{66} - C_{44}}{2C_{44}}$$

A decomposição da frente de onda se dá entre ondas-P - que indicam a intensidade de pressão da onda e são medidas diretamente por sismógrafos e hidrofones [4] - e, ondas-S (do inglês, *shear*, cisalhamento), tipos de onda que atravessam os objetos, de maneira que o movimento é perpendicular à direção de propagação e medidas por cabos oceânicos (OBC, do inglês, *Ocean Bottle Cable*)[4]. Nas equações acima, o tensor de deformação ϵ representa a medida da diferença entre a velocidade de onda-P vertical e a horizontal e é chamado de anisotropia da onda-P; γ é a quantidade de anisotropia da onda-S; e δ é a anisotropia quasivertical. V_p representa a velocidade da onda de pressão e V_s, da onda de cisalhamento.

Como já foi mencionado acima, as principais aproximações do meio elástico são o Isotrópico, o VTI e o TTI, dos quais, para este trabalho, foram utilizados os meios Isotrópico e TTI. Para o meio Isotrópico, somente a frente de onda-P é levada em conta e a preferência de propagação é a mesma para qualquer direção, simplificando a equação da onda na equação (2.7), onde c é a velocidade de propagação da onda no meio.

$$\frac{1}{c^2}\frac{\partial u^2}{\partial t^2} = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) u \tag{2.7}$$

A modelagem mais complexa é a VTI, que leva em conta direções preferenciais de propagação, onde as razões entre as velocidades de propagação das ondas-P dos eixos são ϵ para $\frac{v_x}{v_z}$ e δ para $\frac{v_y}{v_z}$, sendo v_{pz} a velocidade vertical da onda-P, v_{px}, a velocidade horizontal e v_{pn}, a velocidade de translação da onda-P. A modelagem VTI é dada pela equação (2.8), onde P é a frente de pressão da onda, e utiliza o campo auxiliar Q da equação (2.9).

$$\frac{\partial^2 P}{\partial t^2} = v_{px}^2\left(\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2}\right) + v_{pz}v_{pn}\frac{\partial^2 Q}{\partial z^2} \tag{2.8}$$

$$\frac{\partial^2 Q}{\partial t^2} = v_{pz}v_{pn}\left(\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2}\right) + v_{pz}^2\frac{\partial^2 Q}{\partial z^2} \tag{2.9}$$

Aumentando a complexidade da modelagem e, por consequência, a intensidade de cálculo, existe a modelagem TTI, introduzindo os ângulos de deflexão θ entre o eixo x e a velocidade de propagação horizontal, e ϕ entre o eixo z e a velocidade de propagação vertical.

$$\frac{\partial^2 P}{\partial t^2} = v_{px}\left(\hat{H}_1 + \hat{H}_2\right) P + v_{pz}v_{pn}\hat{H}_3 Q \tag{2.10}$$

$$\frac{\partial^2 Q}{\partial t^2} = v_{pz}v_{pn}\left(\hat{H}_1 + \hat{H}_2\right) P + v_{pz}^2\hat{H}_3 Q \tag{2.11}$$

O termo v_{pz} é considerado um parâmetro do modelo sísmico medido previamente; v_{px} e v_{pn} são calculados de acordo com os parâmetros da anisotropia do meio (ϕ e θ) conforme as equações (2.15) e (2.16)[4], e são apresentados na Figura 2.1 (adaptada de [4]), na qual também é possível observar como a direção preferencial de

propagação varia conforme os eixos, cada qual possuindo uma velocidade de propagação diferente. Para facilitar a escrita, foram definidos os operadores $\hat{H}_1$, $\hat{H}_2$ e $\hat{H}_3$, dados pelas equações (2.12), (2.13) e (2.14).

$$\hat{H}_1 = \begin{array}{l} \frac{\partial^2}{\partial x^2} cos^2\theta cos^2\phi + \frac{\partial^2}{\partial y^2} cos^2\theta sen^2\phi + \frac{\partial^2}{\partial z^2} sen^2\theta + \\ \frac{\partial^2}{\partial x \partial y} cos^2\theta sen2\phi + \frac{\partial^2}{\partial x \partial z} sen2\theta cos\phi + \frac{\partial^2}{\partial u \partial z} sen2\theta sen\phi \end{array} \tag{2.12}$$

$$\hat{H}_2 = \frac{\partial^2}{\partial x^2} sen^2\phi + \frac{\partial^2}{\partial y^2} cos^2\phi + \frac{\partial^2}{\partial z^2} sen2\phi \tag{2.13}$$

$$\hat{H}_3 = \begin{array}{l} \frac{\partial^2}{\partial x^2} sen^2\theta cos^2\phi + \frac{\partial^2}{\partial y^2} sen^2\theta sen^2\phi + \frac{\partial^2}{\partial z^2} cos^2\theta + \\ \frac{\partial^2}{\partial x \partial y} sen^2\theta sen2\phi - \frac{\partial^2}{\partial x \partial z} sen2\theta cos\phi - \frac{\partial^2}{\partial u \partial z} sen2\theta sen\phi \end{array} \tag{2.14}$$

$$v_{px} = v_{pz}\sqrt{1 + 2\epsilon} \tag{2.15}$$

$$v_{pn} = v_{pz}\sqrt{1 + 2\delta} \tag{2.16}$$

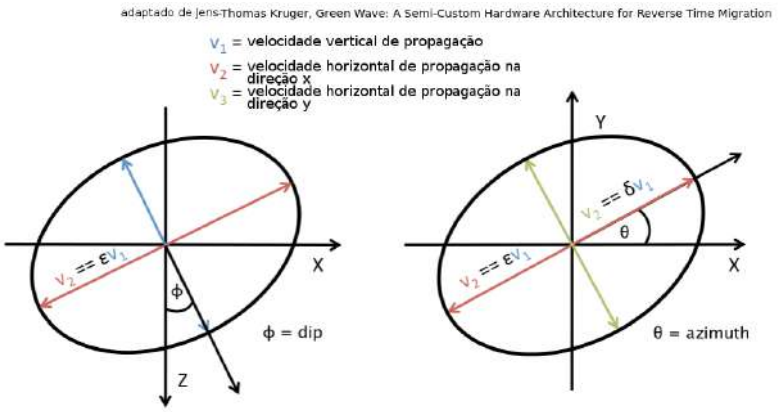

Figura 2.1: Descrição dos parâmetros de anisotropia (adaptado de [4])

Dentro das várias etapas da RTM, a solução da equação da onda é a etapa que mais consome recursos computacionais [1][4][7], motivo que será explicado na próxima seção. A execução do perfilador VTune [8], apresentada na Figura 2.2, confirma a importância da computação da equação da onda frente ao resto do algoritmo, portanto, os esforços da otimização se concentram na equação da onda. Nela, é possível ver o programa desenvolvido separado por seus diferentes módulos e cada módulo,

por sua vez, separado por região do código, ordenados por tempo de execução. Em [7], apresenta-se uma otimização da RTM com 27% do pico de performance, o que mostra que o pico de performance da RTM otimizada pode ficar inerentemente distante do pico de performance do *hardware* utilizado.

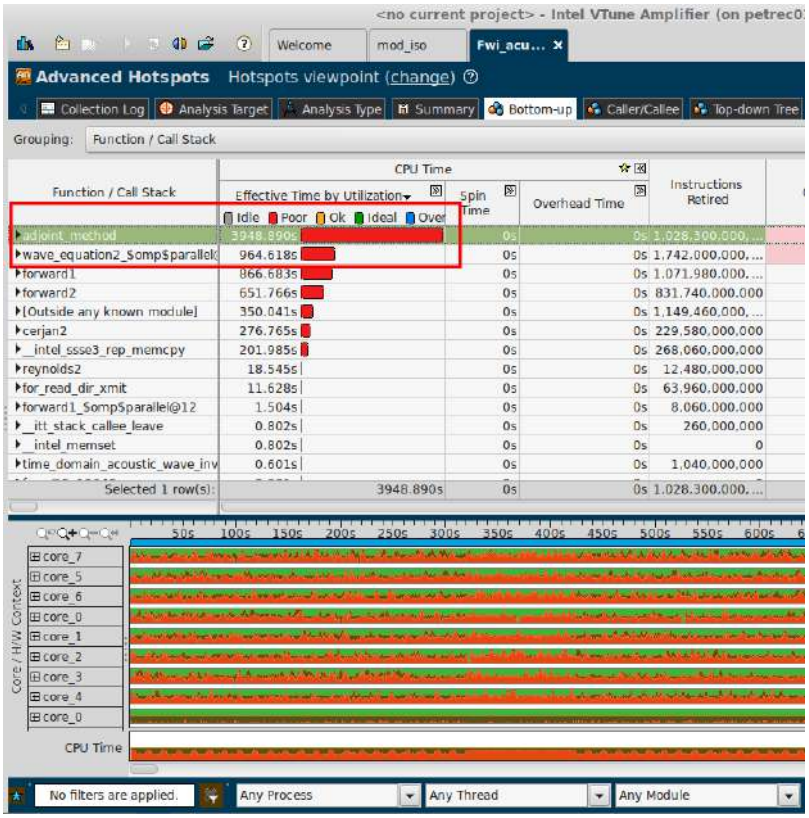

Figura 2.2: Perfilagem de *hotspots* da RTM utilizando o VTune, ressaltando os módulos que chamam o cálculo da equação da onda

2.2 Geração do *Stencil*

A solução da equação da onda usualmente é feita de forma numérica através do Método das Diferenças Finitas (MDF), calculando a função para cada ponto no espaço que, por sua vez, é discretizado, tornando-se um paralelepípedo (para o caso tridimensional) composto por vários pontos, conforme a Figura 2.3. Para este trabalho, considera-se que cada ponto do espaço discretizado é igualmente espaçado por h, qualquer que seja a direção. Para o cálculo de cada ponto deste paralelepípedo, são necessários os valores da função de diversos vizinhos, compondo o que se chama de *stencil*, e chama-se ordem o número de vizinhos necessários para sua computação

em uma direção, como já foi exemplificado na Figura 3.1.

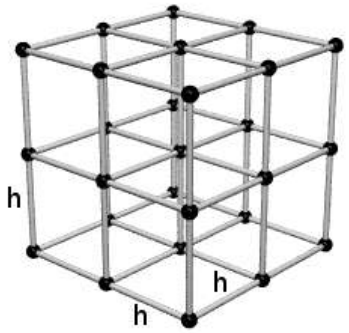

Figura 2.3: Espaço tridimensional discretizado

O MDF pode ser aplicado usando a expansão em série de Taylor, descrita a seguir. Seja ϕ uma função de três variáveis $\phi(x, y, z)$. Sua expansão em série de Taylor na direção x é dada na equação (2.17):

$$\phi(x+h, y, z) = \phi(x, y, z) + h\frac{\partial\phi(x, y, z)}{\partial x} + \frac{h^2}{2}\frac{\partial^2\phi(x, y, z)}{\partial x^2} + ... \tag{2.17}$$

Para a expansão para valores anteriores, pode-se reescrever (2.17), obtendo a equação (2.18):

$$\phi(x-h, y, z) = \phi(x, y, z) - h\frac{\partial\phi(x, y, z)}{\partial x} + \frac{h^2}{2}\frac{\partial^2\phi(x, y, z)}{\partial x^2} - ... \tag{2.18}$$

Subtraindo (2.17) de (2.18) e desprezando os termos de derivada de ordem maior que 3, obtemos a aproximação de segunda ordem para a derivada, conforme a equação (2.19).

$$\phi(x+h, y, z) - \phi(x-h, y, z) = 2h\frac{\partial\phi(x, y, z)}{\partial x} + ...$$

$$\frac{\partial\phi(x, y, z)}{\partial x} \approx \frac{\phi(x+h, y, z) - \phi(x-h, y, z)}{2h} \tag{2.19}$$

A segunda derivada de segunda ordem é obtida através da soma das equações (2.17) e (2.18).

$$\phi(x+h, y, z) + \phi(x-h, y, z) = 2\phi(x, y, z) + 2\frac{h^2}{2}\frac{\partial^2\phi(x, y, z)}{\partial x^2} + ...$$

$$\frac{\partial^2\phi(x, y, z)}{\partial x^2} \approx -\frac{2}{h^2}\phi(x, y, z) + \frac{1}{h^2}\left[\phi(x+h, y, z) + \phi(x-h, y, z)\right] \tag{2.20}$$

No procedimento acima, foram obtidas as expressões discretas para a derivada primeira e segunda, gerando um *stencil* de ordem 2, cuja distância entre os pontos vizinhos é h. A obtenção dos coeficientes de ordem maior (quarta ordem em diante) requer outras ferramentas, como a Transformada de Fourier e o Método Pseudoespectral[9]. Aplicando a Transformada de Fourier à generalização da derivada discreta (equação (2.21)), obtém-se os coeficientes usando a interpolação polinomial de Lagrange[9]. Seja ϕ uma função de várias variáveis, incluindo x, $\phi_n = \phi(x \pm nh)$, h é o espaçamento da malha, c_n são os coeficientes referentes a cada ponto do *stencil* da fórmula do MDF e N é um número inteiro par que descreve a ordem do *stencil*.

$$\left.\frac{\partial^2 \phi(x)}{\partial x^2}\right|_{x=0} = \frac{1}{h^2}\left[c_0\phi_0 + \sum_{n=1}^{N/2} c_n\left(\phi_n + \phi_{-n}\right)\right] \tag{2.21}$$

Aplicando a Transformada de Fourier à equação (2.21), tem-se:

$$-(k_x h)^2 = c_0 + 2\sum_{n=1}^{N/2} c_n \cos(nk_x h) \tag{2.22}$$

onde k_x é o número de onda na direção x. Expandindo o cosseno através da série de Taylor até a N-ésima ordem e igualando os coeficientes da potência $k_x h$, obtém-se o seguinte sistema de equações lineares do qual pode-se obter c_n calculando a inversa da primeira matriz em (2.23) e tomando sua segunda coluna (para isso, basta aplicar a inversa em ambos os lados da equação).

$$\begin{bmatrix} \frac{1}{2} & 1^0 & 2^0 & \cdots & \left(\frac{N}{2}\right)^0 \\ 0 & 1^2 & 2^2 & \cdots & \left(\frac{N}{2}\right)^2 \\ 0 & 1^4 & 2^4 & \cdots & \left(\frac{N}{2}\right)^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1^N & 2^N & \cdots & \left(\frac{N}{2}\right)^N \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N/2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.23}$$

Sendo assim, os coeficientes do MDF Convencional para derivada segunda com N $\in$ [2, 4, 8, 10, 16] são apresentados na Tabela 2.2. O procedimento é repetido para a expansão em seno que caracteriza a derivada segunda (função ímpar, como pode ser visto na equação (2.18)), obtendo a Tabela 2.1.

Tabela 2.1: Coeficientes da primeira derivada para diferentes ordens do MDF

ordem	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
2	$\frac{1}{2}$							
4	$\frac{2}{3}$	$-\frac{1}{12}$						
6	$\frac{3}{4}$	$-\frac{3}{20}$	$\frac{1}{60}$					
8	$\frac{4}{5}$	$-\frac{1}{5}$	$\frac{4}{105}$	$-\frac{1}{280}$				
10	$\frac{5}{6}$	$-\frac{5}{21}$	$\frac{5}{84}$	$-\frac{5}{504}$	$\frac{1}{1260}$			
12	$\frac{6}{7}$	$-\frac{15}{56}$	$\frac{5}{63}$	$-\frac{1}{56}$	$\frac{1}{385}$	$-\frac{1}{5544}$		
14	$\frac{7}{8}$	$-\frac{7}{24}$	$\frac{7}{72}$	$-\frac{7}{264}$	$\frac{7}{1320}$	$-\frac{7}{10296}$	$\frac{1}{24024}$	
16	$\frac{8}{9}$	$-\frac{14}{45}$	$\frac{56}{495}$	$-\frac{7}{198}$	$\frac{56}{6435}$	$-\frac{2}{1287}$	$\frac{8}{45045}$	$-\frac{1}{102960}$

Tabela 2.2: Coeficientes da segunda derivada para diferentes ordens do MDF

ordem	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
2	-2	1							
4	$-\frac{5}{2}$	$\frac{4}{3}$	$-\frac{1}{12}$						
6	$-\frac{49}{18}$	$\frac{3}{2}$	$-\frac{3}{20}$	$\frac{1}{90}$					
8	$-\frac{205}{72}$	$\frac{8}{5}$	$-\frac{1}{5}$	$\frac{8}{315}$	$-\frac{1}{560}$				
10	$-\frac{5269}{1800}$	$\frac{5}{3}$	$-\frac{5}{21}$	$\frac{5}{126}$	$-\frac{5}{1008}$	$\frac{1}{3150}$			
12	$-\frac{5369}{1800}$	$\frac{12}{7}$	$-\frac{15}{56}$	$\frac{10}{189}$	$-\frac{1}{112}$	$\frac{2}{1925}$	$-\frac{1}{16632}$		
14	$-\frac{266681}{88200}$	$\frac{7}{4}$	$-\frac{7}{24}$	$\frac{7}{108}$	$-\frac{7}{528}$	$\frac{7}{3300}$	$-\frac{7}{30888}$	$\frac{1}{84084}$	
16	$-\frac{1077749}{352800}$	$\frac{16}{9}$	$-\frac{14}{45}$	$\frac{112}{1485}$	$-\frac{7}{396}$	$\frac{112}{32175}$	$-\frac{2}{3861}$	$\frac{16}{315315}$	$-\frac{1}{411840}$

O índice do coeficiente das Tabelas 2.1 e 2.2 indica a quantas unidades do *stencil* (ou seja, o número de passos de h) a função é avaliada em relação a um elemento central, como ilustrado pela Figura 2.4, e devem ser multiplicados por uma potência de h, conforme as equações (2.24) e (2.25).

$$\frac{\partial \phi(x,y,z)}{\partial x} = \sum_{k=1}^{N/2} b_k \frac{1}{h}(\phi(x+kh,y,z) - \phi(x-kh,y,z)) \tag{2.24}$$

$$\frac{\partial^2 \phi(x,y,z)}{\partial x^2} = a_0 \frac{1}{h^2}\phi(x,y,z) + \sum_{k=1}^{N/2} a_k \frac{1}{h^2}(\phi(x+kh,y,z) + \phi(x-kh,y,z)) \tag{2.25}$$

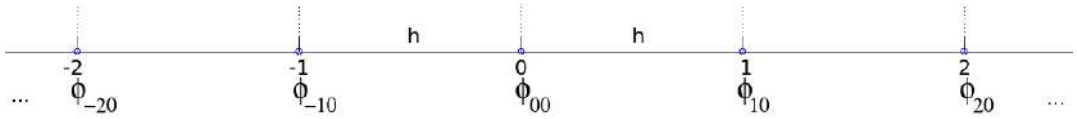

Figura 2.4: Geometria do *stencil* na direção x

A metodologia para obtenção dos coeficientes da derivada primeira e segunda pode ser resumida através dos algoritmos a seguir, conforme [9]. As matrizes das equações (2.26) e (2.27) serão usadas nas equações (2.28) e (2.29).

$$A_1 = \begin{bmatrix} \frac{1}{2} & 1^0 & 2^0 & ... & \left(\frac{N}{2}\right)^0 \\ 0 & 1^2 & 2^2 & ... & \left(\frac{N}{2}\right)^2 \\ 0 & 1^4 & 2^4 & ... & \left(\frac{N}{2}\right)^4 \\ & & ... & & \\ 0 & 1^N & 2^N & ... & \left(\frac{N}{2}\right)^N \end{bmatrix} \tag{2.26}$$

$$A_2 = \begin{bmatrix} 1^1 & 2^1 & 3^1 & ... & \left(\frac{N}{2}\right)^1 \\ 1^3 & 2^3 & 3^3 & ... & \left(\frac{N}{2}\right)^3 \\ 1^5 & 2^5 & 3^5 & ... & \left(\frac{N}{2}\right)^5 \\ & & ... & & \\ 1^{(N-1)} & 2^{(N-1)} & 3^{(N-1)} & ... & \left(\frac{N}{2}\right)^{(N-1)} \end{bmatrix} \tag{2.27}$$

$$B = A_1^{-1} \begin{bmatrix} \frac{1}{2} \\ 0 \\ ... \\ 0 \end{bmatrix} \tag{2.28}$$

$$C = A_2^{-1} \begin{bmatrix} 0 \\ 1 \\ 0 \\ ... \\ 0 \end{bmatrix} \tag{2.29}$$

Foi mostrado, em um estudo comparativo entre diversas ordens do *stencil* isotrópico[10], que a décima sexta ordem espacial obtém a melhor relação de benefício com maior precisão em relação à perda em tempo de execução. Porém, considerando a maior complexidade do *stencil* TTI, optou-se pela **oitava** ordem,

a fim de facilitar o entendimento das otimizações e a escrita do código, sem que haja prejuízo no resultado final, além de ser a ordem usada como padrão dentro da indústria sísmica [4].

