Análise e Paralelização de um algoritmo para Gridding

Carlos A. T. Aguni[1] **, Daniel Cordeiro**[2]

[1]Escola Politécnica, Universidade de São Paulo

[2]Escola de Artes, Ciências e Humanidades, Universidade de São Paulo

{carlos.aguni,daniel.cordeiro}@usp.br

Abstract. *This paper analyzes the performance of a parallel implementation for the Gridding Method Algorithm in two diferent approaches: Intel® Xeon Phi Coprocessor (Many Cores architecture) and Nvidia Tesla K20x (General Purpose Graphics Processing Unit - GPGPU). We study its suitability when optimizing its serial implementation into a multi/many core environment as well as profiling its resource usage on memory, CPU, and cache usage on both CPU and accelerator.*

Resumo. *Este artigo analisa o desempenho do algoritmo de Gridding implementado de forma paralelizada em duas propostas: com uma placa aceleradora Intel® Xeon Phi (arquitetura Many Cores) e uma Nvidia Tesla K20x Unidades de Processamento Gráfico de Propósito Geral (General Purpose Graphics Processing Unit - GPGPU). Estudamos sua adequabilidade quando otimizado para um ambiente multi/many core e o perfilamos em relação ao consumo de recursos como memória, processador, e uso de caches na CPU e placas aceleradoras.*

1. Introdução

O *Square Kilometre Array* (SKA [Dewdney et al. 2009]) é o maior projeto de radiotelescópios do mundo e tem como objetivo reconstruir imagens do céu a partir dos dados coletados de várias antenas. O projeto, constituído por duas fases, iniciou-se em 2011 e pretende impulsionar o desenvolvimento científico em várias áreas da ciência. Tecnologias de satélites, processamento de sinais e métodos de computação e infraestrutura foram combinados para otimizar tanto o desempenho da execução quanto armazenar os dados do *workload* [SDP 2018].

A primeira fase do projeto finalizou-se em 2017 e conta com cerca de 250 antenas que simulam um grande radiotelescópio extremamente sensível [T.J. Cornwell 2012] capaz de coletar dados observados em uma área de um quilômetro quadrado do espaço. As informações capturadas pelas antenas correspondem a uma quantidade de dados que chegam a uma razão de aproximadamente 5,2 Terabytes por segundo.

Os satélites, uma vez configurados, são então acionados para capturar imagens do céu por períodos prolongados de horas a dias gerando, portanto, uma quantidade de dados economicamente inviável de ser armazenada em qualquer infraestrutura existente. Tal problema trouxe a necessidade de tratar os dados antes de seu armazenamento. No entanto, o poder computacional necessário para se tratar essa quantidade de dados é estimado na casa do PetaFLOPs [Dewdney and et. al 2010] na primeira fase do projeto e ExaFLOPs na segunda. Ainda espera-se que a primeira fase permita a captura e o tratamento de apenas 10% da capacidade total do projeto.

O processo do tratamento dos dados, desde a sua captação bruta até o seu armazenamento final, é composto por tarefas computacionalmente intensivas como: a correlação no processamento das imagens [Jacobs 2005], Convolução (*Convolutional Gridding* [Beatty and Pauly 2005]), Transformada de Fourier (FFT) [Frigo and Johnson 2005], etc. Este trabalho devota-se a estudar a etapa de Convolução, normalmente referenciada na literatura pelo seu termo em inglês *gridding*.

A computação do *gridding* é considerada um dos maiores desafios do projeto SKA [Braam and Wortmann 2016, Cornwell 2006]. *Convolutional gridding* é uma técnica utilizada em interferometria (combinação de n antenas) para converter as medições feitas pelas n antenas em dados numa malha regular. Em seguida, somente após calculada a inversa da transformada de Fourier, é possível obter algo próximo da imagem final do espaço. Vale ressaltar que, para realizar a FFT é necessário que a malha esteja disposta em uma grade regular, por isso a importância da convolução. O processo de convolução seguida da transformada de Fourier é, além de bastante estudado por astrônomos, também, muito utilizado nas áreas de ciências médicas, especialmente associado ao uso de máquinas de ressonância magnética, tomografia, etc. [O'Sullivan 1985, A. Eklund and LaConte 2013].

Vistos o ecossistema, a importância, relevância e fatores de gargalo do algoritmo *gridding* o presente artigo propõe um estudo de otimização e análise do algoritmo em arquiteturas de computadores orientadas ao total de trabalho realizado por unidade de tempo (*throughput-oriented architecture*). Uma abordagem diferente à computação convencional nas CPUs onde a importância está no tempo em que cada tarefa é executada (*latency-oriented architecture*).

O objetivo deste trabalho é otimizar a execução e análise do algoritmo de *convolutional gridding*, desenvolvido e utilizado pelo projeto SKA, utilizando métodos de programação paralela apresentando uma caracterização detalhada do consumo de recursos nos sistemas em que foram executados. A implementação proposta foi estudada em termos de suas relações carga/memória, carga/CPU e tempos de execução, e sua implementação baseia-se em uma divisão eficiente da carga das tarefas entre os recursos computacionais nas diferentes arquiteturas avaliadas.

Os *hardwares* orientado à tarefas (*throughput-oriented programming*) utilizados foram dois: Intel Xeon Phi [Fang et al. 2013] e Nvidia Tesla [Huang et al. 2008]. A utilização de tais placas aceleradoras não pode ser feita de forma independente, ou seja, elas trabalham em conjunto com a CPU (*latency-oriented programming*), portanto, as versões paralelas implementadas para ambas arquiteturas contemplam, também, otimizações paralelas realizadas na CPU. Por fim, a partir da versão serial foram criadas duas versões paralelas do algoritmo, uma para cada arquitetura de placa aceleradora:

1. $Máquina^{1}_{hospedeira} + Máquina^{1}_{coprocessador^{1}}$
2. $Máquina^{2}_{hospedeira} + Máquina^{2}_{coprocessador^{2}}$

Logo, além da versão serial já existente, o artigo concentra-se em descrever os métodos de paralelização, escolha de parâmetros (quantidade de *threads*, memória, cache) e *profiling* para as versões que envolvem a placa aceleradora e sua respectiva máquina hospedeira. O artigo encontra-se organizado da seguinte forma. A seção 2 explica com maiores detalhes o algoritmo de *convolution gridding*. A seção 3 introduz sobre o ambiente utilizado e as seções 4 e 5 apresentam de que forma o algoritmo fora paralelizado e

quais foram os resultados obtidos respectivamente e, por fim, as conclusões discutidas na seção 6.

2. Contexto

De forma sintética, o algoritmo analisado sintetiza as imagens do céu recuperadas pelas antenas e as combinam em uma única imagem final. Tal processo pode ser descrito e separado em três partes principais: a inicialização, o *gridding* e a finalização.

A inicialização envolve todo o processo de pré-configuração da malha final onde serão armazenados os dados referentes ao espaço. Cada antena realiza as medições em uma sistema de coordenadas UVW representada na Figura 1. Nessa fase também são geradas as funções de convolução que serão utilizadas na fase do *gridding*. Na etapa seguinte, é realizada a re-amostragem convolucional, ou *gridding*, cujo objetivo é juntar em uma só malha os dados obtidos pelas antenas; é esta etapa que o presente trabalho se propõe em otimizar. Por fim, durante a etapa de finalização é executada a inversa da transformada de Fourier e, em seguida, é feito o armazenamento da imagem final no *storage*. Tais passos podem ser vistos na Figura 2 onde a área azul representa o algoritmo de *gridding*.

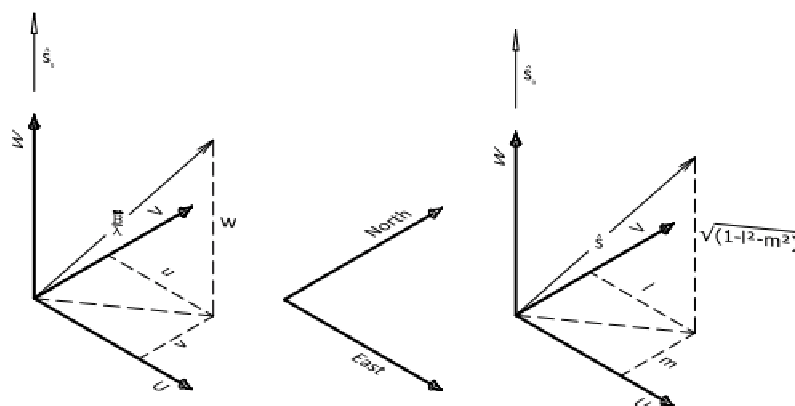

Figura 1. Representação do sistema de coordenadas UVW. A esquerda a sua representação em componentes (u, v, w) e a direita quando apontado para um objeto celeste. Imagem retirada de [Muscat 2014].