2.3 Critérios para não dispersão e estabilidade

Para controlar a dispersão numérica na modelagem, provocada pelo erro intrínseco da solução do MDF, existe uma relação entre a menor velocidade do meio contínuo C_{min}, o parâmetro G, que representa o número de pontos necessários para representar o menor comprimento de onda da malha λ_{min}e a frequência de corte (f_{corte}), que limita o máximo valor do espaçamento da malha de forma a não ter excessiva dispersão de energia [11], essa relação é dada por:

$$h \leq \frac{\lambda_{min}}{G} = \frac{C_{min}}{G f_{corte}}, \tag{2.30}$$

Para o critério de estabilidade, também existe uma relação para controle dos valores dos intervalos do tempo de amostragem, evitando que o sistema se torne numericamente instável, como mostrado na equação (2.31):

$$dt \leq \frac{h}{\beta C_{max}}, \tag{2.31}$$

onde dt é o intervalo de tempo, C_{max} é a velocidade máxima do meio contínuo e o parâmetro β determina quantos intervalos de tempo serão necessários para que a frente de onda percorra uma distância equivalente ao espaçamento entre os pontos da malha, considerando a maior velocidade de propagação [12]. Empiricamente, o valor ótimo encontrado para β é 5[13].

2.4 Fonte sísmica

O termo fonte utilizado nesta dissertação para a geração do sinal sísmico é denominada wavelet de Ricker [14] cuja forma é escrita como segue:

$$f(t) = \left[1 - 2\pi \left(\pi f_c t_d\right)^2\right] e^{-\pi(\pi f_c t_d)^2}, \tag{2.32}$$

onde $t_d = n\Delta t - t_f$ representa uma translação temporal da fonte no tempo de modo que a wavelet inicie na origem, e sendo t_f o período da função Gaussiana, dado por:

$$t_f = \frac{2\sqrt{\pi}}{f_c} \tag{2.33}$$

e f_c a frequência central da fonte, que é escrita em termos da maior frequência contida no espectro da função fonte, f_{corte}, dada por:

$$f_c = \frac{f_{corte}}{3\sqrt{\pi}}. \tag{2.34}$$

A Figura 2.5 ilustra um exemplo da função fonte no domínio do tempo e seu correspondente espectro de frequência.

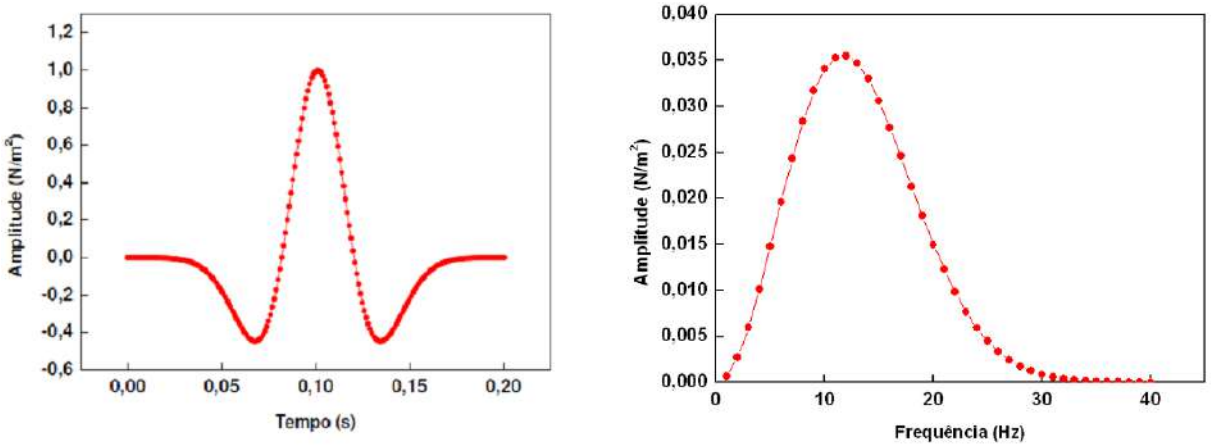

Figura 2.5: Função fonte e seu espectro de frequência para $f_{corte} = 40$Hz.

2.5 Bordas não-reflexivas

A fim de simular problemas com domínios infinitos ou semi-infinitos visando garantir um correto truncamento do modelo numérico sem o surgimento de reflexões artificiais nos respectivos bordos, descreve-se alguns artifícios comumente empregados nas simulações geofísicas.

Um artifício bastante utilizado, devido à introdução de contornos artificiais, é aquela denominada por condição de contorno não-reflexiva ou condição de contorno absorsora [15]. Assim, pode-se dizer que as condições de contorno não-reflexivas visam absorver a energia da frente de ondas quando esta incide no contorno.

O modo mais simples de se derivar uma condição de contorno não-reflexiva é através da fatoração da equação da frente de onda P, definida pelas equações (2.8) e (2.10), onde c é a velocidade da onda no meio, tal que

$$\frac{\partial^2 P}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 P}{\partial t^2} = \left(\frac{\partial}{\partial x} + \frac{1}{c}\frac{\partial}{\partial t}\right)\left(\frac{\partial}{\partial x} - \frac{1}{c}\frac{\partial}{\partial t}\right) P = 0. \tag{2.35}$$

O primeiro termo do segundo membro da equação (2.35) é relativo a ondas viajando no sentido negativo do eixo x e o segundo termo, no sentido positivo. Assim, para eliminar as reflexões no contorno esquerdo e direito basta empregar, respectivamente as equações (2.36) e (2.37) abaixo:

$$\left(\frac{\partial}{\partial x} + \frac{1}{c}\frac{\partial}{\partial t}\right) P = 0, \tag{2.36}$$

$$\left(\frac{\partial}{\partial x} - \frac{1}{c}\frac{\partial}{\partial t}\right) P = 0. \tag{2.37}$$

Uma forma de reduzir as reflexões espúrias geradas pelo truncamento dos contornos, é aplicar camadas de amortecimento (*damping zones*) numérico para reduzir a intensidade da onda antes desta incidir sobre o contorno [16].

O procedimento é feito aplicando a expressão (2.38) a cada ponto da malha pertencente à camada de amortecimento (Figura 2.6) durante a propagação do campo de ondas. A redução da amplitude no ponto (i, j) é feita graças aos fatores multiplicativos gerados pela expressão (2.39).

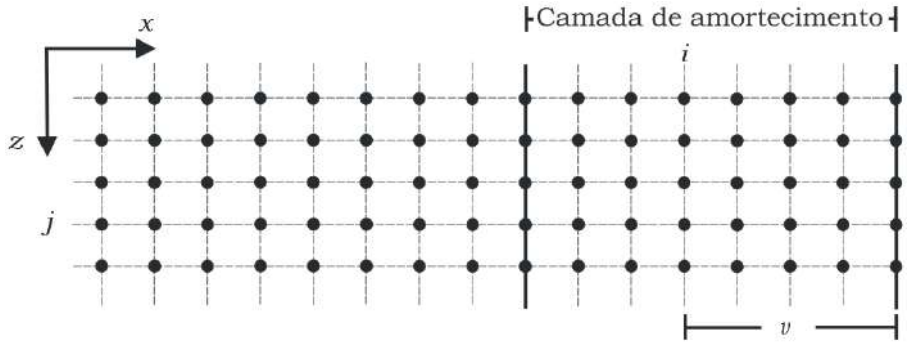

Figura 2.6: Camada de amortecimento do lado direito da malha ilustrando o ponto de aplicação do amortecimento (retirado de [17])

$$P^n_{i,j} = f_{mu}(\upsilon) P^n_{i,j}, \tag{2.38}$$

sendo

$$f_{mu}(\upsilon) = e^{-[fat(n_{amort}-\upsilon)]^2}, \tag{2.39}$$

onde f_{mu} é o fator multiplicativo para atenuar o campo de pressão, υ é um índice que denota a distância de determinado ponto em relação ao contorno do modelo, fat é o fator de amortecimento referente à intensidade da redução de amplitude da pressão que se deseja e n_{amort} é o número de pontos que constitui a camada de amortecimento.

Capítulo 3

Paralelização e Otimização do Código

3.1 Histórico

Em 1967[18], propôs-se a Lei de Amdahl, afirmando que, para uma dada aplicação a ser paralelizada, o máximo ganho obtido pela paralelização seria limitado pela região serial do programa, de tal maneira que o aumento do paralelismo não conseguiria trazer benefício suficiente para compensar a porção serial. Este estudo trouxe muito ceticismo ao desenvolvimento da computação paralela, entretanto, em 1989, foi proposta uma nova interpretação para a Lei de Amdahl, levando em conta a escalabilidade do problema de acordo com o seu tamanho, isto é, um problema maior exige um poder computacional proporcionalmente maior. Isto seria verdade na área paralelizável, mas não na parte serial, portanto, quanto maior o problema, menor a contribuição da parte serial para o problema, resultando na Lei de Gustafson[19]. A partir desta formulação, o desenvolvimento da computação paralela ampliou-se[20], culminando na criação, em 1993 [21], na lista do TOP500, que, através de um *benchmark* (LINPACK e, mais recentemente, HPL), aponta os computadores com aplicação científica de maior poder, calculando a quantidade de operações de ponto flutuante por segundo - FLOP/s - que é capaz de realizar, ao executar tal *benchmark*. A Tabela 3.1[21] apresenta um trecho da lista mais recente, divulgada em Novembro de 2016. A Tabela mostra as instituições responsáveis, destacando-se as posições dos sistemas brasileiros. Entre as aplicações apontadas, destacam-

se: Energia, Geofísica, Biofísica, Meteorologia, Mudanças Climáticas, Cosmologia, Segurança da Informação, Combate ao Terrorismo, Química Computacional, Visualização Científica, Ciência dos Materiais, Mecânica dos Fluidos[22][23][24], entre outras.

Tabela 3.1: 5 primeiras posições do Top500 e a presença brasileira na lista

Posição	País	Ambiente	PetaFLOP/s	Potência dissipada (kW)
1	China	Pesquisa	93,0	15371
2	China	Pesquisa	33,9	17808
3	E.U.A.	Pesquisa	17,6	8209
4	E.U.A.	Pesquisa	17,1	7890
5	E.U.A.	Pesquisa	14,0	3939
364	Brasil	Acadêmico	0,457	371
433	Brasil	Pesquisa	0,405	2580
476	Brasil	Acadêmico	0,363	859

A partir dos anos 2000[25], começou a haver uma preocupação significativa pelo consumo de energia, surgindo a lista do Green500, que pondera o poder computacional, dividindo a métrica analisada pelo Top500 (FLOP/s) pela potência dissipada, isto é, a energia consumida. Também, por conta disso, houve maior investimento na integração de núcleos em um mesmo *chip*, com o surgimento da Xeon Phi e do CUDA (*Compute Unified Device Architecture*). Neste contexto, a busca pela "exaescala" (do inglês, *exascale*), isto é, um computador capaz de realizar 10^{18} operações de ponto flutuante por segundo (1 ExaFLOP/s), também está limitada pelo consumo de energia. Para fins de comparação, de acordo com a lista do Green500 de Novembro de 2016, o sistema mais eficiente tem aproximadamente 6674 MFLOP/s/W[26], portanto, para alcançar 1 ExaFLOP/s, mantendo a mesma proporção, seriam necessários 149 MW, o que está dentro de escala de grandeza da usina de Angra 1, que é capaz de gerar 640 MW e atendeu, em 2011, a 11% da população do estado do Rio de Janeiro[27].

3.2 Técnicas de Otimização

3.2.1 Introdução

Diversos problemas com enormes aplicações são soluções de Equações Diferenciais Parciais (EDP) e um desses problemas é estudado pela Geofísica, para imageamento de subsuperfícies, utilizando a Equação da Onda como parte central do algoritmo. Uma das soluções para esta EDP utiliza o Método das Diferenças Finitas (MDF).

Para a Equação da Onda, em particular, este método permite que o campo de propagação seja calculado em uma coordenada do espaço discretizado em função de um elemento central e seus vizinhos, gerando o *stencil* , que é modelado, computacionalmente, como uma série de dois ou três *loops* aninhados, dependendo se o problema for bidimensional (2D) ou tridimensional (3D). Esta ideia é ilustrada pela Figura 3.1 e pelo Código 3.1. O cálculo do elemento central $C[0, 0, 0]$ depende do valor de seus vizinhos mais próximos nas direções x, y e z; na Figura 3.1, o elemento central de um *stencil* isotrópico (isto é, sem derivadas espaciais cruzadas) de sexta ordem depende do seu valor anterior e do valor de 18 vizinhos, entre eles, são destacados os vizinhos da posição $[x, y - 1, z]$ e $[x, y, z + 3]$, além do elemento central $[x, y, z]$.

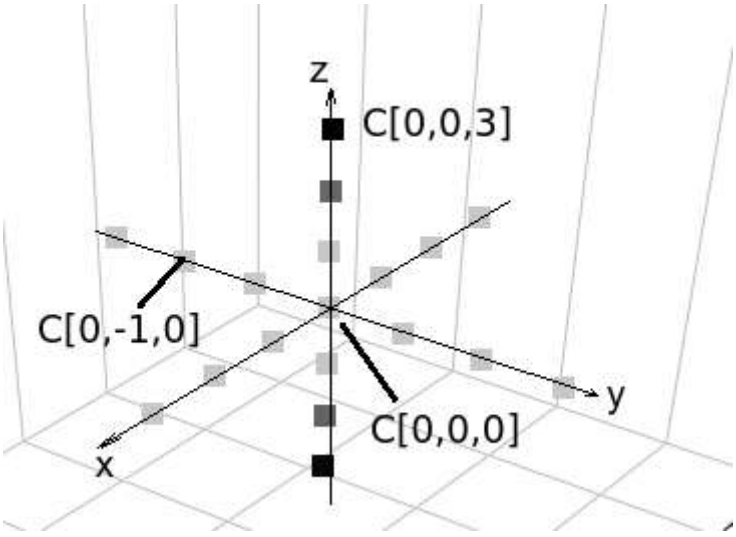

Figura 3.1: Stencil isotrópico 3D de sexta ordem

Código 3.1: Stencil isotrópico 3D de segunda ordem

```
#pragma omp parallel for
for(k = 3; k < Nz-3; k++){
  for(j = 3; j < Ny-3; j++){
    for(i = 3; i < Nx-3; i++){
      C[i + (Nx*j) + (Nx*Ny*k)] =
        a0 * C[i + (Nx*j) + (Nx*Ny*k)]
              + a1 * (C[i-1 + (Nx*j) + (Nx*Ny*k)]
                + C[i+1 + (Nx*j) + (Nx*Ny*k)]
                + C[i + (Nx*(j-1)) + (Nx*Ny*k)]
                + C[i + (Nx*(j+1)) + (Nx*Ny*k)]
                + C[i + (Nx*j) + (Nx*Ny*(k-1))]
                + C[i + (Nx*j) + (Nx*Ny*(k+1))])
              +  a2 * (C[i-2 + (Nx*j) + (Nx*Ny*k)]
                + C[i+2 + (Nx*j) + (Nx*Ny*k)]
                + C[i + (Nx*(j-2)) + (Nx*Ny*k)]
                + C[i + (Nx*(j+2)) + (Nx*Ny*k)]
                + C[i + (Nx*j) + (Nx*Ny*(k-2))]
                + C[i + (Nx*j) + (Nx*Ny*(k+2))])
              + a3 * (C[i-3 + (Nx*j) + (Nx*Ny*k)]
                + C[i+3 + (Nx*j) + (Nx*Ny*k)]
                + C[i + (Nx*(j-3)) + (Nx*Ny*k)]
                + P2[i + (Nx*(j+3)) + (Nx*Ny*k)]
                + C[i + (Nx*j) + (Nx*Ny*(k-3))]
                + C[i + (Nx*j) + (Nx*Ny*(k+3))]);
    }
  }
}
```

Analisando o Código 3.1, observa-se que o valor do elemento sendo calculado depende dos valores atuais dos seus vizinhos sem dependências cruzadas, de ma-

neira que, em um caso extremo, todos os valores poderiam ser atualizados simultaneamente se não houvesse limitações de recursos. Esta é a característica de um problema extremamente adequado à computação paralela. Para estas aplicações, é amplamente utilizado o OpenMP[28], que permite que um programa seja escrito de forma serial, utilizando diretivas ou `pragmas` que não alteram a lógica do programa. No Código 3.1, a primeira linha é correspondente ao OpenMP. Para compilar um programa com OpenMP, é necessário utilizar a *flag* `-qopenmp`[29]; se for necessário que se ignore todas as diretivas do OpenMP, utiliza-se a `-qopenmp-stubs`[30].

Os problemas de otimização computacional podem ser categorizados em dois tipos[3]: limitados por memória (*Memory Bound*), quando o tempo de acesso aos elementos é maior que o tempo para realizar as operações matemáticas sobre eles, e limitados por computação ou por CPU (*CPU Bound, Compute Bound*), caso contrário. O problema proposto é do tipo limitado por memória (como analisado em [4] e [31]), portanto, se faz necessário entender como acontece o acesso a ela .

O acesso à memória volátil é feito em quatro níveis, onde três deles são diferentes níveis da memória *cache* , que guarda informações recentemente utilizadas, e o último e mais lento acesso é à memória principal, RAM. Os níveis de *cache* são definidos pela sua proximidade à CPU e, por consequência, sua latência, isto é, o tempo perdido entre a requisição da informação e sua entrega à CPU, e são chamados de L1, L2 e L3, em ordem crescente de atraso, decorrente da diminuição da proximidade com os registradores. Para ilustrar, as Tabelas 3.2 (conforme [32]) e 3.3 (obtida com o comando `dmidec`) mostram o tamanho dos níveis *cache* da Xeon Phi, que possui dois níveis de *cache*, e Xeon E5-2650, com três níveis.

Tabela 3.2: Parâmetros de *cache* da Xeon Phi

Nível	Tamanho
L1	32 kB[32]
L2	512 kB por *core*[32]

O coprocessador Xeon Phi é um sistema eletrônico que possui os principais componentes de um computador, como memória, sistema de ventilação e núcleos de pro-

Tabela 3.3: Parâmetros de *cache* da Xeon E5-2650 *codename* Sandy Bridge

Nível	Tamanho
L1	512 kB
L2	2048 kB
L3	20480 kB
RAM	>1GB

cessamento, porém, não é um computador em si, sendo conectada a uma máquina de alto desempenho (conhecido como *host*) através de portas PCIe x16[33], rodando um sistema operacional Linux próprio mais enxuto e aumentando o poder de processamento, como se fossem adicionados nós a um *cluster*. O coprocessador conta com a tecnologia MIC (*Many Integrated Cores*) que reúne, em um mesmo chip, mais de 50 núcleos, interconectados por um anel bidirecional de alta velocidade, como no esquema da Figura 3.2. Cada núcleo possui uma memória *cache* dedicada L1 e, através do anel bidirecional, conectam suas memórias *cache* L2 de forma compartilhada e coerente, de maneira que a *cache* L2 de um núcleo pode ser acessada por qualquer outro núcleo, resultando em uma *cache* L2 total de mais de 30 MB.