Note que há vários outros métodos [Muscat 2014, Sørensen et al. 2008] que permitem a transformação dos sinais recebidos pelos telescópios em dados coerentes para serem analisados. *Convolution gridding* [Humphreys and Cornwell 2011] foi o método escolhido pelo projeto SKA.

2.1. Descrição do algoritmo de *Gridding*

Transformadas de Fourier podem ser rapidamente calculadas através do algoritmo conhecido como *Fast Fourier Transform* (FFT). Mas seu uso impõe como pré-condição que os dados estejam dispostos em intervalos regulares. Quando os dados de entrada não são regulares, faz-se necessário realizar um pré-processamento para torná-los regular.

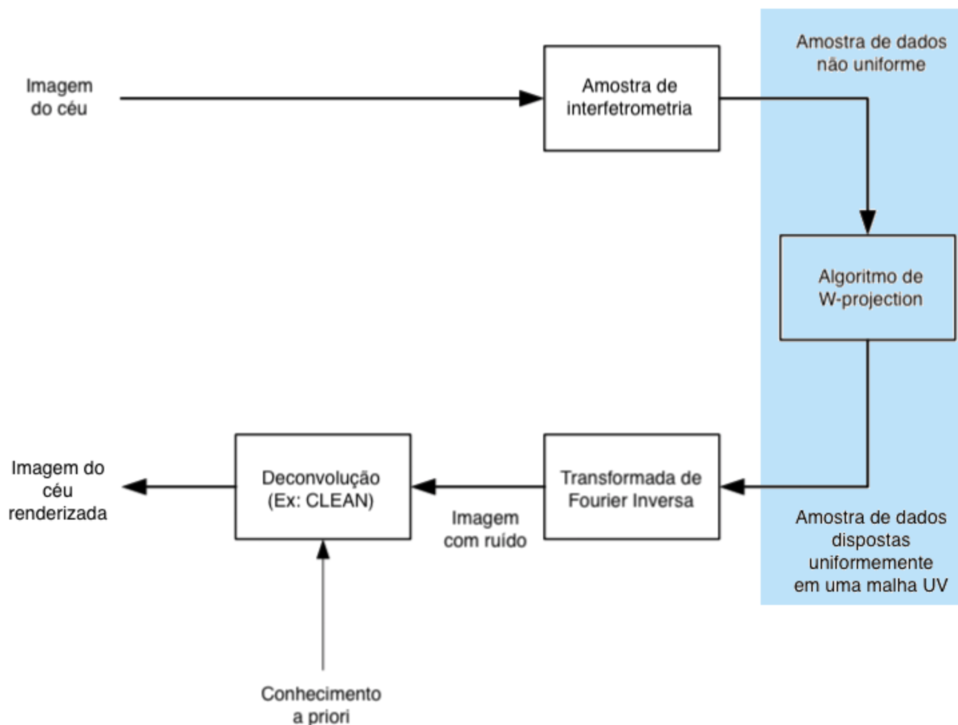

Figura 2. Procedimento de captura, processamento e armazenamento das imagens capturadas do espaço. Adaptada de [Muscat 2014].

Convolution gridding é um dos métodos utilizados para regularizar os dados. Através dele é possível realizar a convolução de espaços irregulares de dados para regular. Realizar a convolução significa passar a malha de dados por um filtro (função de convolução representado na Figura 3). Funções de convolução são bastante utilizadas na área da física quando a intenção é de se atenuar ou remover ruído como por exemplo de um áudio, ou imagem, etc. No SKA é utilizado o método conhecido como *W-projection* [Cornwell 2004], capaz de juntar n malhas em uma única [Muscat 2014].

O algoritmo de *gridding* pode ser expresso de forma relativamente simples, como demonstrado no pseudocódigo 1. O algoritmo descrito dessa forma, entretanto, possui muitas dependências de dados. Por exemplo, existe uma dependência de dados no vetor `grid`, onde uma tentativa ingênua de paralelização do cálculo dos *samples* acarretaria em uma situação de concorrência no acesso aos dados caso o `d_index` fosse modificado por duas ou mais *threads*.

2.2. Descrição do *workload*

Utilizamos como base da implementação paralela o código serial disponibilizado pela *Australia Telescope National Facility* (projeto CSIRO) em sua conta no GitHub [Humphreys et al. 2013]. O código compreende também um conjunto de testes automatizados que utilizamos com a finalidade de verificar a corretude da versão paralela.

O *workload* utilizado, disponibilizado junto com o código, contém 3.200.000 amostras, onde cada amostra possui 536MB no formato de uma malha 2D de tamanho

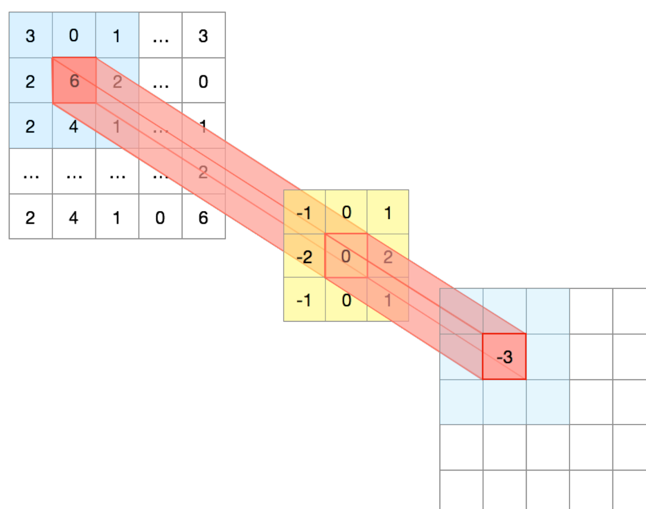

Figura 3. Representação da utilização de uma função de convolução.

Pseudocódigo 1. Pseudocódigo do algoritmo de *gridding* onde cada *sample* seria a saída de uma observação de cada telescópio. O `d_index` armazena sua posição da malha única e `c_index` armazena a posição que iniciará os valores da função de convolução.

```
1  entrada: malha , função de convolução, amostras
2  saida: malha atualizada
3  inicio
4      para cada: amostra em amostras (índice d_index itera sobre amostras)
5          g_index ← Posicão inicial na malha da amostra (u, v, w)
6          c_index ← Posição inicial na função de convolução
7
8          para: suppv até comprimentoSup
9              peso ← Peso da amostra da função de convolução
10             para: suppu até comprimentoSup
11                 malha[g_index++] = += peso * cfunc[c_index++]
12             g_index += comprimentoMalha
13     fim
14     retorna malha
15 fim
```

$4096 \times 4096 = 16.777.216$ elementos que servirão de entrada do algoritmo de *Gridding*. O projeto também disponibiliza um programa chamado *verify* que computa o erro entre os resultados de diferentes versões utilizando a norma L^1 do vetor a partir da fórmula $S = \sum_{i=1}^{n} |y_i - f(x_i)|$, onde o resultado **(S)** é a soma das diferenças absolutas entre o resultado (Y_i) e o valor estimado ($f(x_i)$). Este trabalho utiliza-se do *verify* para assegurar a corretude da implementação.

3. Características do Ambiente de Testes

Estudamos a paralelização do algoritmo usando tanto GPUs como Intel Xeon PHIs. A análise experimental foi feita utilizando os seguintes recursos computacionais:

- AMD Opteron(tm) Processor 6380 2,5GHz + 1 Placa Nvidia Tesla K20x.
- Intel Xeon E5-2650 2,6 Ghz + 1 Placa Intel Xeon PHI 5110P 60 Cores

A Tabela 1 apresenta uma comparação ente as placas utilizadas neste trabalho.

Para fins de desambiguação, iremos utilizar os termos:

	K20x	Phi 5110P
# Processadores	2688	60
Proc. Velocidade	732 MHz	1,05GHz
Memória RAM Total	6 GB	8 GB
Chipset	GK110	Knights Corner

Tabela 1. Comparação entre as placas Nvidia K20x e Xeon Phi 5110P utilizadas.