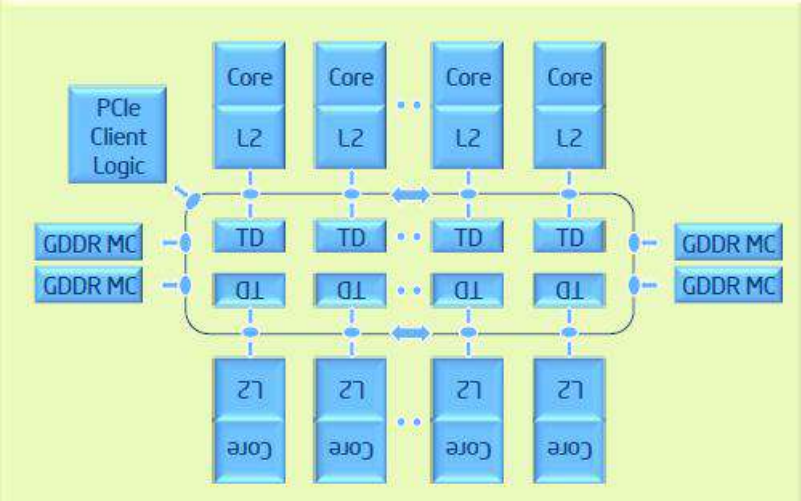

Figura 3.2: Ilustração da arquitetura MIC (retirado de [34])

Como o coprocessador está conectado a um outro computador através da porta PCI, o acesso ao seu sistema operacional pode ser feito através de SSH, a partir do *host*. Usualmente, o ambiente de um *cluster* já é remoto, então, o mais comum é

realizar um SSH ao *host*, onde é feita a compilação, usando a *flag* `-mmic`, do código que será executável no ambiente MIC, e, em seguida, um SSH ao coprocessador, onde o binário gerado é executado, sendo este modo de operação conhecido como nativo. Entretanto, também é possível compilar e executar a aplicação no próprio *host*, o que é conhecido como execução *offload*, onde é possível executar trechos no *host* e outros trechos no MIC de maneira que, se não houver um coprocessador conectado, todo o código é automaticamente executado somente pelo *host*. Porém, a escrita do código se torna muito presa ao par *host*-MIC e, para aplicações altamente paralelizáveis e limitadas por memória, há uma recomendação da própria Intel[35] argumentando que o modo nativo é mais adequado. Vale observar que um código desenvolvido para o *host* pode ser executado no ambiente MIC apenas adicionando a *flag* `-mmic` em sua compilação. Porém, para obter desempenho com o coprocessador, é necessário utilizar as técnicas de otimização apresentadas ao longo deste capítulo.

A seguir, são apresentadas algumas técnicas de otimização que podem ser feitas manualmente ou pelo compilador. Um levantamento semelhante pode ser encontrado em [4], [32] e [36].

3.2.2 SIMD

O método *Single Instruction, Multiple Data* (SIMD) permite realização de uma operação simultaneamente a mais de um dado, sendo conhecido, para as arquiteturas Intel, como vetorização. Esta arquitetura, em vez de aplicar uma operação várias vezes a cada trecho de um vetor, aplica uma operação a todo o vetor (por exemplo, `float[32]`), ou, pelo menos, um trecho maior que uma única unidade (um único `float`), como exemplificado pelos Códigos 3.2 e 3.3, e pela Figura 3.3 (retirada de [37]). Serão usadas, nesse trabalho, três tecnologias disponíveis para SIMD: SSE (*Streaming SIMD Extensions*), com capacidade de 128 *bits*[38]; AVX2 (*Advanced Vector Extensions*), com 256 *bits*[39]; e IMCI (*Initial Many Core Instructions*), com 512 *bits*[40].

Código 3.2: Sem vetorização. Realiza 32 somas

```
int a[32];
int b[32];
int c[32];
for (i = 0; i<32; i++)
  a[i] = b[i] + c[i];
```

Código 3.3: Com vetorização. Realiza 1 soma com 32 elementos simultaneamente

```
int a[32];
__vector signed int *va = (__vector signed int *) a;
int b[32];
__vector signed int *vb = (__vector signed int *) b;
vc = vec_add(va, vb);
```

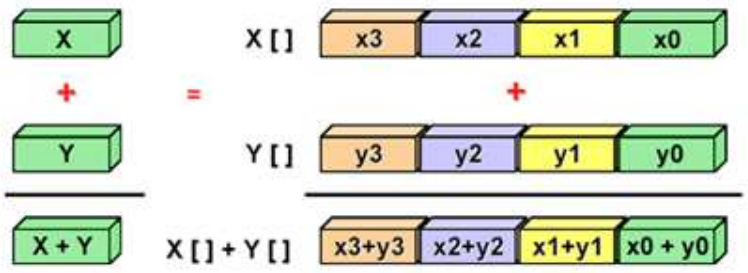

Figura 3.3: Ilustração da otimização SIMD

3.2.3 *Loop peeling* e `ivdep`

Muitas otimizações são feitas pelo próprio compilador, através da *flag* -O, escalando o nível de -O0 a -O3. Em geral, a otimização -O2 é a mais indicada e segura[41], porém, para obter performance de otimizações de *loops*, é preciso utilizar -O3, procurando auxiliar o compilador para tomar a decisão de certas otimizações, inclusive a vetorização. O código de um *loop* interno deve ser o mais conciso possível, pois condicionais e desalinhamento de posições de memória podem impedir otimizações

do nível O3. Para evitar isto, pode ser realizado, manualmente ou pelo compilador, o *loop peeling*, que retira as condicionais e variáveis desnecessárias, tornando o *loop* mais simples. Se, ainda assim, houver dependências no *loop* impedindo a vetorização, pode ser adicionada a pragma `ivdep`, que pede ao compilador a vetorização, ignorando possíveis dependências dos índices dos vetores (de forma mais bruta, # `pragma simd`[31]). Para obter mais informações sobre o comportamento das otimizações feitas pelo compilador, utiliza-se a *flag* `-qopt_report`(em versões mais antigas do compilador, `-vec-report`), seguido de um número de 0 a 5, sendo 0 nenhuma informação (padrão) e 5 o maior detalhamento possível, incluindo justificativas para *loops* que forem vetorizados e para os que não forem [42].

3.2.4 Alterações na lógica do *loop*

Para reduzir o tempo de execução do *stencil*, existem diversas técnicas que envolvem o acesso à memória *cache* . A aplicação dessas técnicas, muitas vezes, é feita de forma empírica, observando o tempo de execução para a computação de todos os elementos do *stencil* . Em [43], são apresentadas três dessas técnicas: blocagem (*tiling*), *loop permutation* e *loop unrolling*. A blocagem consiste em dividir a quantidade de elementos carregados por cálculo de maneira que, a cada passo do algoritmo, mais elementos estejam disponíveis na *cache*, e, por consequência, diminua o tempo de execução. Idealmente, deseja-se que somente seja acessado um nível da *cache*, porém, nem sempre é possível manter todos os elementos de um bloco. É preciso ter um compromisso entre a quantidade de elementos carregados e o tamanho do bloco. Em [44], foi feita uma ferramenta para a escolha automática deste parâmetro, entretanto, normalmente, a escolha do tamanho do bloco é feita por tentativa e erro, observando as melhorias nos tempos de execução. O Código 3.4 ilustra o procedimento da blocagem.

A outra técnica utilizada, permutação, consiste em alterar a ordem dos *loops* aninhados: matematicamente, o resultado é o mesmo, mas, com relação ao acesso a endereços de memória, a permutação pode fazer com que o acesso deixe de ser em posições distantes na memória para ser em vizinhos.

Código 3.4: Ilustrando a blocagem com blocos 512x512

```
#define NX 2000
#define NZ 2000
#define block_i_size 512
#define block_k_size 512

#pragma omp parallel num_threads(cores) private(i,k,block_k,block_i)
#pragma omp for schedule(static,1) collapse(2)
for (block_k=8;block_k < Nz-8;block_k+=block_k_size){
  for (block_i=8;block_i < Nx-8;block_i+=block_i_size){
    for(k = block_k;
        k < MIN(block_k_size+block_k,Nz-8);
        k++)
    {
      #pragma vector always
      #pragma omp simd safelen(8)
        for(i = block_i;
            i < MIN(block_i_size+block_i,Nx-8);
            i+=SIMD_STEP)
        {
          c[i+NX*k] = c[i+NX*k]*a[i+NX*k] + b[i+NX*k];
        }
    }
  }
}
```

O *loop unrolling*, acontece quando se escreve o *loop* mais interno várias vezes (esse número de vezes é chamado de fator de *unroll*) em vez de submetê-lo à aritmética do índice mais interno. Um exemplo apenas para fins didáticos pode ser encontrado no Código 3.5.

3.2.5 *Prefetch*

O *prefetch* (termo curto para *instruction prefetch*) é a antecipação de uma instrução antes de ser, de fato, necessária. O processador requisita esta informação à memória principal e a mantém na *cache*, onde fica disponível com uma latência menor. O *prefetch* é habilitado por padrão para otimizações -O2 e -O3, mas pode ser desabilitado, dependendo da necessidade de poupar espaço em *cache*.

Código 3.5: Exemplo de *loop unrolling* de fator 2

```
int i, j, x, y, a[100][100], b[100], c[100];
int n = 100;
int by = 2;
for (i = 0; i < n; i ++) {
  for (j = 0; j < n; j += by) {
        c[i] = c[i] + a[i][j] * b[j];
        c[i+1] = c[i+1] + a[i+1][j+1] * b[j+1];
      }
    }
  }
}
```

3.2.6 Primeiro Toque e Afinidade de *Threads*

Os sistemas *multicore* que tem bancos de memória dedicados a cada um dos núcleos são chamados NUMA (*Non-Uniform Memory Access*), o que quer dizer que o acesso a um banco de memória de outro núcleo é possível, mas não contém a mesma latência que o acesso ao banco local. Para obter um melhor desempenho usando essa arquitetura, procura-se acessar elementos guardados na memória local, o que é feito usando a estratégia da afinidade de *threads*. A afinidade de *threads* é definida por variáveis de ambiente ou *flags* de compilação[45]. Além disso, o endereçamento de uma *array* deve ser feito de maneira que estejam nos bancos de memória locais, para, quando forem processados, não sofrerem com a latência. O problema reside no fato de que o endereçamento não é feito no momento da alocação

de memória (*malloc*)[46], mas no primeiro acesso, isto é, o primeiro toque, tornando necessário, portanto, que seja feito um primeiro acesso ao *array* somente para definir o núcleo que irá processá-lo. A política do primeiro toque não é a única existente para otimizar a distribuição de variáveis entre os núcleos[47], mas é uma forma interessante quando existe controle sobre a declaração de variáveis e a afinidade de *threads*.

3.2.7 *Intrinsics*

Instrinsics são chamadas de nível de programação relativamente baixo que permitem realizar otimizações a nível de Assembly ou mesmo programar *hardware*, como microcontroladores. A Intel tem desenvolvida *intrinsics* para as linguagens C e C++ que realizam operações SIMD de movimentação de dados, aritméticas e lógicas, com menor latência, resultando em um menor tempo. Devido a seu baixo nível, as *intrincs* são orientadas a uma determinada arquitetura SIMD. Estas instruções usam tipos de variáveis diferentes dos tipos clássicos (`int`, `float`, etc), e são caracterizadas por incluir, em uma só variável, elementos vetorizados. Por exemplo, existem os tipos __m64, com capacidade para 64 *bits*, o que pode acomodar 4 variáveis de 16 *bits* (4 `int`), 2 variáveis de 32 (2 `floats`) ou 1 de 64 (`double`). Uma introdução às *intrinsics* pode ser vista em [48] e [49]. A Tabela 3.4 ilustra os tipos e sua disponibilidade com as arquiteturas utilizadas.

3.2.8 *Meia Precisão*

Utilizando as *intrinsics*, é possível operar variáveis do tipo `float` com 16 *bits* no ambiente MIC. Não é possível usar a meia precisão diretamente em operações aritméticas por razões de arquitetura, entretanto, é possível utilizar as operações aritméticas por meio das *intrinsics* em vetores de 32 *bits* contendo os elementos de 16 *bits* convertidos sob uma latência próxima à de uma operação aritmética[50], e o armazenamento em memória usando 16 *bits*. As conversões são feitas com as seguintes *intrincs*:

- `_mm512_extload_ps`: Converte um vetor em elementos de precisão simples (32 *bits*).

Tabela 3.4: Tipos das *intrinsics*, suas capacidades e arquiteturas

Tipo	Tamanho	Tipos suposrtados	Arquiteturas
__m64	8×8 *bits* até 1 × 64 *bits*	`int`, `float`, `double`	SSE, AVX2 e MIC
__m128	4×32 *bits*	`float`	SSE, AVX2 e MIC
__m128i	16×8 *bits* até 2 × 64 *bits*	`int`	SSE, AVX2 e MIC
__m128d	2×64 *bits*	`double`	SSE, AVX2 e MIC
__m256	8×32 *bits*	`float`	AVX2 e MIC
__m256i	32×8 *bits* até 4 × 64 *bits*	`int`	AVX2 e MIC
__m256d	4×64 *bits*	`double`	AVX2 e MIC
__m512	16×32 *bits*	`float`	MIC
__m512i	256×8 *bits* - 8 × 64 *bits*	`int`	MIC
__m512d	16×64 *bits*	`double`	MIC

- `_mm512_extstore_ps`: Converte um vetor de precisão simples (32 *bits*) em elementos de tamanho menor que 32 *bits*.

Foi mostrado que esta técnica permitiu um decréscimo de 20% no uso de memória usando um *stencil* TTI 3D[50].

3.3 Limites da Otimização - Modelo de *roofline*

O modelo de *roofline* é um plano 2D, onde o eixo y representa a performance (em FLOP/s) e o x, a intensidade aritmética (IA, em FLOP/byte), limitado por, pelo menos, duas retas: a reta do limite de pico de performance (em FLOP/s, paralela ao eixo x) e a reta do limite de banda de memória (*bandwidth*, em GB/s, inclinada), como pode ser visto na Figura 3.4.

Para uma dada arquitetura, a figura obtida será sempre a mesma, o que varia é o ponto de operação, conforme a aplicação. Sua utilidade consiste em saber o quão próximo a otimização desenvolvida está dos limites operacionais. Para uma aplicação compilada, mede-se sua intensidade aritmética, observando a quantidade de operações realizadas, a quantidade de dados transferida e o tempo de execução[51], obtendo um ponto no gráfico, o ponto de operação. A IA é definida

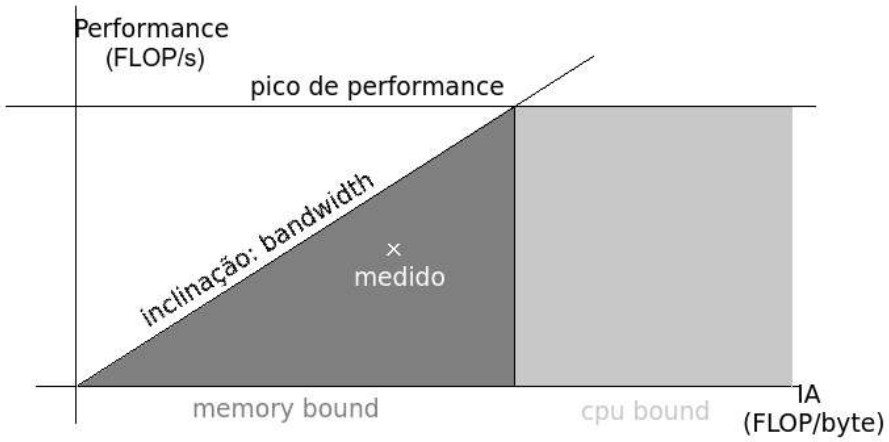

Figura 3.4: Modelo de *roofline*

pela quantidade de operações matemáticas dividida pela quantidade, em *bytes*, de memória, o que, por sua vez, é definido pela quantidade de operações de memória (*loads* e *stores*) multiplicado pelo tamanho da unidade transferida. Em resumo, a IA é dada pela equação (3.1). Para este trabalho, como as operações são com precisão simples (`float`), este tamanho é igual a 4 *bytes*[36].

$$IA = \frac{N_{\text{somas}} + N_{\text{multiplicações}}}{(N_{\text{loads}} + N_{\text{stores}}) \times \text{word\_size}} \tag{3.1}$$

Apesar de simples, a equação (3.1) tem diversas interpretações na literatura. No artigo original, o autor frisa que o eixo x refere-se à intensidade **operacional** em vez de aritmética[3], relacionando-se com a transferência da *cache* para a RAM, mas outros trabalhos apresentam como transferência entre CPU e RAM [51] ou mesmo com o nome de intensidade aritmética[52]. Em [53], é feita uma breve análise da IA de um *stencil* da Equação de Calor, similar ao *stencil* da Equação da Onda deste trabalho, e a contagem de transferência de dados leva em conta a quantidade de pontos do *stencil*, sem contar o elemento central nem os *stores*. Já em [31], a contagem é feita pelo número de matrizes e não por elemento da matriz. Para ilustrar, observando o Código 3.1, a contagem, de acordo com estes autores, é feita como um *load* e um *store* na matriz `C`. Similar ao que está sendo apresentado aqui, há o trabalho desenvolvido em [54], que também trata da perfilagem da solução da equação da onda por diferenças finitas e também monta o modelo de *roofline*.

Em [4], há uma análise para o *stencil* isotrópico, onde a quantidade de dados transferidos é dada pela razão entre o total de pontos acessados, internamente e na borda, pela quantidade de pontos calculados, ou seja, somente os internos, excluindo os pontos na borda, que não são calculados, como ilustrado pela Figura 3.5 e pela equação (3.2), onde N_{total} é o número de pontos calculados e N_{borda} são os pontos acessados no início e no fim dos *loops* dos eixos. Na figura, K é metade da ordem do *stencil*. Para calcular a IA, divide-se a quantidade de *bytes* por ponto pela quantidade de FLOP por ponto. O número total de pontos acessados é ilustrado pela Figura 3.6, que mostra uma das regiões de borda, e pela Figura 3.7, que mostra todas os pontos acessados, tanto os internos (calculados) quanto os de bordas (apenas acessados).

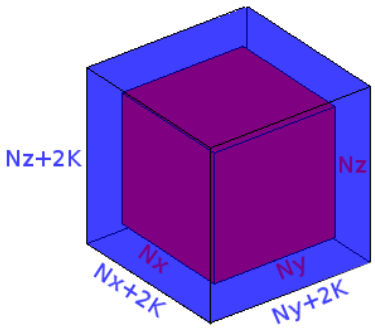

Figura 3.5: Espaço discretizado e sua borda

$$\text{bytes/ponto} = \text{word\_size}\frac{N_{\text{total}} + N_{\text{borda}}}{N_{\text{total}}} \tag{3.2}$$

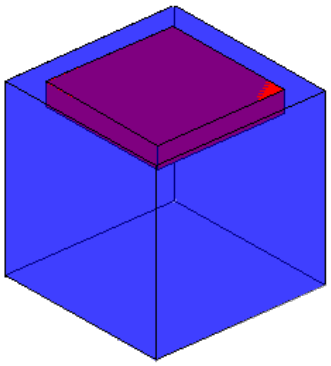

Figura 3.6: Destaque de uma das regiões de borda, onde há acesso, mas não há cálculo

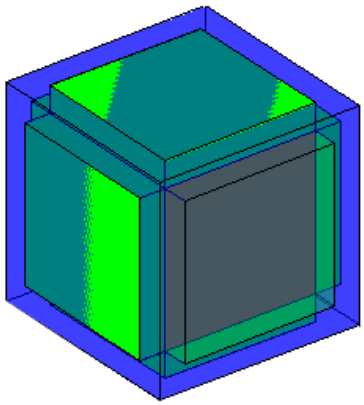

Figura 3.7: Regiões acessadas pelo *stencil*

O gráfico de *roofline* divide-se em duas regiões delimitadas por uma reta vertical passando pelo ponto de interseção entre as duas retas de limite. Se o ponto de operação obtido for abaixo da reta de banda de memória, o problema é *Memory-Bound* (limitado por memória); se for abaixo da reta de pico de performance, é um problema *CPU-Bound* e esta divisão implica em diferentes estratégias de otimização.