1. GPU para se referir às placas aceleradores Nvidia
2. MIC (Intel® *Many Integrated Cores*) para se referir à Xeon Phi.
3. $HOST_{GPU}$ para se referir ao nó computacional que inicia o código a ser executado na GPU
4. $HOST_{MIC}$ para se referir ao nó computacional que inicia o código a ser executado na MIC
5. "Versão híbrida": $HOST_X + X, X \in \{MIC, GPU\}$ onde X refere-se à arquitetura sendo discutida naquela seção ou subseção salvo exceções onde a máquina é então especificada ou quando faz-se menção à ambas as máquinas.

A arquitetura Xeon Phi oferece dois modelos de programação paralela: *native* e *offloading*. No primeiro, o código é executado nativamente no coprocessador e suas dependências precisam ser previamente carregadas na MIC. No segundo, a aplicação é executada no HOST e partes do código são carregadas no coprocessador.

O presente trabalho opta pelo uso do modo *offloading* para a otimização da execução do algoritmo devido à maior flexibilidade para organizar os dados do workload, o que permite melhor controle de sincronização de dados entre HOST e MIC como também transferência de trechos de código pré-escritos para serem executados na MIC sob demanda. Note que os recursos de uma MIC são limitados, então a ideia é aproveitá-la o máximo possível tanto em termo de processamento, quanto em termos de memória, e não usá-la para E/S de dados. Utilizamos OpenMP (paralelização por *threads*) para habilitar paralelismo na máquina.

Uma GPU permite apenas o modo *offloading*. O modo de acesso e execução em uma placa GPU é muito semelhante à MIC, uma vez que também é necessário explicitar quais componentes do código serão executados na GPU e é necessário gerenciar manualmente quais dados serão transferidos para/das placas.

No HOST existem dois modos principais de obter-se paralelismo: com o uso de *threads* ou com o uso de processos (via *fork*). Esses modos são facilitados com o uso de bibliotecas que implementam um conjunto de funcionalidades de controle e de troca de mensagens de mais alto nível, tais como: OpenMP, que oferece um conjunto de diretivas (*pragmas*) e programação paralela com *threads*, e MPI (*Message Parsing Interface*) que oferece uma biblioteca com um conjunto de funções para controle de processos. Os prós e contas em se utilizar um ou outro dependerá sempre da natureza do problema em questão. Para a otimização do algoritmo de *gridding*, visto a necessidade de haver um controle maior sobre o uso de memória *intra-node*, optou-se pela utilização do OpenMP.

4. Estratégias de paralelização

Após um estudo do algoritmo serial de *Gridding* implementado pela *Australia Telescope National Facility*, e a partir da utilização de ferramentas de *profiling*, observamos que

um único trecho de código, responsável pela soma e multiplicação de números complexos, estava ocupando quase todo o tempo de processamento da aplicação. Sendo assim, decidimos focar os esforços em diminuir o tempo computacional neste trecho.

A versão sequencial precisou ser adaptada para poder ser executada nas aceleradoras. A versão original, escrita em C++, utiliza de uma grande quantidade de estruturas de dados complexas oferecidas pela linguagem (listas ligadas, objetos, etc.), enquanto que as aceleradores só permitem o uso de tipos de dados primitivos do C. A adaptação não só facilitou a manipulação e transferência de dados, como também trouxe um significativo ganho de desempenho.

A versão híbrida do algoritmo leva em conta a possibilidade de execução de forma assíncrona de determinados trechos do código, que podem ser executados nas aceleradoras. Antes da execução, porém, os dados devem ser tratados e, então, copiados para as aceleradoras. Após a execução, os resultados devem então ser convertidos para as estruturas de dados principais do programa.

O processo de paralelização híbrida segue três etapas. Na primeira etapa é necessário aplicar uma solução de paralelização no HOST, seguida por uma solução na MIC/GPU para só depois mesclar as duas. Tais fases facilitam a organização do código, depurações, execuções de testes, etc.

A seção a seguir descreve o perfil de desempenho da versão sequencial e mostra os fatores que guiaram o processo de paralelização para execução no HOST, Xeon PHI e GPU.

4.1. Paralelização no HOST

Através da ferramenta Intel VTune Amplifier[1] foi possível encontrar pontos de gargalo na execução serial da função de convolução. Dos resultados obtidos destacamos que do total do tempo de execução da aplicação, de 508 segundos, o código gastou 273 segundos na linha 11 do pseudocodigo 1 mais interna do "*for*"aninhado responsável pelo processo de convolução da malha.

A versão paralela para execução no HOST demonstrou possuir uma grande concorrência de dados nos valores da malha quando várias *threads* são executadas ao mesmo tempo. Detectou-se, por exemplo, um cenário onde dois $ginds$ (índice da malha) eram escolhidos no mesmo instante, acarretando a atualização dos valores no mesmo local por duas *threads* concorrentes.

Para resolver o problema da concorrência, a solução foi criar uma cópia da malha para cada nova *thread* criada. Desse modo, cada *thread* teria sua própria malha temporária para trabalhar sequencialmente e a atualização das malhas temporárias na malha principal seria feita apenas no fim. Sendo assim, a complexidade de espaço utilizado pela aplicação seria de modo linear para cada *thread* criada igual a $\mathcal{O}$(n). Dado que cada ponto na malha necessita de um tipo complexo, temos que o uso total de RAM na máquina pode ser calculada pela fórmula: $RAM_{\text{utilizada}} = nThreads * tamanhoMalha * 2 * 8$.

Com a criação de cópias das malhas para novas *threads* foi possível dividir a carga de trabalho do número de amostras para ser executada de forma paralela entre as

[1] https://software.intel.com/intel-vtune-amplifier-xe

n *threads*. Como podemos ver na Figura 4 a quantidade de amostras da esquerda agora pode ser paralelizada.

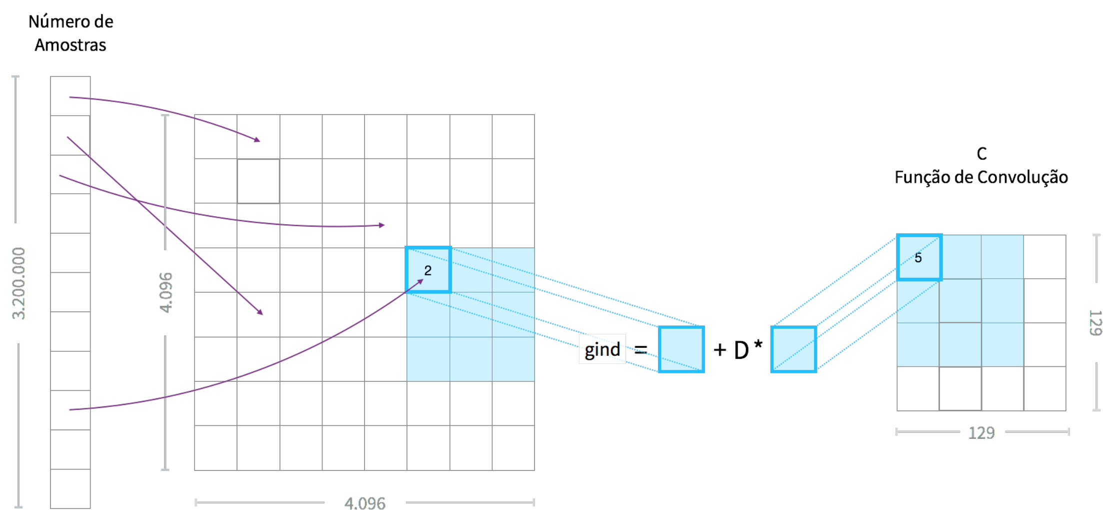

Figura 4. Funcionamento do algoritmo de *gridding*. O número de amostras à esquerda é endereçado pelo d_index 1 que representa sua posição inicial na malha.

4.2. Paralelização na Xeon PHI e GPU

A estratégia de paralelismo para a Xeon PHI e GPU foi a mesma, o uso do modo *offloading*. A diferença foi o arcabouço de programação paralela usado: OpenMP para a Xeon Phi e CUDA para a GPU. O método de cópia da malha feito nos $HOSTs$ foi mantida mas, percebe-se que as placas aceleradoras possuem muito menos espaço em RAM disponível (7GB) quando comparado ao $HOST$, ou seja, se optássemos por utilizar o mesmo método realizado no $HOST$ para a MIC/GPU, levando-se em conta a possibilidade de criar 240 *threads* ($\#processadores * 4(hyperthread)$) na MIC e 28.672 ($\#threadsporSM(2048) * SMs(14)$) seria preciso $RAM_{utilizada} = 240 * 4096^2 * 2 * 8 = 64GB$ de espaço disponível na MIC e muito maior na GPU.