Pode-se considerar que a aplicação tem intensidade aritmética constante [3], portanto, traçando uma reta vertical passando pelo ponto de operação, os novos pontos de operação obtidos com estratégias de otimização que estejam acima do ponto atual podem ser considerados pontos melhores, por estarem mais próximos ao limite da máquina. Em [3], o autor divide o *Roofline* em três regiões, limitadas por 6 estratégias de otimização, são elas:

1. Paralelismo com *threads*
2. SIMD
3. Desbalanceamento entre soma e multiplicação
4. Passo unitário no acesso à memória
5. Afinidade de *threads*
6. *Prefetch*

A primeira técnica é considerada o trivial de uma estratégia de otimização. A maioria delas já foi mencionada na seção anterior, restando explicar a número 3. Para realizar o cálculo de intensidade aritmética, deve-se levar em conta que as CPUs podem realizar, simultaneamente, uma operação de soma e uma de multiplicação[55], através de uma estratégia de arquitetura conhecida como FMA (*Fused Multiply-Add*). Então, quando se calcula o número de operações, considerando uma aplicação CPU-Bound, a quantidade de operações é o máximo entre o número de adições e o número de multiplicações (conforme a equação (3.3)[31]). Devido a este desbalanceamento, é possível aumentar o número de multiplicações ou adições para aproveitar ao máximo a unidade aritmética, mantendo o número de operações balanceado entre essas operações. Além disso, o cálculo do desbalanceamento influencia os limites máximos de operação do *hardware*, uma vez que o máximo teórico de operações vai acontecer quando houver perfeito balanceamento entre adições e multiplicações, alterando as curvas limitantes no modelo de *roofline*.

$$N_{FLOP} = \frac{N_{ADD} + N_{MUL}}{2 \times max(N_{ADD}, N_{MUL})} \tag{3.3}$$

3.4 Análise de Consumo de Energia

Para realizar análises relacionadas ao consumo de energia de computadores, as principais empresas do ramo criaram, em 1996 [56], um padrão para categorizar em estados o comportamento dos componentes com relação ao consumo. Os estados de interesse para performance são os estados-P(*Performance-States* ou, simplesmente, *P-States*) e os estados-C (*Processor-States* ou, simplesmente, *C-States*, C vindo de *Core*), além dos *Package C-States*, para coprocessadores[57]. Os estados são classificados como P_0 até P_n (ou C_0 a C_n), onde n depende do fabricante e 0 representa maior performance (portanto, maior consumo energético) e alguns estados são opcionais (como o C_2)[56].

3.4.1 LikwidPowermeter

Uma das ferramentas disponíveis para a análise de consumo energético é o LikwidPowermeter, do pacote Likwid, que acessa registradores específicos da CPU para

exibir parâmetros como Energia e Potência da CPU e das memórias e frequência de *clock*. Um exemplo do seu uso é encontrado na Figura 3.8. Mais informações podem ser obtidas em seu manual, disponível localmente e na sua página[58].

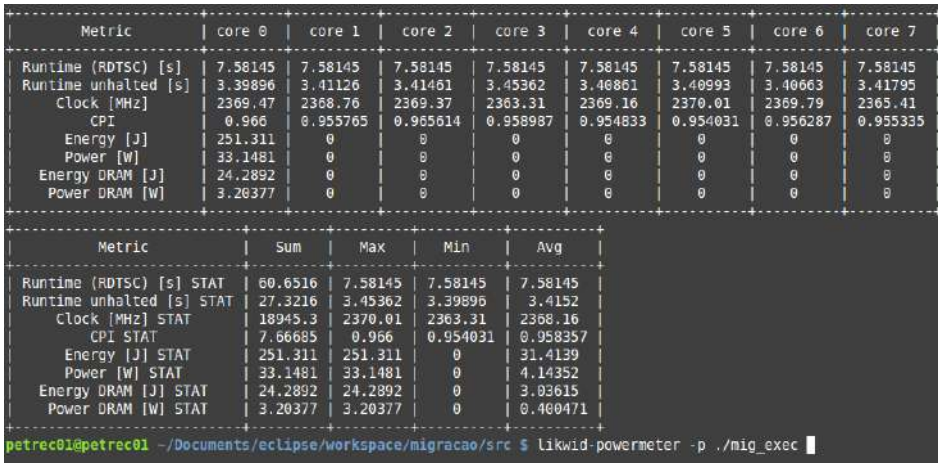

Figura 3.8: Saída do LikwidPowermeter

3.4.2 micsmc

Para a arquitetura MIC, existe a aplicação micsmc, para monitorar o comportamento energético dos coprocessadores conectados. Embora exista uma interface gráfica, a versão por linha de comando é mais usual, já que o acesso ao *host* normalmente já é feito de forma remota. É possível ver a potência e a frequência com o comando `micsmc -f` (conforme a Figura 3.9) e a temperatura com `micsmc -t` (além de outras entradas, que não estão no escopo deste trabalho). A relação entre a potência despendida pelas fontes de tensão e os estados energéticos para o modelo 3120 (conforme seu manual [33]) é mostrada na tabela 3.5.

Tabela 3.5: Relação entre potência dissipada e estados energéticos para a Xeon Phi

Estado Energético	Consumo
C0	300 W
C1	<115 W
PC3	<50 W
PC6	<30 W

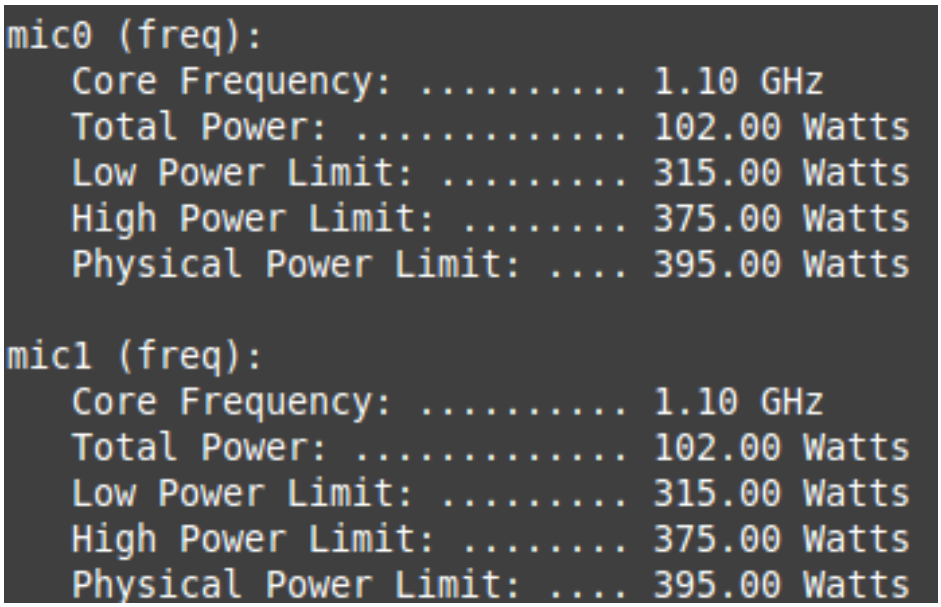

Figura 3.9: *Output* do micsmc

3.4.3 VTune

VTune é uma ferramenta desenvolvida pela própria Intel, capaz de indicar as métricas de performance com relação ao código da aplicação, linha por linha. Conta com duas análise relacionadas ao consumo energético: *CPU Sleep States* e *CPU Frequency*, além de análises personalizadas. Além disso, permite a obtenção de dados de registradores que fornecem a quantidade de operações aritméticas realizadas, já que o compilador altera as operações escritas explicitamente no código. Um exemplo é a operação FMA, que permite, a nível de Assembly, realizar uma operação do tipo `a = a + b*c` em somente uma operação, em vez de duas. Usando as métricas fornecidas por estes registradores de desempenho que o VTune tem acesso, é possível estimar melhor a quantidade de FLOP/s.

3.5 Arquiteturas avaliadas

A estrutura computacional disponível para este trabalho está listada na tabela 3.6, junto com seus parâmetros de performance citados na seção 3.3, sendo apresentados os limites fornecidos pelo fabricante e os limites obtidos através dos *benchmarks* Linpack (*CPU-Bound*) e STREAM (*Memory-Bound*). Os dados teóricos para pico

de FLOP/s, apresentados na Tabela 3.7, são calculados usando a fórmula (3.4)

$$\text{GFLOP/s}_{\text{max}} = N_{SIMD} \times \alpha_{FMA} \times f_{clock} \times NUM\_THREADS \tag{3.4}$$

onde:

- N_{SIMD} é o número de elementos por vetorização,
- α_{FMA} é um fator que vale 2 se a arquitetura permitir FMA,
- f_{clock} é a frequência de *clock* do processador e
- NUM_THREADS é o número de *threads*.

O parâmetro N_{SIMD} é obtido dividindo o tamanho destinado a SIMD, como mostrado na seção 3.2.2, pelo tamanho de um elemento de ponto flutuante, em específico de precisão simples (SP, *Single Precision*), isto é, 32 *bits* ou 4 *bytes*. As outras colunas da Tabela são valores, respectivamente, referentes ao observado executando o *benchmark*, e disponíveis na literatura.

Tabela 3.6: SIMD e *Clock* das arquiteturas utilizadas

Arquitetura	SIMD	f_{clock}
Xeon E5-2650 ("Sandy Bridge")	256 *bits*	2,0 GHz
Xeon E5-2670 ("Sandy Bridge")	256 *bits*	2,6 GHz
Xeon Phi 3120P ("Knights Corner")	512 *bits*	1,1 GHz

Tabela 3.7: Valor de pico de GFLOP/s

Arquitetura	Teórico (SP)	LINPACK (DP)	Referência
Xeon E5-2650	512	273	262[59]
Xeon E5-2670	666	316	321 (DP)[60]
Xeon Phi 3120P	2006	635	715 (DP)[61]

Observação: O LINPACK sempre roda com `float` de 64 *bits*[62] (DP, *Double Precision*)

O *benchmark* STREAM é executado usando a linha de compilação abaixo, e o valor obtido é para a operação *Triad* (a[i]=b[i]+k*c[i])[63]. O valor NUMERO_MAX_THREADS depende da arquitetura e vale 32 para Sandy Bridge e 228 para a Xeon Phi.

```
icc -fopenmp -D_OPENMP stream.c -o stream
export KMP_AFFINITY=scatter
export OMP_NUM_THREADS=NUMERO_MAX_THREADS
```

A Tabela 3.8 apresenta os valores de pico da largura de banda segundo os seguintes critérios: de acordo com o manual do fabricante; obtido rodando o *benchmark* compilado com gcc e com icc, e obtidos na literatura.

Tabela 3.8: Valor de pico de *bandwith*, em GB/s

Arquitetura	Manual	STREAM		Referência
		gcc	icc	
"Sandy Bridge" Xeon E5-2650	68[64]	46,7	61,5	64[65]
"Knights Corner" Xeon Phi 3120P	240[66]	×	131,2	170[61]

3.6 Uso de *cache*

Uma análise necessária para o processo de otimização é o uso da memória *cache* no *stencil*. O tempo de execução do código pode ser diretamente impactado pela quantidade de pontos do *stencil* carregados nos diferentes níveis de *cache* apresentados nas Tabelas 3.2 e 3.3. É comum a denominação LLC (*last level cache*) para se referir ao último nível de *cache* antes de ocorrer um acesso à memória principal. Quando isto ocorre, utiliza-se a denominação de LLC *miss*. Por ter dois níveis de cache, o LLC da Xeon Phi é o L2 e, para Sandy Bridge, com 3 níveis de cache, isto é, o L3. O LLC da Xeon Phi é compartilhado entre todos os *cores*, portanto, o tamanho total é o de 57 *cores* multiplicado pelo tamanho individual da *cache*. Já para a arquitetura Sandy Bridge, os níveis L1 e L2 são individuais para cada *core* e L3 é compartilhado.

Tabela 3.9: Tamanho dos níveis de memória *cache* das arquiteturas disponíveis

Nível	Xeon E5-2650		Xeon Phi 3120P	
	Tamanho	Latência	Tamanho	Latência
L1	512 kB	5 ciclos[67]	32 kB[32]	3 ciclos [32][68]
L2	2048 kB	12 ciclos[67]	28,5 MB*[32]	15 ciclos[32][69] 24 ciclos [68]
L3	20 MB	28 ciclos[67]	×	×

*57×512 kB

Usando os dados da Tabela 3.9, pode-se calcular a quantidade de pontos possíveis de serem carregados na *cache*, divindo a capacidade da *cache* pelo tamanho de um ponto do *stencil*, ou seja, um `float` de 32 *bits*, e o resultado é apresentado na Tabela 3.10.

Tabela 3.10: Capacidade da *cache*, em pontos

Nível	Xeon E5-2650	Xeon Phi 3120P
L2	64	912
L3	640	×

Para facilitar a compreensão, primeiramente, é feita a análise do *stencil* isotrópico 2D. Observando o código 3.1, pode-se notar que o tamanho do passo na direção z é N_x, pois, como as matrizes são representadas como vetores unidimensionais pela memória, quando se avança um passo na direção z, o laço mais interno caminhou na direção x, portanto, N_x vezes. Portanto, para o cálculo de um ponto, são necessários carregamentos de pontos nesta direção, de maneira que, estando distantes, haverá mais chance de um L2 *miss*. Este efeito é amenizado pela blocagem do código, conforme ilustrado pelas Figuras 3.10 e 3.11.

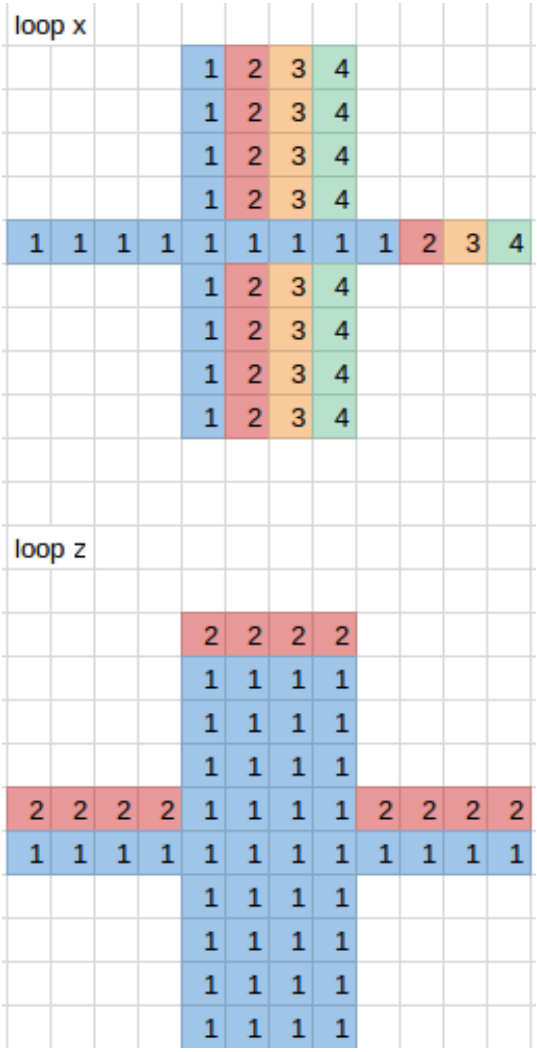

Figura 3.10: Varredura da matriz P2 do Código 4.1 (com ordem 8) sem blocagem, iterando, respectivamente x e z

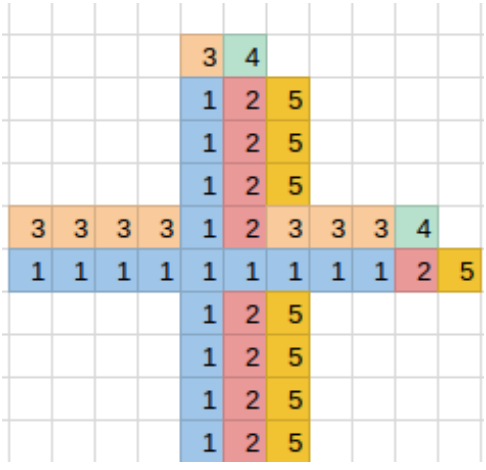

Figura 3.11: Varredura de P2 com blocagem 2×2

Para o *stencil* TTI 3D, é necessária a computação das segundas derivadas de P e Q com relação a x, y, e z, e das derivadas cruzadas xy, yz e xz, conforme o Código 3.6 e as equações (2.12), (2.13) e (2.14). As derivadas não cruzadas ($\partial^2 P/\partial x^2$, $\partial^2 P/\partial y^2$, $\partial^2 P/\partial z^2$, $\partial^2 Q/\partial x^2$, $\partial^2 Q/\partial y^2$ e $\partial^2 Q/\partial z^2$) requerem dados que correspondem ao *stencil* isotrópico, conforme a Figura 3.12. A primeira otimização aplicada é o carregamento vetorizado da camada mais interna do *stencil*, isto é, os vizinhos cuja posição na memória segue em passos unitários. Por conta disso, todos os valores na direção x são carregados de uma só vez (na Figura 3.12, são os pontos em vermelho).