Sabendo disso já é possível se calcular os limites de cópias que preenchem cada placa. Para a MIC o número de cópias/*threads* não pode passar de $nThreads * 4096^2 * 2 * 8 <= 8GB$, ou seja, $nThreads$ para MIC não pode passar de 22. E para a GPU o número de cópias não pode passar de $nThreads * 4096^2 * 2 * 8 <= 6GB$, ou seja, $nThreads$ não pode passar de 18.

Agora nota-se que o número máximo de *threads* possíveis de se utilizar sem que ocorra concorrência de dados é muito menor que o número máximo de *threads* capaz de serem executadas nas placas. Se executado o algoritmo dessa forma, mesmo o número de *threads* sendo maior que aquelas criadas no $HOST$ percebe-se que o resultado será muito pior que o $HOST$ visto os processadores das placas aceleradoras dotadas de menos instruções e velocidade de processamento.

Agora como atingir a máxima ocupação de *threads* na MIC e GPU? A solução encontrada foi dividir as multiplicações das linhas e colunas do array da função de convolução em tiras para um número novo de threads criadas, ou seja, o número de máximo de amostras a serem computadas dentro das placas nunca será maior que o

número de cópias da malha. No entanto, é possível controlar o número de *threads* que irão processar cada amostra. Por exemplo, consideremos a GPU, o número máximo de amostras calculadas por vez é 18 ($nThreads * 4096^2 * 2 * 8 <= 6GB$). No entanto é possível, por exemplo, criar mais 10 *threads* e torná-las cada uma responsáveis por calcular $\frac{1}{10}$ das linhas da função de convolução. O que pode ser visto na figura 5.

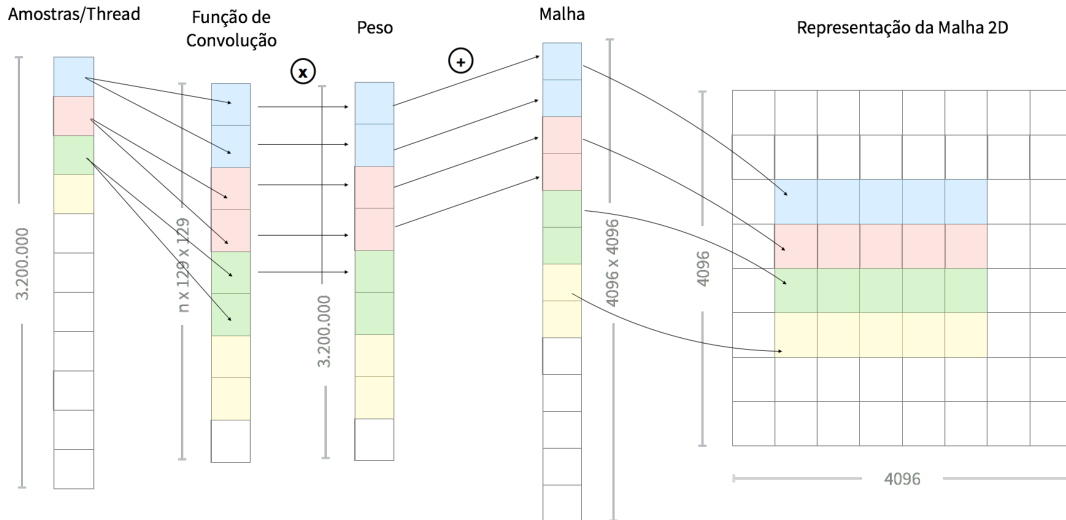

Figura 5. Esquema do fluxo percorrido pela *thread* refatorado para se explorar o *cache*

Outro fator importante comentar foi que a adição de *threads* para dividir a carga de trabalho realizada na função de convolução ajudou a granularizar o problema possibilitando o acesso de cada *thread* a pequenos trechos da memória. Dessa forma o objetivo é encontrar parâmetros de granularidade para que o nível de ocupação na MIC e GPU se aproximem da máxima ao mesmo tempo que se pudesse aumentar a frequência de utilização dos *caches* $L1$ e $L2$ mais próximos do processador. Na figura 6 é possível ver a representação do algoritmo *gridding* que se despreocupa com a utilização eficiente do *cache* o que acaba levando a perdas de *cache* e uma sobrecarga devido a coerência de *caches*.