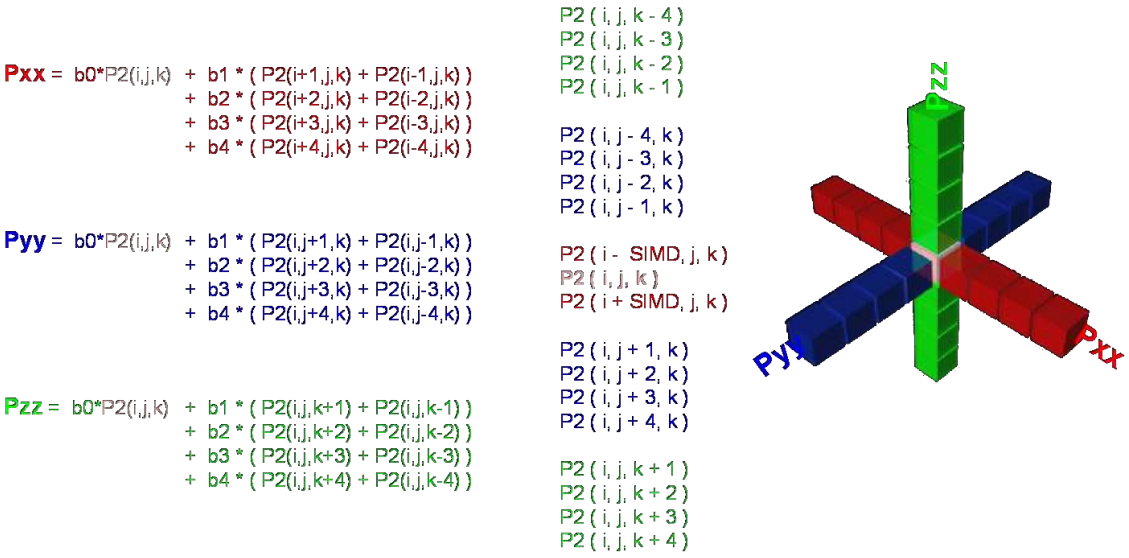

Figura 3.12: Cálculo dos termos não cruzados

Código 3.6: Stencil TTI 3D de oitava ordem

```
for k=4, Nz−4
    for j 4, Ny−4
        for i 4, Nx−4
          Pxx = b0 ∗ P2(i,j,k) + b1 ∗ ( P2(i+1,j,k) + P2(i−1,j,k) )+
                ...+ b4 ∗ ( P2(i+4,j,k) + P2(i−4,j,k) )
          Pyy = b0 ∗ P2(i,j,k) + b1 ∗ ( P2(i,j+1,k) + P2(i,j−1,k) )+
               ...+ b4 ∗ ( P2(i,j+4,k) + P2(i,j−4,k) )
          Pzz = b0 ∗ P2(i,j,k) + b1 ∗ ( P2(i,j,k+1) + P2(i,j,k−1) )+
               ...+ b4 ∗ ( P2(i,j,k+4) + P2(i,j,k−4) )
          Pxy = a1∗a1∗(
                  P2(i+1,j+1,k) − P2(i−1,j+1,k) − P2(i+1,j−1,k) + P2(i−1,j−1,k)
                )+...+
                a4∗a4∗(
                  P2(i+4,j+4,k) − P2(i−4,j+4,k) − P2(i+4,j−4,k) + P2(i−4,j−4,k)
                )
          Pxz = a1∗a1∗(
                  P2(i+1,j,k+1) − P2(i−1,j,k+1) − P2(i+1,j,k−1) + P2(i−1,j,k−1)
                )+...+
                a4∗a4∗(
                  P2(i+4,j,k+4) − P2(i−4,j,k+4) − P2(i+4,j,k−4) + P2(i−4,j,k−4)
                )
          Pyz = a1∗a1∗(
                  P2(i,j+1,k+1) − P2(i,j−1,k+1) − P2(i,j+1,k−1) + P2(i,j−1,k−1)
               )+...+ a4∗a4∗(
                  P2(i,j+4,k+4) − P2(i,j−4,k+4) − P2(i,j+4,k−4) + P2(i,j−4,k−4)
                )
            ...
        end
    end
end
```

Para os termos cruzados, são necessários os vizinhos nos planos xy, yz e xz. Para os planos que contém o eixo x, cada linha é carregada usando vetorização, conforme as Figuras 3.13 e 3.14.

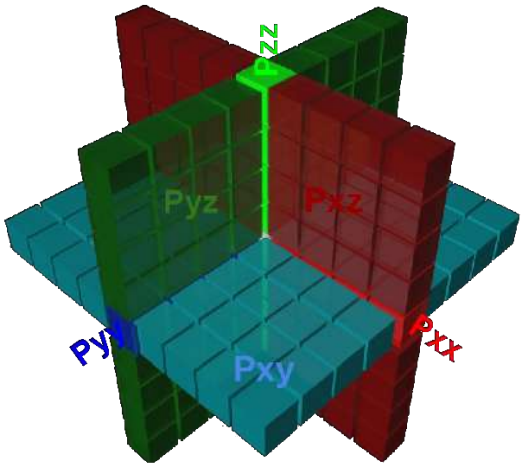

Figura 3.13: *Stencil* TTI 3D

```
P2 ( i, j - 4, k - 4 )   P2 ( i, j - 4, k - 2 )   P2 ( i - SIMD, j - 4, k )   P2 ( i, j - 4, k + 1 )   P2 ( i, j - 4, k + 3 )
P2 ( i, j - 3, k - 4 )   P2 ( i, j - 3, k - 2 )   P2 ( i, j - 4, k )         P2 ( i, j - 3, k + 1 )   P2 ( i, j - 3, k + 3 )
P2 ( i, j - 2, k - 4 )   P2 ( i, j - 2, k - 2 )   P2 ( i + SIMD, j - 4, k )  P2 ( i, j - 2, k + 1 )   P2 ( i, j - 2, k + 3 )
P2 ( i, j - 1, k - 4 )   P2 ( i, j - 1, k - 2 )   P2 ( i - SIMD, j - 3, k )  P2 ( i, j - 1, k + 1 )   P2 ( i, j - 1, k + 3 )
P2 ( i - SIMD, j, k - 4 ) P2 ( i - SIMD, j, k - 2 ) P2 ( i, j - 3, k )       P2 ( i - SIMD, j, k + 1 ) P2 ( i - SIMD, j, k + 3 )
P2 ( i, j, k - 4 )       P2 ( i, j, k - 2 )       P2 ( i + SIMD, j - 3, k )  P2 ( i, j, k + 1 )       P2 ( i, j, k + 3 )
P2 ( i + SIMD, j, k - 4 ) P2 ( i + SIMD, j, k - 2 ) P2 ( i - SIMD, j - 2, k ) P2 ( i + SIMD, j, k + 1 ) P2 ( i + SIMD, j, k + 3 )
P2 ( i, j + 1, k - 4 )   P2 ( i, j + 1, k - 2 )   P2 ( i + SIMD, j - 2, k )  P2 ( i, j + 1, k + 1 )   P2 ( i, j + 1, k + 3 )
P2 ( i, j + 2, k - 4 )   P2 ( i, j + 2, k - 2 )   P2 ( i - SIMD, j - 1, k )  P2 ( i, j + 2, k + 1 )   P2 ( i, j + 2, k + 3 )
P2 ( i, j + 3, k - 4 )   P2 ( i, j + 3, k - 2 )   P2 ( i, j - 1, k )         P2 ( i, j + 3, k + 1 )   P2 ( i, j + 3, k + 3 )
P2 ( i, j + 4, k - 4 )   P2 ( i, j + 4, k - 2 )   P2 ( i + SIMD, j - 1, k )  P2 ( i, j + 4, k + 1 )   P2 ( i, j + 4, k + 3 )
                                                  P2 ( i - SIMD, j, k )
P2 ( i, j - 4, k - 3 )   P2 ( i, j - 4, k - 1 )   P2 ( i, j, k )
P2 ( i, j - 3, k - 3 )   P2 ( i, j - 3, k - 1 )   P2 ( i + SIMD, j, k )     P2 ( i, j - 4, k + 2 )   P2 ( i, j - 4, k + 4 )
P2 ( i, j - 2, k - 3 )   P2 ( i, j - 2, k - 1 )   P2 ( i - SIMD, j + 1, k )  P2 ( i, j - 3, k + 2 )   P2 ( i, j - 3, k + 4 )
P2 ( i, j - 1, k - 3 )   P2 ( i, j - 1, k - 1 )   P2 ( i, j + 1, k )         P2 ( i, j - 2, k + 2 )   P2 ( i, j - 2, k + 4 )
P2 ( i - SIMD, j, k - 3 ) P2 ( i - SIMD, j, k - 1 ) P2 ( i + SIMD, j + 1, k ) P2 ( i, j - 1, k + 2 )   P2 ( i, j - 1, k + 4 )
P2 ( i, j, k - 3 )       P2 ( i, j, k - 1 )       P2 ( i - SIMD, j + 2, k )  P2 ( i - SIMD, j, k + 2 ) P2 ( i - SIMD, j, k + 4 )
P2 ( i + SIMD, j, k - 3 ) P2 ( i + SIMD, j, k - 1 ) P2 ( i, j + 2, k )       P2 ( i, j, k + 2 )       P2 ( i, j, k + 4 )
P2 ( i, j + 1, k - 3 )   P2 ( i, j + 1, k - 1 )   P2 ( i + SIMD, j + 2, k )  P2 ( i + SIMD, j, k + 2 ) P2 ( i + SIMD, j, k + 4 )
P2 ( i, j + 2, k - 3 )   P2 ( i, j + 2, k - 1 )   P2 ( i - SIMD, j + 3, k )  P2 ( i, j + 1, k + 2 )   P2 ( i, j + 1, k + 4 )
P2 ( i, j + 3, k - 3 )   P2 ( i, j + 3, k - 1 )   P2 ( i, j + 3, k )         P2 ( i, j + 2, k + 2 )   P2 ( i, j + 2, k + 4 )
P2 ( i, j + 4, k - 3 )   P2 ( i, j + 4, k - 1 )   P2 ( i + SIMD, j + 3, k )  P2 ( i, j + 3, k + 2 )   P2 ( i, j + 3, k + 4 )
                                                  P2 ( i - SIMD, j + 4, k )  P2 ( i, j + 4, k + 2 )   P2 ( i, j + 4, k + 4 )
                                                  P2 ( i, j + 4, k )
                                                  P2 ( i + SIMD, j + 4, k )
```

Figura 3.14: Cálculo dos termos cruzados

Por se tratar de um *benchmark*, foi definido que o tamanho do problema no eixo z seja igual ao número de *threads* disponíveis (isto é, $N_Z = 228$), de maneira que a melhor performance seja obtida durante a paralelização.

Capítulo 4

Análise de Desempenho

4.1 Modelo de *Roofline*

Os parâmetros necessários para a construção do modelo já foram apresentados na Seção 3.5, sendo possível, com eles, construir o as curvas limites do modelo (Figura 4.2). É importante observar que, no gráfico obtido, foram usados os valores da Tabela 3.7 medidos com *benchmark* e obtidos na literatura multiplicados por dois, pois os dados eram referentes ao cálculo usando precisão dupla (64 *bits*). Como o problema sempre considera a precisão simples, foi feita essa extrapolação. A Tabela 4.1 mostra a IA limite para cada curva das Figuras 4.1 e 4.2, segundo a fórmula (4.1). Para a primeira coluna (valor teórico), foram usados o valor máximo teórico de FLOP/s de acordo com a equação (3.4), e de largura de banda (BW, do inglês *bandwidth*) de acordo com o manual do fabricante. Para a segunda coluna (valor de *benchmark*), foram usados os valores de FLOP/s e BW, utilizando os seus respectivos *benchmarks* (Linkpack e STREAM), conforme a seção 3.5. E, para a terceira coluna, foram utilizados os valores de FLOP/s e BW obtidos na literatura , apresentados na seção 3.5, nas Tabelas 3.7 e 3.8. Resta, portanto, o cálculo da intensidade aritmética da aplicação, para o qual são necessários o número de operações de ponto flutuante realizados e a quantidade de bytes acessados, seja na leitura ($LOAD$) ou na escrita ($STORE$).

$$IA_{\text{limite}} = \frac{FLOP/s_{\text{pico}}}{BW} \tag{4.1}$$

Tabela 4.1: Valor da intensidade aritmética

Arquitetura	IA_{limite} (Teórico)	IA_{limite} (*Benchmark*)	IA_{limite} (Referência)
Xeon E5-2650	7,5 FLOP/byte	4,4 FLOP/byte	4,1 FLOP/byte
Xeon Phi 3120P	8,4 FLOP/byte	9,7 FLOP/byte	8,4 FLOP/byte

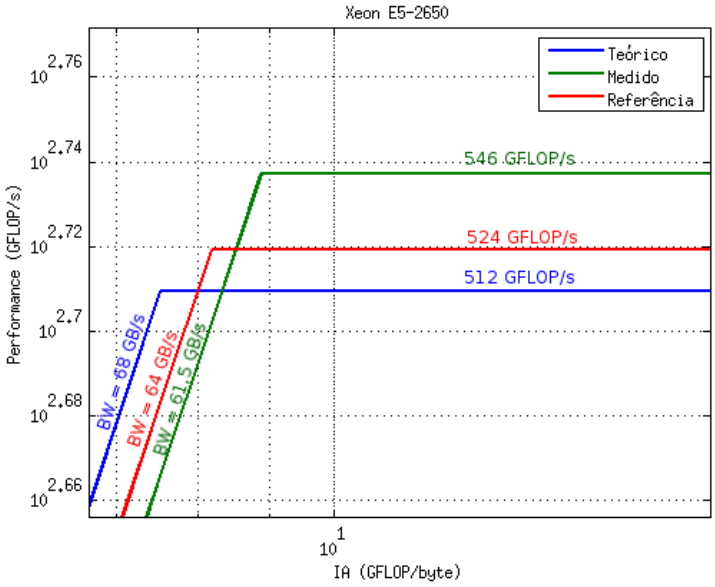

Figura 4.1: *Roofline* para a Xeon Sandy Bridge E5-2650, usando as Tabelas 3.7 e 3.8

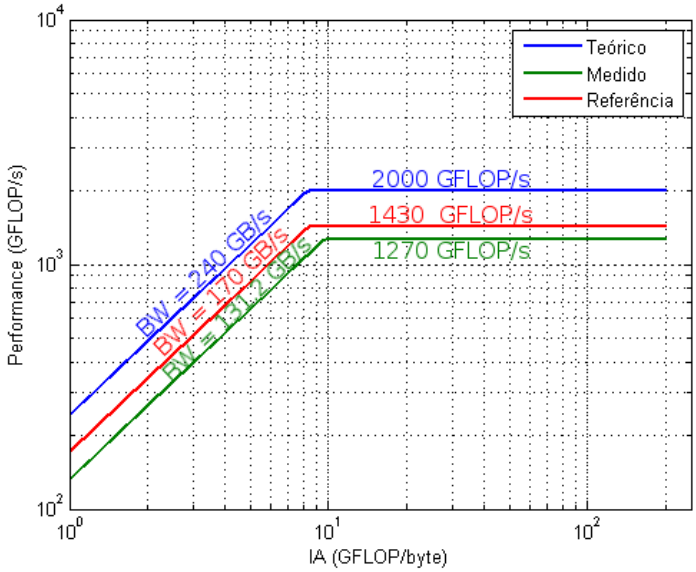

Figura 4.2: *Roofline* para a Xeon Phi 3120P, usando as Tabelas 3.7 e 3.8

4.1.1 Estimando a IA por análise do código

O cálculo mais simples da quantidade de FLOP e de acessos à memória é contando, manualmente, a quantidade de somas, multiplicações, carregamentos e atribuições no corpo do código. Este método não tem a precisão ideal, porque não leva em conta as otimizações do compilador, que altera as instruções a nível de ASSEMBLY, mas é um bom parâmetro, sendo o mais encontrado na literatura, como em [52], [55], [10] e [70].

Inicialmente, o cálculo é feito para o *stencil* isotrópico, seguindo a equação (3.1) (repetida a seguir por conveniência), como feito em [31], onde *word size* vale 32 *bits* ou 4 *bytes* para ponto flutuante de precisão simples.

$$IA = \frac{N_{\text{somas}} + N_{\text{multiplicações}}}{(N_{\text{loads}} + N_{\text{stores}}) \times \text{word\_size}} \tag{4.2}$$

Isotrópico 2D

Para exemplificar, o Código 4.1 mostra o cálculo da frente de propagação usando um *stencil* 2D de décima sexta ordem, onde P3 indica a frente de onda no tempo atual, P2, no tempo passado, e C, os multiplicadores da velocidade do meio.

Código 4.1: *Stencil* 2D de décima sexta ordem

```
for(n = 0; n < Ntotal; n++){
 for(k = 8; k < Nz-8; k++){
  for(i = 8; i < Nx-8; i++){
    P3(x,y) =  2.0f * P2(x,y) - P3(x,y) + (C(x,y) *(
      a016 *  P2(x,y) +
      a116 * (P2(x-1,y) + P2(x+1,y) + P2(x,y+1) + P2(x,y-1)) +
      a216 * (P2(x-2,y) + P2(x+2,y) + P2(x,y+2) + P2(x,y-2)) +
      a316 * (P2(x-3,y) + P2(x+3,y) + P2(x,y+3) + P2(x,y-3)) +
      a416 * (P2(x-4,y) + P2(x+4,y) + P2(x,y+4) + P2(x,y-4)) +
      a516 * (P2(x-5,y) + P2(x+5,y) + P2(x,y+5) + P2(x,y-5)) +
      a616 * (P2(x-6,y) + P2(x+6,y) + P2(x,y+6) + P2(x,y-6)) +
      a716 * (P2(x-7,y) + P2(x+7,y) + P2(x,y+7) + P2(x,y-7)) +
      a816 * (P2(x-8,y) + P2(x+8,y) + P2(x,y+8) + P2(x,y-8))
```

```
    ));
  }
 }
}
```

Pela análise do código, pode-se computar as operações de pontos flutuantes, obtendo 4K+2 somas e K+3 multiplicações, onde K é metade da ordem do *stencil*. No exemplo, são 34 somas e 11 multiplicações, para K = 16/2. De acordo com a metodologia de [31], a quantidade de *loads* é igual à quantidade de variáveis demandadas, no caso, P2, P3 e C, totalizando 3 *loads* e 1 *store* (P3). Substituindo os valores do Código 4.1 na equação (3.1), obtém-se a equação (4.3).

$$IA_{16}^{iso2D} = \frac{34 + 11}{4 \times 4} = 2,81 \text{ FLOP/byte} \tag{4.3}$$

Vale ressaltar que, ao realizar este cálculo, não se deve levar em conta o número de *threads* e o fator de SIMD, pois estes são incorporados, de forma indireta, através da diminuição do tempo de execução. Ou seja, quando o cálculo é feito, são consideradas as somas, multiplicações e atribuições entre o início e o fim do *loop* como um código serial, e o tempo de execução é medido antes e após o *loop*. A Tabela 4.2 mostra a IA calculada na equação (4.3) e a performance medida para o *stencil* isotrópico de ordem 16, usando a equação (4.4), para um *stencil* de tamanho $N_x = 751$ por $N_z = 301$, compilado, para a Sandy Bridge, com as *flags* a seguir:

- `-openmp`
- `-O3`
- `-fno-alias`
- `-opt-assume-safe-padding`
- `-opt-prefetch-distance=32,1`
- `-restrict`

e utilizando as seguintes variáveis de ambiente:

- `export OMP_NUM_THREADS=16`
- `export KMP_AFFINITY="compact,granularity=fine"`

Para a Xeon Phi, foi compilado com as seguintes *flags*:

- `-mmic`
- `-openmp`
- `-openmp-simd`
- `-O3`
- `-fno-alias`
- `-opt-assume-safe-padding`
- `-opt-prefetch-distance=2,1`
- `-restrict`

e foi executado com as variáveis de ambiente abaixo:

- `export LD_LIBRARY_PATH=$TBBROOT/../compiler/lib/mic`
- `export OMP_NUM_THREADS=228`
- `export KMP_STACKSIZE=2M`
- `export KMP_AFFINITY=compact,granularity=fine`

O cálculo da IA é igual para ambas arquiteturas, pois a quantidade de operações e movimentações de dados é a mesma independentemente de onde o código estiver sendo executado, isto é, a IA é uma característica do algoritmo. O mesmo vale para o código básico com relação ao otimizado, onde a diferença fundamental fica no tempo de execução, que altera somente o valor de FLOP/s. Para o valor máximo de FLOP/s, multiplica-se a IA do algoritmo pelo valor de *bandwidth*, se o código for *Memory Bound*, caso contrário, utiliza-se o valor de pico de FLOP/s diretamente (*CPU Bound*). Este cálculo está resumido na equação (4.5), onde $\text{IA}_{\text{limite}}$ é encontrada na Tabela 3.7. Seguindo a metodologia de [31], é utilizada a equação

(4.6) para calcular o valor de pico, considerando o desbalanceamento entre adições e multiplicações, caso a arquitetura tenha FMA.

$$FLOP/s = \frac{(N_x - 16)(N_z - 16)(N_{\text{somas}} + N_{\text{multiplicações}})}{t_{\text{execução}}} \tag{4.4}$$

$$FLOP/s_{max} = \begin{cases} IA_{\text{aplicação}} \times BW & \text{, se } IA_{\text{aplicação}} < IA_{limite} \\ FLOP/s_{pico} & \text{, se } IA_{\text{aplicação}} \geq IA_{limite} \end{cases} \tag{4.5}$$

$$FLOP/s_{max,corrigido} = FLOP/s_{max} \times \frac{N_{\text{somas}} + N_{\text{multiplicações}}}{2 \times \texttt{max}(N_{\text{somas}}, N_{\text{multiplicações}})} \tag{4.6}$$

Tabela 4.2: Resultados dos parâmetros de *roofline* por análise de código para o *stencil* isotrópico 2D

Versão	IA (Flop/byte)	GFlop/s	
		Sandy Bridge	Xeon Phi
Código Serial	2,81	14,5	2,65
Código Paralelo	2,81	88,6	256
Valor máximo	2,81	173	369
Valor máximo ponderado	2,81	114	244

TTI 3D

Devido à complexidade desta estrutura, o cálculo das operações é muito sujeito a erros se for feito por inspeção do código, justificando a necessidade de uma ferramenta de computação do número de operações, como apresentado na próxima seção. Entretanto, ainda assim, foi feita a análise por inspeção para ordem 8, resumida na Tabela 4.3, para as operações aritméticas e 4.4 para as operações de memória. A equação (4.7), utilizando os valores destas tabelas, apresenta o cálculo da IA para o *stencil* TTI 3D.

$$IA_8^{tti} = \frac{450 + 158}{4 \times 34} = 4,47 \text{ FLOP/byte} \tag{4.7}$$

Consultando a Tabela 3.7, observa-se que a IA calculada é maior que a IA limite para a arquitetura Sandy Bridge (*Compute Bound*), porém, menor que para a arquitetura Xeon Phi (*Memory Bound*). Portanto, usando a equação (4.5), obtém-se o valor máximo da arquitetura para Sandy Bridge, que, de acordo com a tabela 3.7, é de 316 GFlop/s para precisão dupla, o que resulta em 632 GFlop/s utilizando precisão simples. Para a arquitetura Xeon Phi, o cálculo se dá multiplicando o valor de *bandwidth* pela IA calculada, resultando em 586 GFlop/s. Esses resultados estão resumidos na Tabela 4.5. O fator de desbalanceamento entre somas e multiplicações é dado por:

$$\frac{450 + 158}{2 \times 450} = 0,676$$

Tabela 4.3: Contagem do número de FLOP do *stencil* TTI 3D

Termos	Multiplicador	Somas	Multiplicações	FLOP
P_{xx}, P_{yy}, P_{zz}	3 ×	8	5	13
Q_{xx}, Q_{yy}, Q_{zz}	3 ×	8	5	13
P_{xy}, P_{yz}, P_{xz}	3 ×	63	16	79
Q_{xy}, Q_{yz}, Q_{xz}	3 ×	63	16	79
$s^2\theta c^2\phi$	1 ×	0	2	2
$s^2\theta s^2\phi$	1 ×	0	2	2
$s\theta c\phi s\theta c\phi$	1 ×	0	3	3
coef_A	1 ×	3	0	3
coef_B	1 ×	3	1	4
coef_C	1 ×	5	9	14
coef_D	1 ×	5	10	15
P3	1 ×	4	3	7
Q3	1 ×	4	2	6
Total		450	158	608

As *flags* de compilação utilizadas foram as seguintes:

- `-mmic`
- `-openmp`

- `-O3`
- `-ldl`

As variáveis de ambiente configuradas foram as seguintes:

- `export LD_LIBRARY_PATH=$TBBROOT/../compiler/lib/mic`
- `export OMP_NUM_THREADS=228`
- `export KMP_AFFINITY=balanced,granularity=fine`

Tabela 4.4: Contagem do número de *loads* e *stores* do *stencil* TTI 3D

Termos	Multiplicador	*loads*	*stores*	Total
P_{xx}, P_{yy}, P_{zz}	3x	1	1	2
Q_{xx}, Q_{yy}, Q_{zz}	3x	1	1	2
P_{xy}, P_{yz}, P_{xz}	3x	1	1	2
Q_{xy}, Q_{yz}, Q_{xz}	3x	1	1	2
$s^2\theta c^2\phi$	1x	1	0	1
$s^2\theta s^2\phi$	1x	1	0	1
$s\theta c\phi s\theta c\phi$	1x	1	0	1
Vpx,Vpn,Vsz	3x	1	0	1
P2	1x	1	1	2
P3	1x	1	1	2
Total		20	14	34

Tabela 4.5: Resultados dos parâmetros de *roofline* por análise de código para o *stencil* TTI 3D

Versão	IA	GFlop/s	
	(Flop/byte)	*Sandy Bridge*	*Xeon Phi*
Valor máximo	4,47	632	586
Valor máximo ponderado	4,47	427	396

4.1.2 Calculando a IA usando o VTune

Cada arquitetura possui registradores que permitem descrever a performance. Para a arquitetura Sandy Bridge, o evento `SIMD_FP_256.PACKED_SINGLE` mostra a contagem de FLOP e `CPU_CLK_UNHALTED.THREAD` a contagem de ciclos de *clock*. Considerando que o problema é *memory-bound*, o tráfego de dados pela memória é obtido pelo valor de pico de *bandwidth*. Já para a Xeon Phi, o evento destinado a observar a contagem de operações é `VPU_ELEMENTS_ACTIVE`. A descrição destes eventos é apresentada abaixo e pode ser conferida nos manuais das arquiteturas [33][71]. Esses valores podem mostrar uma contagem superestimada do número de operações, por adicionarem algumas operações de movimentação de dados [72].

- `SIMD_FP_256.PACKED_SINGLE` - Conta a quantidade de micro-operações de precisão simples usando AVX-256, por ciclo.
- `CPU_CLK_UNHALTED.REF_P` - Conta o número de ciclos de um *clock* de referência (133 MHz) para os quais a *thread* não está em estado de espera (isto é, não está executando a instrução `HLT`).
- `VPU_ELEMENTS_ACTIVE` - Conta a quantidade de elementos vetorizados que são processados, a nível de *thread*.
- `VPU_INSTRUCTIONS_EXECUTED` - Conta a quantidade de vezes que uma operação SIMD é executada, a nível de *thread*.

O tempo de execução pode ser obtido diretamente por uma função da linguagem usada (para este trabalho, feito em C, usou-se a função `gettimeofday`), mas também pode ser calculado usando a métrica `CPU_CLK_UNHALTED.REF_P`, através da equação (4.8), onde t_{exec} é o tempo de execução, f_{clock} é a frequência da CPU e $N_{threads}$ é o número de *threads*. Para calcular a quantidade de FLOP para a arquitetura Sandy Bridge, usa-se a equação (4.9), que multiplica o evento pela quantidade de elementos dentro de um vetor SIMD (para precisão simples, 8 elementos). Para Xeon Phi, a quantidade de FLOP já é dada diretamente pela métrica `VPU_ELEMENTS_ACTIVE`, pois já computa operações para elementos vetorizados.

$$t_{exec} = \frac{\texttt{CPU\_CLK\_UNHALTED.REF\_P}}{f_{clock} \times N_{threads}} \tag{4.8}$$

$$FLOP_{SB} = N_{SIMD} \times \texttt{SIMD\_FP\_256.PACKED\_SINGLE} \tag{4.9}$$

$$FLOP_{phi} = \texttt{VPU\_ELEMENTS\_ACTIVE} \tag{4.10}$$

4.2 VTune

4.2.1 Introdução

O programa utilizado para perfilagem do código é o VTune, que possui alguns levantamentos pré-determinados a respeito da performance do código, sendo capaz de identificar o trecho do código associado a determinados eventos de *hardware*. A análise mais básica é a de *Hotspots*, que mostra a distribuição do tempo de execução por trecho de código (Figura 4.3), identificando quais são os responsáveis por limitar a velocidade de execução. Além disso, mostra a distribuição de *threads* usadas ao longo da execução do programa(Figura 4.4). As análises mais avançadas são *General Exploration* e *Bandwidth*. A análise de *General Exploration* (Figura 4.5) mostra a contagem de registradores de desempenho presentes nos processadores, como a quantidade de *cache misses* e *hits*, isto é, quantas instruções conseguiram aproveitar dados na memória *cache* e quantas precisaram de um nível mais afastado. Essas métricas auxiliam na medida da intensidade aritmética e da quantidade de operações. A análise de *Bandwidth* mostra o pico de transferência de dados entre a memória e a CPU e o comportamento dessa transferência (Figura 4.6).

A Figura 4.3 mostra, como era esperado, que o gargalo do programa está no laço onde as operações aritméticas acontecem, e a cor verde mostra que houve bom aproveitamento das *threads*. Quando a cor predominante é vermelha, deve-se procurar melhorias no paralelismo, combinando as técnicas da seção 3.2. A Figura 4.4 mostra que uma parte significativa do tempo foi executada usando o máximo de *threads* disponível para o programa, portanto, um bom uso do recurso. A região vermelha mostra que houve um tempo significativo em execução com poucas *threads*, o que caracteriza a parte intrinsicamente serial do programa. A Figura 4.6 mostra que o comportamento do programa é de transferência de dados intensa alternada com

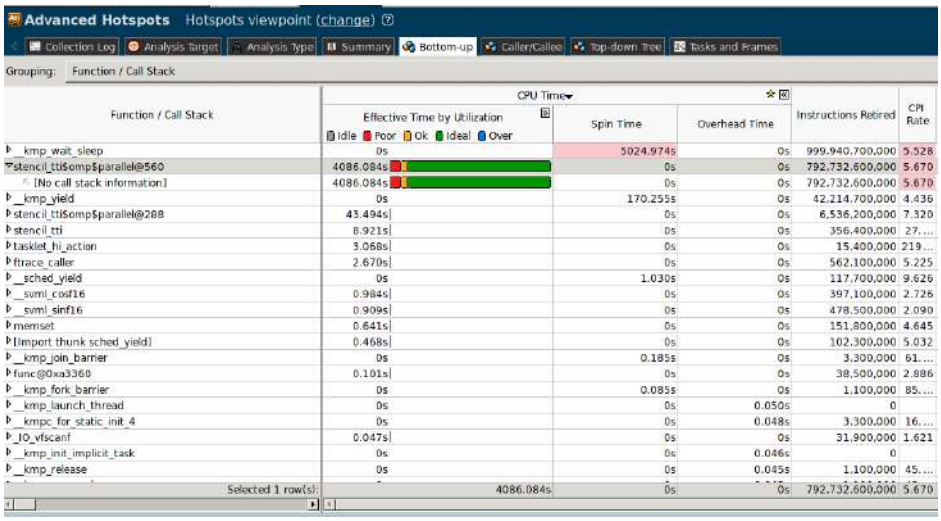

Figura 4.3: Análise de *Hotspots* do VTune

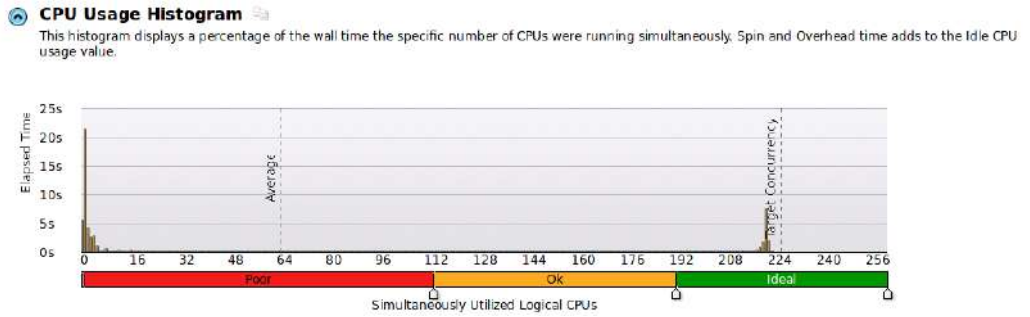

Figura 4.4: Quantidade de *threads* em uso

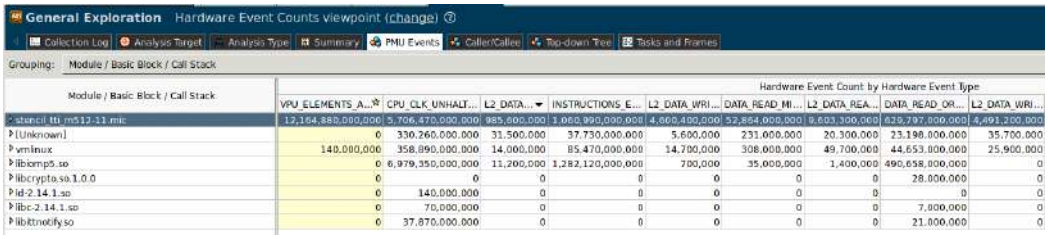

Figura 4.5: Análise de *General Exploration* do VTune

trechos sem nenhuma transferência, o que reflete a característica do código, que realiza as operações aritméticas (áreas vazias) e transfere os dados calculados entre as matrizes P e Q (áreas pretas).

4.2.2 Uso do programa

Existem dois modos de utilizar o programa: pela linha de comando ou usando a interface gráfica. A execução sem coprocessador é mais simples, com acesso di-

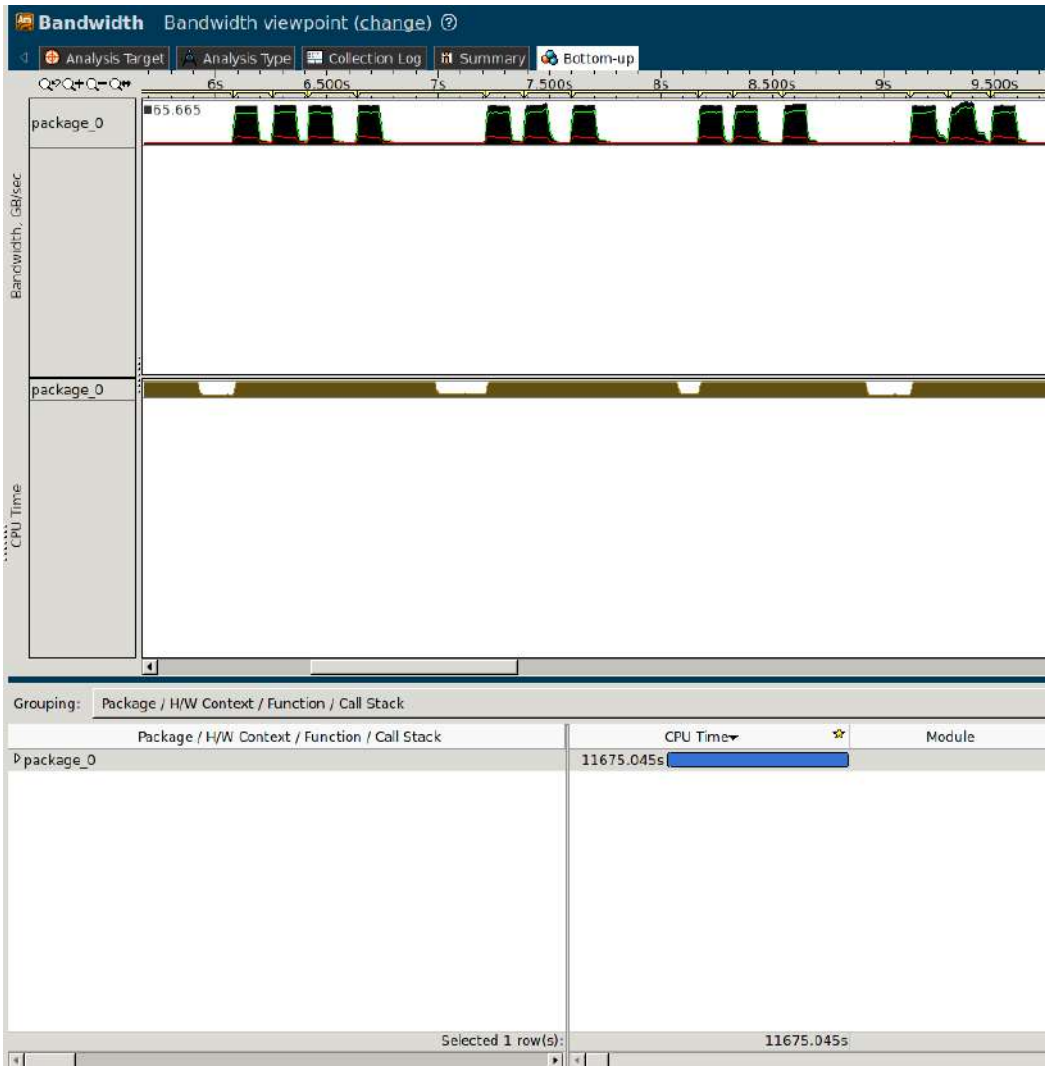

Figura 4.6: Análise de *Bandwidth* do VTune

reto do VTune ao executável. Quando se trata de uma perfilagem usando o co-processador Xeon Phi, apesar de ser possível, através de um `ssh -X`, acessar o VTune gráfico remotamente, executando o modo MIC native, é mais usual executar o VTune através de um *script* , sendo necessária a configuração de algumas variáveis de ambiente (nesse caso, usou-se o modo `mic-host-launch`). A variável `LD_LIBRARY_PATH` indica o diretório onde as bibliotecas do compilador podem ser encontradas, especificamente, para a arquitetura MIC. As variáveis `KMP_FOR_TPROFILE` e `KMP_FORKJOIN_FRAMES_MODE` devem receber o valor 1, para que o VTune possa delimitar a região de OpenMP do código (Figura 4.7). Durante a compilação, é necessário que se use a *flag* `-g` (versão de *debug*) para que o VTune interprete o binário de acordo com as linhas de código, permitindo que haja o detalhamento da performance, linha

por linha.

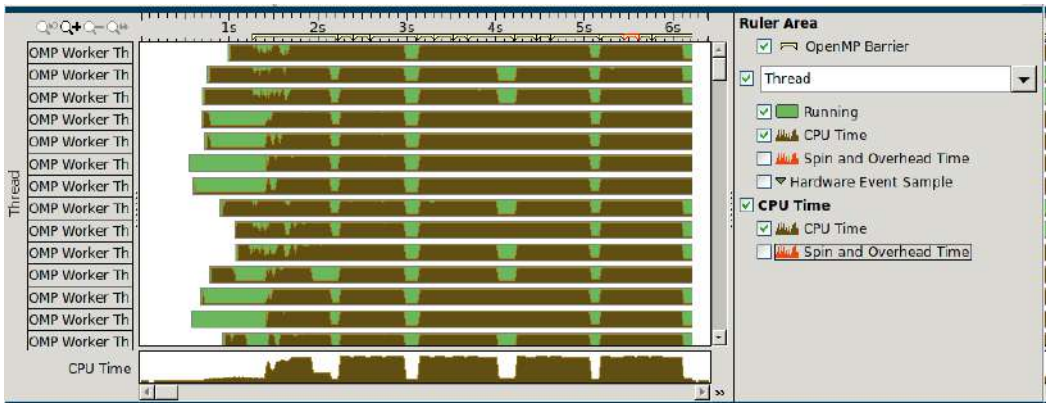

Figura 4.7: Barreiras do OpenMP no VTune

4.3 Validação

A seguir, são comparados os valores de FLOP na seção 4.1.1 com os obtidos usando as métricas de perfilagem apresentadas na seção 4.1.2 usando um *benchmark* de FMA (conforme [32]) e o *stencil* isotrópico de ordem 16. A expectativa é que sejam reproduzidos os valores calculados de FLOP/s e a quantidade de transferência de dados em *cache* de um problema conhecido, o algoritmo isotrópico 2D, através da execução pelo *software* de perfilagem.

As Tabelas 4.6 e 4.7 mostram os valores obtidos das métricas do VTune. A coluna "dimensões" mostra o tamanho da discretização; a coluna "t_{med}" representa o valor do tempo de execução medido diretamente pela função `gettimeofday`; "t_{calc}", o tempo de acordo com a equação (4.8), tomando como base a métrica `CPU_CLK_UNHALTED.REF_P`; e t_{graf} foi o tempo observado de acordo com o gráfico do VTune, onde é mostrada a linha de tempo da execução do programa, como pode ser visto na parte inferior da Figura 4.8. Os resultados foram observados pela interface gráfica do VTune, selecionando uma área de execução do *loop* entre as várias repetições (Figura 4.8), onde mais de uma área do *loop* foi selecionada e os valores das Tabelas são uma média desses valores (na Figura 4.8, foram selecionados 27 *loops*, portanto, os valores de `CPU_CLK_UNHALTED`, `SIMD_FP_256` e t_{graf} foram divididos por 27).

Tabela 4.6: Parâmetros medidos pelo VTune para o *stencil* isotrópico 2D (Sandy Bridge)

Dimensões	CPU_CLK_UNHALTED	t_{med} (ms)	t_{calc} (ms)	t_{graf} (ms)	SIMD_FP_256
2000 x 2000	171407664	2,23	2,35	2,17	94666809
2304 x 900	96857288	1,31	1,33	1,28	49942932
752 x 301	127143045	0,150	0,175	0,150	5857157

Tabela 4.7: Parâmetros medidos pelo VTune para o *stencil* isotrópico 2D (Xeon Phi)

Dimensões	CPU_CLK_UNHALTED	t_{med} (ms)	t_{calc} (ms)	t_{graf} (ms)	VPU_ELEMENTS_ACTIVE
2000 x 2000	150193774	1,79	2,40	0,906	255613670
2304 x 900	72105371	52,6	1,15	0,495	132526713
752 x 301	7600011	0,0849	0,121	0,075	14200043

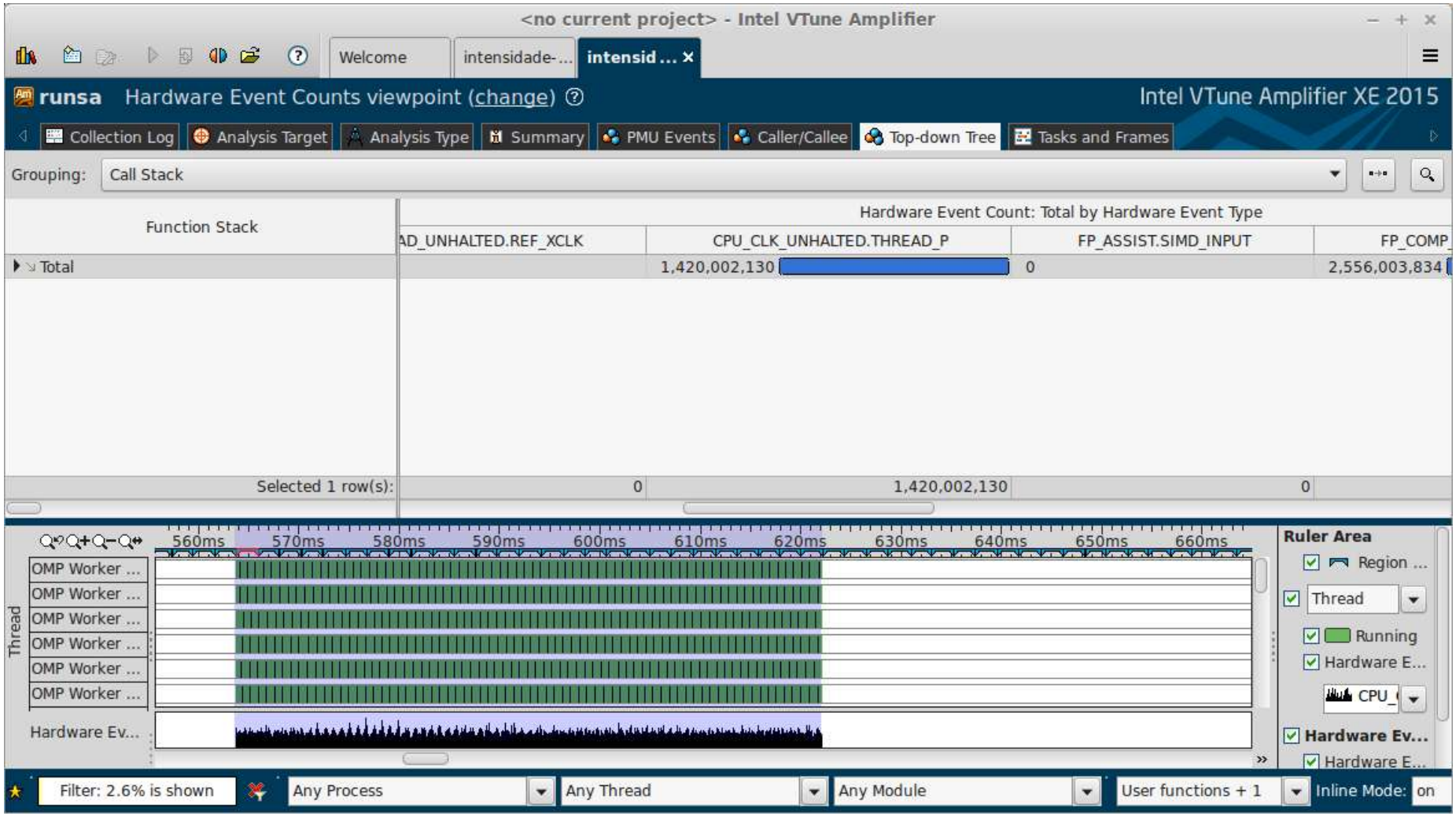

Figura 4.8: Obtenção dos dados da Tabela 4.6 pela interface gráfica do VTune

Com os dados apresentados, usando como referência as equações (4.8), (4.9) e (4.10), calculam-se os valores da performance, em FLOP/s, e da largura de banda, que são apresentados nas Tabelas 4.8 e 4.9 junto aos dados calculados, para demonstrar a aplicabilidade do método utilizado.

Tabela 4.8: Métricas derivadas dos contadores de eventos para o *stencil* isotrópico 2D (Sandy Bridge)

Dimensões	GFLOP/s (previsto)	GFLOP/s (medido)	IA (prevista)
2000 x 2000	79,4	322	2,81 FLOP/byte
2304 x 900	69,5	300	2,81 FLOP/byte
752 x 301	62,9	268	2,81 FLOP/byte

Tabela 4.9: Métricas derivadas dos contadores de eventos para o *stencil* isotrópico 2D (Xeon Phi)

Dimensões	GFLOP/s (previsto)	GFLOP/s (medido)	IA (prevista)
2000 x 2000	98,7	107	2,81 FLOP/byte
2304 x 900	173	115	2,81 FLOP/byte
752 x 301	111	117	2,81 FLOP/byte

Analisando os dados da Tabela 4.8 e 4.9, observa-se que os valores são condizentes com os esperados. Porém, na Tabela 4.9, o valor do tempo de execução é maior que o tempo de execução da aplicação, sem a perfilagem sendo executada, o que é mostrado na Tabela 4.11. Uma estratégia foi obter o número de operações usando a metodologia proposta, através do VTune, porém, o tempo de execução a ser utilizado no cálculo de Flop/s seria o tempo de execução da aplicação sozinha, sem a intervenção do VTune. Este resultado é mostrado na Tabela 4.12, onde se nota que, utilizando esta abordagem, as métricas de Flop/s obtidas acabam por ser superiores ao limite permitido pela aplicação mostrados na Tabela 4.2 de 369 GFlop/s. Para fins de comparação, as diferenças de tempos para a plataforma Sandy Bridge também foram calculadas e mostradas na Tabela 4.10.

Tabela 4.10: Diferenças de tempo de execução do *stencil* isotrópico 2D com e sem o VTune (Sandy Bridge)

Dimensões	Sem VTune	Com VTune	Diferença	Diferença Percentual
2000 x 2000	2,12 ms	2,23 ms	0,11 ms	5,2 %
2304 x 900	1,30 ms	1,31 ms	0,01 ms	0,77%
752 x 301	0,125 ms	0,150 ms	0,025 ms	20 %

Tabela 4.11: Diferenças de tempo de execução do *stencil* isotrópico 2D com e sem o VTune (Xeon Phi)

Dimensões	Sem VTune	Com VTune	Diferença	Diferença Percentual
2000 x 2000	0,691 ms	1,79 ms	1,10 ms	160 %
2304 x 900	0,419 ms	0,526 ms	0,107 ms	25,6 %
752 x 301	66,7 μs	84,9 μs	18,1 μs	27,2%

Tabela 4.12: Medida de GFlop/s para o *stencil* isotrópico 2D com e sem o VTune (Xeon Phi)

Dimensões	GFlop/s Sem VTune	GFlop/s Com VTune	Diferença (GFlop/s)	Diferença Percentual
2000 x 2000	370	107	263	71,1 %
2304 x 900	316	115	201	63,6 %
752 x 301	213	117	95,6	44,9 %

Capítulo 5

Resultados e Discussões

No capítulo anterior, foram mostrados os resultados para a medida de GFLOP/s para o *stencil* isotrópico 2D utilizando o cálculo manual da quantidade de operações e através de contadores presentes nos processadores. Neste capítulo, serão adicionados os resultados para o *stencil* TTI 3D, além da análise do consumo energético.

5.1 Perfilagem do *stencil* TTI 3D

As Tabelas 5.1 e 5.2 mostram o resultado das métricas de performance, obtidas através do VTune, para o problema TTI 3D, apresentado de forma similar ao problema ISO 2D do capítulo anterior. Nota-se que há diferenças entre os tempos de execução do programa através das três medidas apresentadas, confirmando a influência do VTune no desempenho do programa.

Tabela 5.1: Parâmetros medidos pelo VTune para o *stencil* TTI 3D (Xeon Phi)

Dimensões	CPU_CLK_UNHALTED	t_{med} (ms)	t_{calc} (ms)	t_{graf} (ms)	VPU_ELEMENTS_ACTIVE
240 x 236 x 464	10413853459	0,396	0,703	0,0108	21069684830

5.2 Escalabilidade

Para medir a influência do paralelismo na eficiência de um programa, define-se a métrica do *Speedup*, ou seja, a relação entre o tempo de execução utilizando uma

Tabela 5.2: Métricas derivadas dos contadores de eventos para o *stencil* TTI 3D (Xeon Phi)

Dimensões	GFLOP/s (previsto)	GFLOP/s (medido)	IA
240 x 236 x 464	339	304	4,47 FLOP/byte

única *thread* pelo tempo utilizando múltiplas *threads*.

$$S_p = \frac{t_{serial}}{t_{paralelo}} \tag{5.1}$$

No caso ideal, a razão entre os tempos de execução é exatamente a quantidade de *threads* utilizada, isto é, um sistema perfeitamente paralelo. Porém, na prática, isto não ocorre. Uma grandeza correlata é a eficiência, mostrada pela equação (5.2), que mede o quão longe ou próximo do ideal o paralelismo ficou, isto é,

$$E_p = \frac{S_p}{p} \tag{5.2}$$

onde p é o número de *threads* e S_p é o *speedup* definido pela equação (5.1).

As Figuras 5.1 e 5.2 foram geradas executando o código do *stencil* isotrópico 2D, variando o número de *threads* e o tamanho do problema, respectivamente no processador Sandy Bridge e Xeon Phi. Em ambas as figuras, nota-se que o comportamento é parecido independente do tamanho do problema, porém, para a arquitetura Sandy Bridge, o comportamento é especialmente mais similar, com as curvas praticamente sobrepostas. Já para a Xeon Phi, o *Speedup* alcançado foi maior para os problemas maiores, possivelmente pelo aumento do número de *threads* característico da arquitetura.

A Figura 5.3 foi gerada usando a mesma ideia da Figura 5.2, executando o código do *stencil* TTI 3D. Observa-se que há um ganho significativo no primeiro quarto do gráfico, isto é, até 57 *threads*, onde é usado apenas o paralelismo com *threads* físicas. A partir dessa região, está sendo usado o *hyperthreading*, tendo ganho significativo até o último quarto, onde o ganho fica estagnado. Ou seja, para o problema mencionado, o *hyperthreading* com 2 *threads* virtuais produziu ganho aproximadamente

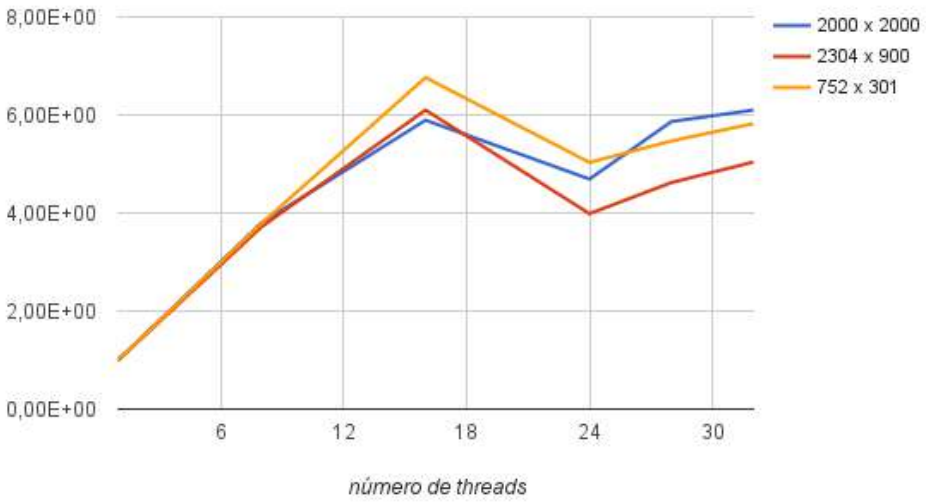

Figura 5.1: *Speedup* do *stencil* isotrópico 2D na arquitetura Sandy Bridge

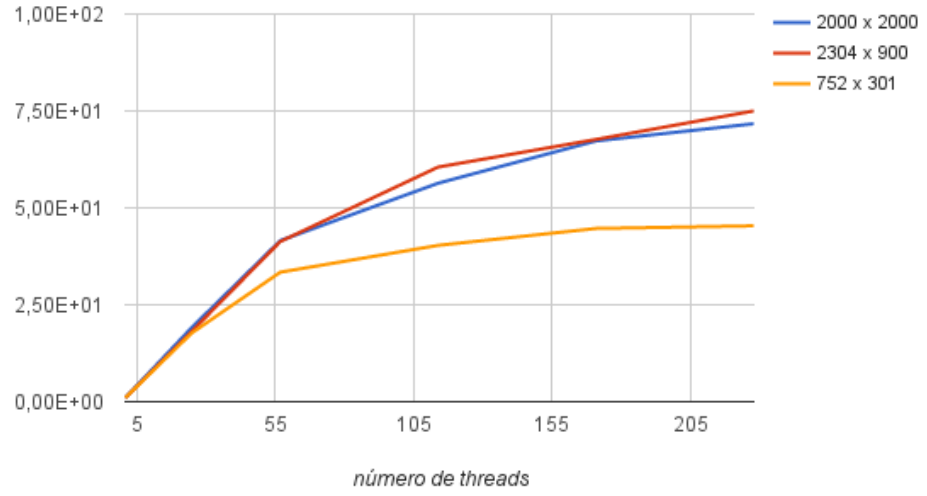

Figura 5.2: *Speedup* do *stencil* isotrópico 2D na arquitetura Xeon Phi

linear, porém, para 3 *threads* virtuais, o ganho não foi observado.

Ao otimizar um código, é comum associar a performance com o número de processadores e com o tamanho do problema[36], de maneira que se torna inviável falar sobre *Speedup* quando essas duas variáveis estão envolvidas. Para isto, utiliza-se o conceito de escalabilidade, o qual associa a quantidade de processadores ou *threads* ao tamanho do problema. A escalabilidade pode ser desmembrada em dois tipos: a Escalabilidade Forte e a Fraca. A Escalabilidade Forte está diretamente direcionada ao *speedup*, isto é, um programa tem escalabilidade forte quando o tempo de execução é inversamente proporcional à quantidade de *threads* utilizadas.

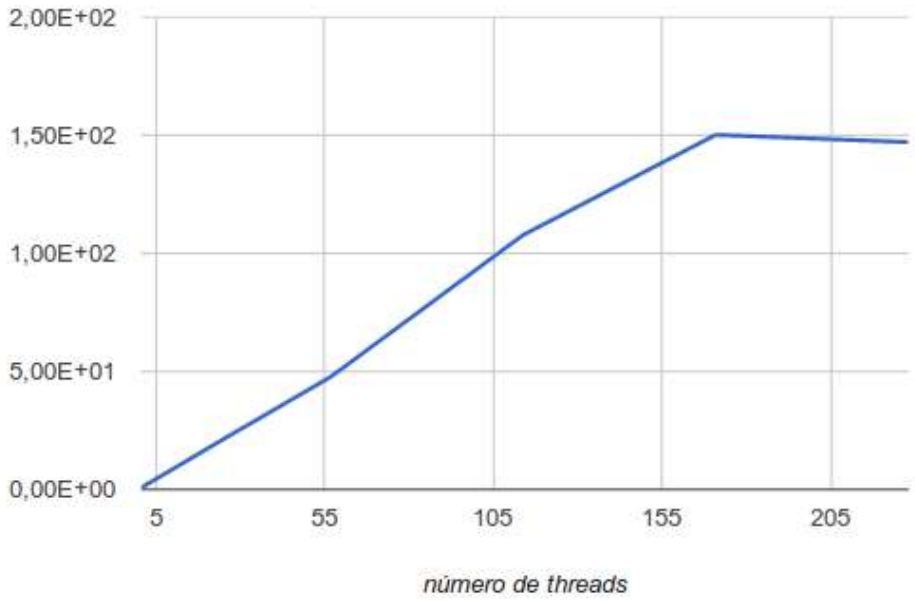

Figura 5.3: *Speedup* do *stencil* TTI 3D na arquitetura Xeon Phi

A Escalabilidade Fraca relaciona não só o tempo de execução ao número de processadores ou *threads*, mas também o tamanho do problema. Um programa possui Escalabilidade Fraca se, ao aumentar o tamanho do problema e a quantidade de processadores de maneira tal que a quantidade de dados por processador fique constante, a quantidade de operações por segundo em cada processador também fica constante [36].

5.3 *Roofline*

Obtendo os valores das quantidades de operações e utilizando como base a Figura 4.2, é possível marcar a posição do código desenvolvido no gráfico de *roofline*, gerando a Figura 5.4. Observa-se que o desempenho obtido é próximo do patamar superior da arquitetura. Deve-se levar em conta que o gráfico é aproximado, devido aos diversos fatores já mencionados ao longo deste texto. Primeiramente, as medidas apresentadas pelo texto original levam em conta a intensidade operacional em vez de aritmética[3], porém, esta é uma medida de execução difícil. Os textos de referência utilizados (como [31], [52] e [73]) adotam, para o modelo de *Roofline*, a intensidade aritmética, porém, mesmo essa medida provoca discrepância dependendo do método usado, além de poucas referências para comparações e validação,

trazendo dificuldades para gerar resultados quantitativos e para reproduzir outros resultados da literatura. Por fim, o cálculo de FLOP é incerto, pois, tanto o cálculo por análise de código, quanto por contagem de métricas de *hardware* podem ser mascarados por otimizações dos processadores.

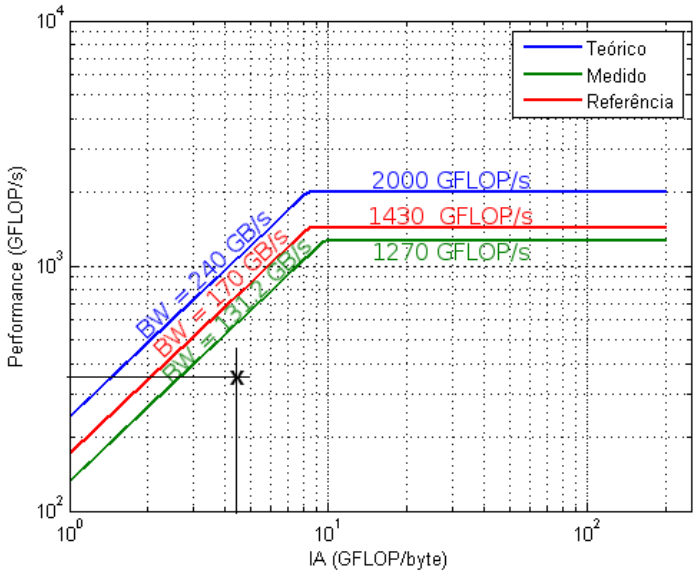

Figura 5.4: *Roofline* para a Xeon Phi 3120P, incluindo marcação do desempenho obtido

5.4 Consumo Energético

O consumo energético foi medido para uma execução completa do código TTI 3D na Xeon Phi, gerando valores de potência e de temperatura, mostrados, respectivamente, nas Figuras 5.5 e 5.6. Nota-se que a potência aumenta rapidamente no início da execução do código e volta ao seu valor de estado energético ocioso, com uma média calculada, durante a execução, de 182 W.

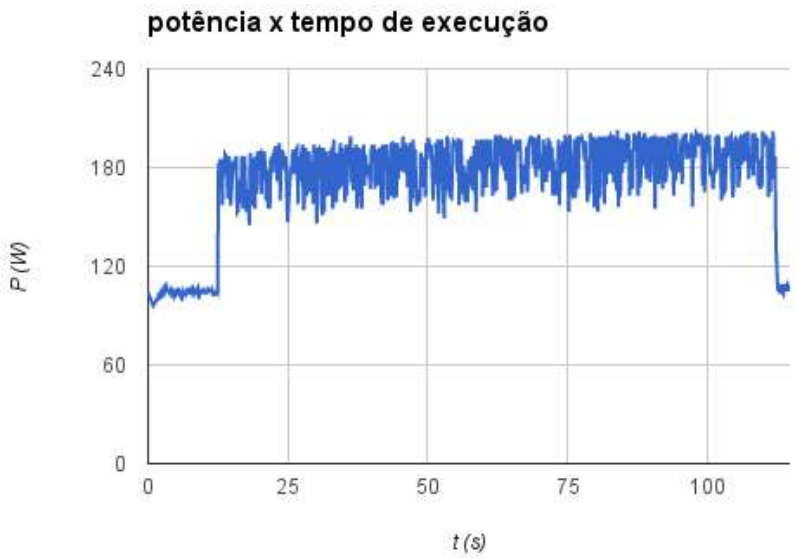

Figura 5.5: Curva do consumo energético do código TTI 3D na Xeon Phi

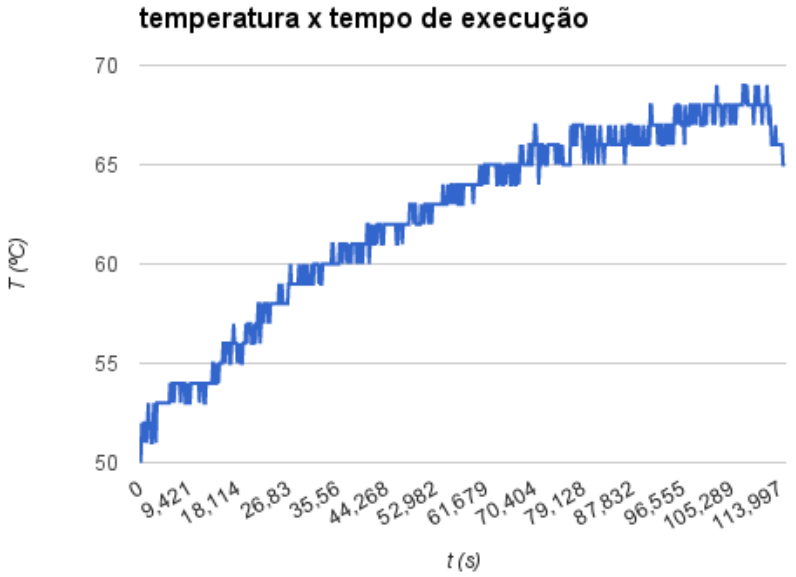

Figura 5.6: Curva da temperatura na Xeon Phi durante a execução do código TTI 3D

Capítulo 6

Conclusões e Trabalhos Futuros

A arquitetura MIC utilizada pelo coprocessador Xeon Phi se mostrou uma forma de melhorar o desempenho sem ser necessário adicionar um nó ao sistema, sendo possível melhorar o desempenho com apenas um nó, sem utilizar as técnicas de MPI. Este trabalho e o trabalho desenvolvido com MPI [13] indicam que as técnicas podem ser utilizadas em conjunto, com melhoria no desempenho. Os ambientes multicore e multinó são padrão da Computação de Alto Desempenho e são muito adequados à solução da equação da onda pelo MDF, sendo fundamental que, para o avanço da HPC nas universidades brasileiras, novos trabalhos sejam desenvolvidos em cima do problema estudado. Os valores de FLOP/s obtidos foram pouco confiáveis quantitativamente, porém, tem um valor qualitativo precioso, pois é possível observar os valores de duas (ou mais) versões de um programa e comparar qual obteve um valor maior sendo, portanto, melhor.

Concluiu-se, também, que o código desenvolvido explorou muito dos limites de desempenho do equipamento utilizado. Apesar disso, de acordo com o modelo de *roofline* obtido, ainda não se localiza no limite físico do desempenho. Apesar da promessa da Intel de obter ganho em desempenho com pouco esforço na modificação do código, é necessário um entendimento profundo das técnicas de otimização para conseguir explorar os limites do equipamento. Ainda assim, a inclusão do coprocessador melhora significativamente o desempenho, além de aproveitar melhor os recursos físicos, com relação ao espaço físico numa sala de *cluster* e com relação ao consumo energético. O consumo energético médio obtido para a execução do código

TTI 3D foi de 182 W, a quantidade de FLOP/s calculada foi de 304 GFLOP/s, resultando em 1,67 GFLOP/s/W, e a IA foi de 4,47 FLOP/byte. Para obter o valor para a Sandy Bridge, será necessária a instalação de programas de perfilagem de temperatura como os apresentados na seção 3.4 e este comparativo ficará para trabalhos futuros. Entretando, é possível comparar com um valor obtidos da literatura de 18,4 MFLOP/s/W[74]. Portanto, a eficiência energética da Xeon Phi foi muito maior que a da Sandy Bridge, estando em um patamar de GFLOP/s/W da posição número 128 da lista Green500[26]. Vale notar que, como pode ser visto na Figura 5.5, existe um patamar de consumo sem carga da Xeon Phi de aproximadamente 110 W, de modo que, quando a aplicação está sendo executada, o valor aumentou, na média, em 72 W.

Para trabalhos futuros, sugere-se que seja feita a comparação de consumo energético com as arquiteturas mais recentes da Intel, tanto do *host*, quanto da Xeon Phi (KNL - Knights Landing), assim como da NVIDIA. Para os modelos de *roofline*, pode-se realizar as medidas do VTune utilizando os *benchmarks* de FLOP/s e de largura de banda, ou, também, utilizar outros perfiladores. Por fim, é necessário que se realize um trabalho de estudo nas medidas de intensidade aritmética, pois os valores apresentados na literatura são escassos e pouco reprodutíveis. Assim como foi feito neste trabalho com as medidas de FLOP/s, é desejável um trabalho para as medidas de intensidade aritmética, utilizando *benchmarks* e diferentes abordagens, como a medida de BW do VTune, contagem manual de variáveis e contagem, também através de medidas de registradores de desempenho, dos *cache misses*.

Referências Bibliográficas

[1] M. Araya-Polo, F. Rubio, R. de la Cruz, M. Hanzich, J. M. Cela, and D. P. Scarpazza, "3d seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors," *Sci. Program.*, vol. 17, pp. 185–198, Jan. 2009.

[2] D. Butler, "The petaflop challenge," *Nature*, vol. 448, pp. 6–7, 2007.

[3] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insigthful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[4] J.-T. Krüger, "Green wave: A semi-custom hardware architecture for reverse time migration," 2012.

[5] C. H. dos Santos Barbosa, "Migração reversa no tempo em meios transversalmente isotrópicos," Master's thesis, Universidade Federal do Rio de Janeiro, 3 2015.

[6] L. Thomsen, "Weak elastic anisotropy," *Geophysics*, vol. 51, pp. 1954–1966, 1986.

[7] F. Ortigosa, M. A. Polo, F. Rubio, J. M. Cela, R. de la Cruz, and M. Hanzich, "Evaluation of 3d rtm on hpc platforms," *Society of Exploration Geophysicists*, 2008.

[8] "Intel ® vtune™amplifier — intel ® software." `https://software.intel.com/en-us/intel-vtune-amplifier-xe`. Acessado: 11/11/2016.

[9] B. de Souza Silva, "Avaliação de operadores convolucionais na solução da equação acústica da onda," Master's thesis, Universidade Federal do Rio de Janeiro, 3 2014.

[10] D. Costa, A. L. G. A. Coutinho, B. S. Silva, J. J. Silva, and L. Borges, "A trade-off analysis between high-order seismic rtm and computational performance tuning," in *Pan-American Congress on Computational Mechanics*, 2015.

[11] I. R. Mufti, "Large-scale three-dimensional seismic models and their interpretive significance," *Geophysics*, vol. 55, pp. 1166–1182, 1990.

[12] A. Bulcão, *Modelagem e Migração Reversa no Tempo Empregando Operadores Elásticos e Acústicos.* PhD thesis, UFRJ, Rio de Janeiro-RJ, Outubro 2004.

[13] S. C. Jorge, "Explorando paralelismo híbrido na propagação da onda acútica 3d," Master's thesis, Universidade Federal do Rio de Janeiro, 9 2016.

[14] R. Norman, "The form and laws of propagation of seismic wavelets," *Geophysics*, vol. 18, no. 1, pp. 10–40, 1953.

[15] A. Reynolds, "Boundary conditions for numerical solution of wave," *Geophysics*, vol. 43, pp. 1099–1110, 1978.

[16] C. Cerjan, D. Kosloff, and M. Reshef, "A nonreflecting boundary condition for discrete acoustic and elastic wave equations," *Geophysics*, vol. 50, pp. 705–708, 1985.

[17] W. S. Duarte, *Uma metodologia para extrapolação de ângulo de reflexão em profundidade utilizando matrizes de tempo de trânsito.* PhD thesis, UFRJ, Rio de Janeiro-RJ, Fevereiro 2011.

[18] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[19] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, 1988.

[20] R. Rahman, "Intel xeon phi application design and implementation considerations." http://intel.ly/1G0mOEs. Acessado: 31/03/2017.

[21] "Tp500 supercomputer sites." http://www.top500.org. Acessado: 18/11/2016.

[22] "Oak ridge national laboratory." https://www.ornl.gov/content/seventy-years-great-science. Acessado: 31/03/2017.

[23] "Lawrence livermore national laboratory." http://computation.llnl.gov/research/research-and-development. Acessado: 31/03/2017.

[24] "Riken advanced institute for computational science." http://www.aics.riken.jp/en/science/research-highlights. Acessado: 31/03/2017.

[25] S. Sharma, C.-H. Hsu, and W.-c. Feng, "Making a case for a green500 list," Disponível em https://www.top500.org/files/green500/hp-pac2006.pdf, Acessado: 31/03/2017.

[26] "The green500." https://www.top500.org/green500/list/2016/06/. Acessado: 18/11/2016.

[27] "Eletrobras eletronuclear." http://www.eletronuclear.gov.br/Saibamais/Perguntasfrequentes/Angra1desempenhoeprodu%C3%A7%C3%A3o.aspx. Acessado: 31/03/2017.

[28] "The openmp api specification for parallel programming." http://www.openmp.org/. Acessado: 04/04/2017.

[29] "qopenmp, qopenmp — intel ® software." https://software.intel.com/en-us/node/682582. Acessado: 04/04/2017.

[30] "qopenmp-stubs, qopenmp-stubs — intel ® software." https://software.intel.com/en-us/node/522966. Acessado: 04/04/2017.

[31] C. Andreolli, P. Thierry, L. Borges, G. Skinner, and C. Yount, "Characterization and optimization methodology applied to stencil computations," in *High Performance Parallelism Pearls* (J. R. Jeffers, ed.), pp. xv – xxxvii, Boston: Morgan Kaufmann, 1 ed., 2015.

[32] R. Rahman, *Intel ® Xeon Phi™ Coprocessor Architecture and Tools*. ApressOpen, 1 ed., 2013.

[33] "Intel ® xeon phi™ coprocessor datasheet," April 2014. Number: 328209.

[34] L. Palma, "Arquitetura do coprocessador intel ® xeon phi™ amplifier - intel software conference 2013." `https://pt.slideshare.net/IntelSoftwareBR/isc-2013811arquitetura-intelxeonphi`. Acessado: 06/03/2017.

[35] C. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, and R. McGuire, "Offload compiler runtine for the intel ® xeon phi™ coprocessor," Disponível em `https://software.intel.com/sites/default/files/ee/90/offload-compiler-runtime-for-the-intel-xeon-phi-coprocessor-130315.pdf`, Acessado: 04/04/2017.

[36] V. Eijkhout, *Introduction to High Performance Scientific Computing*. 2 ed., 2014. Disponível em `http://www.tacc.utexas.edu/~eijkhout/istc/istc.html`, Acessado: 31/03/2017.

[37] `https://computing.llnl.gov/tutorials/linux_clusters/`. Acessado: 31/03/2017.

[38] "Intel support - processors: Define sse2, sse3 and sse4." `http://www.intel.com/support/pt/processors/sb/cs-030123.htm`. Acessado: 31/03/2017.

[39] "Isa extensions intel avx — intel ® developer zone." `https://software.intel.com/en-us/isa-extensions/intel-avx`. Acessado: 31/03/2017.

[40] G. Chrysos, "Intel ® xeon phi^{TM}coprocessor - the architecture." https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner. Acessado: 31/03/2017.

[41] "Intel ® 64 and ia-32 architectures optimization reference manual," September 2014. Order Number: 248966-030.

[42] https://software.intel.com/en-us/node/522935. Acessado: 31/03/2017.

[43] H. Fu, R. G. Clapp, O. Lindtjorn, T. Wei, and G. Yang, "Revisiting finite difference and spectral migration methods on diverse parallel architectures," *Computers & Geosciences*, vol. 43, no. 0, pp. 187 – 196, 2012.

[44] M. Christen and O. Schenk, "A performance study of an anelastic wave propagation code using auto-tuned stencil computations.," in *ICCS* (H. H. Ali, Y. Shi, D. Khazanchi, M. Lees, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, eds.), vol. 9 of *Procedia Computer Science*, pp. 956–965, Elsevier, 2012.

[45] "Using intel openmp thread affinity for pinning - hecc knowledge base." http://www.nas.nasa.gov/hecc/support/kb/Using-Intel-OpenMP-Thread-Affinity-for-Pinning_285.html. Criado: 12/07/2011, Acessado: 31/03/2017.

[46] A. Srinivasa, M. Sosonkina, P. Maris, and J. P. Vary, "Dynamic adaptations in ab-initio nuclear physics calculations on multicore computer architectures," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, (Washington, DC, USA), pp. 1332–1339, IEEE Computer Society, 2011.

[47] R. Iyer, H. Wang, and L. Bhuyan, "Design and analysis of static memory management policies for cc-numa multiprocessors," tech. rep., College Station, TX, USA, 1998.

[48] "Intel ® c++ intrinsic reference," 2007. Document Number: 312482-003US.

[49] J. D. Wieber-Jr and G. M. Zoppetti, "How to use intrinsics — intel ® developer zone." https://software.intel.com/en-us/articles/how-to-use-intrinsics. Acessado: 31/03/2017.

[50] D. Costa, L. Borges, L. Leite, C. Barbosa, and J. Silva, "77th eage conference & exhibition (madrid ifema, spain, 1-4 june 2015)," Half-precision Performance Benefits on a Xeon Phi TTI RTM Application, (Madrid, Spain), 2015.

[51] R. Steinmann, "Applying the roofline model," Master's thesis, ETH Zürich.

[52] C. Andreolli, P. Thierry, L. Borges, C. Yount, and G. Skinner, "Genetic algorithm based auto-tuning of seismic applications on multi and manycore computers," in *EAGE Workshop on High Performance Computing for Upstream*, 2014.

[53] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 4:1–4:12, IEEE Press, 2008.

[54] M. Louboutin, M. Lange, F. Herrmann, N. Kukreja, and G. Gorman, "Performance prediction of finite-difference solvers for different computer architectures," *CoRR*, vol. abs/1608.03984, 2016.

[55] D. Imbert, K. Imadoueddine, P. Thierry, H. Chauris, and L. Borges, *Tips and tricks for finite difference and i/o less FWI*, ch. 620, pp. 3174–3178.

[56] "Acpi- advanced configuration & power interface." http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf. Acessado: 31/03/2017.

[57] T. Kidd, "Power management states: P-states, c-states, and package c-states." http://intel.ly/1EbqG6C. Acessado: 31/03/2017.

[58] "Likwidpowermeter: Tool for accessing rapl counters and quert turbo mode steps on intel processor." https://code.google.com/p/likwid/wiki/LikwidPowermeter. Acessado: 31/03/2017.

[59] "Xeon e5 v3 haswell-ep performance." https://www.pugetsystems.com/labs/articles/Xeon-E5-v3-Haswell-EP-Performance----Linpack-595/. Acessado: 31/03/2017.

[60] "Boston labs - intel ® xeon ® e5-2600 v2 tested." http://www.bostonlabs.co.uk/boston-labs-intel-xeon-e5-2600-v2-tested-part-ii/. Acessado: 31/03/2017.

[61] http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-linpack-stream.html. Acessado: 31/03/2017.

[62] http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html. Acessado: 12/01/2016.

[63] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[64] http://ark.intel.com/pt-br/products/81705/. Acessado: 31/03/2017.

[65] http://www.karlrupp.net/2015/02/stream-benchmark-results-on-intel-xeon-and-xeon-phi/. Acessado: 31/03/2017.

[66] http://ark.intel.com/pt-br/products/75798/. Acessado: 31/03/2017.

[67] http://www.7-cpu.com/cpu/SandyBridge.html. Acessado: 31/03/2017.

[68] J. Fang, A. L. Varbanescu, H. J. Sips, L. Zhang, Y. Che, and C. Xu, "An empirical study of intel xeon phi," *CoRR*, vol. abs/1310.5842, 2013.

[69] https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture. Acessado: 31/03/2017.

[70] J. Krueger, D. Donofrio, J. Shalf, M. Mohiyuddin, S. Williams, L. Oliker, and F.-J. Pfreund, "Hardware/software co-design for energy-efficient seismic modeling," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 73:1–73:12, ACM, 2011.

[71] "Intel ® 64 and ia-32 architecturessotware developer's manual," June 2015. Number: 253669.

[72] `http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops`. Acessado: 04/11/2016.

[73] P. Souza, T. Teixeira, L. Borges, A. Neto, C. Andreolli, and P. Thierry, "Reverse time migration with heterogeneous multicore and manycore clusters," in *EAGE Workshop on High Performance Computing for Upstream*, 2014.

[74] J. Hofmann and D. Fey, "An ecm-based energy-efficiency optimization approach for bandwidth-limited streaming kernels on recent intel xeon processors," in *Proceedings of E2SC 2016: 4th International Workshop on Energy Efficient Supercomputing - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 31–38, 2016.