Comparando-se as figura 5 com a figura 6 é possível apontar algumas vantagens que a versão *cache-friendly* tem da versão que podemos chamar de ingênua: há uma melhor divisão de trabalho para cada *thread*, os *caches* são melhores explorados, é possível controlar cada tira a ser realizada pela sua *thread* respectiva. O próximo passo, agora, seria encontrar um valor $nTiras_x$ e $nTiras_y$ onde cada variável é responsável por calcular a tira, supondo o vector da função de convolução sendo C, $largura_x$ sua largura e $largura_y$ sua altura, a tira responsável por essa *thread* pode ser representada da seguinte forma $C[largura_x : nTiras_x][largura_y : nTiras_y]$ onde o id que representa a *thread* pode ser usado para calcular o deslocamento na linha e/ou coluna. Lembrando que cada valor diferente dado leva a um molde totalmente do problema como pode se ver na figura 7.

Dessa forma, cada *thread* fica responsável por sua respectiva tira da função de convolução o que resulta num maior aproveitamento dos *caches*. Por exemplo, caso a divisão de tarefas fosse feita assumindo-se $largura_x = 129$ e $largura_y = 1$ teríamos um cenário onde cada *thread* se ocuparia em obter $largura_x * largura_y * complex_type =$

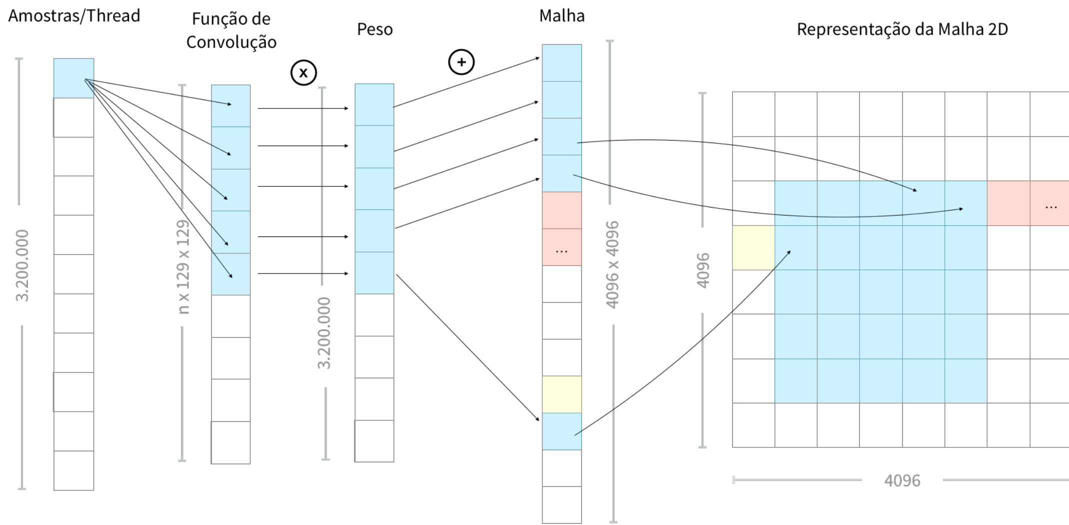

Figura 6. Imagem representando os acessos à memória realizados por uma *thread*. O acesso à próxima linha pela função de convolução não é o mesmo quando este é refletivo para a malha. Vê-se que a mesma *thread* precisa executar um grande "salto"de memória.

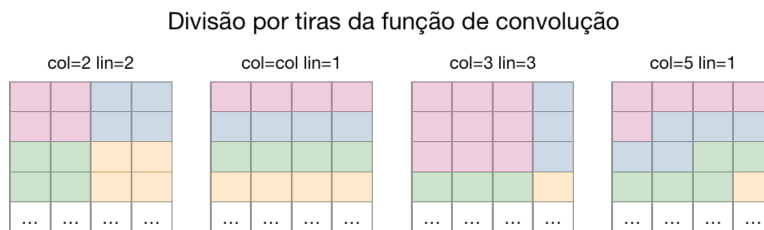

Figura 7. Imagem representa o balanço de carga para cada *thread* variando-se a granularidade da divisão de linhas e colunas.

$129 * 1 * 16 = 2KBytes$. Somando-se à essa *thread* outras constantes de dados e controle, acesso ao vetor de pesos e à malha podemos supor que cada *thread* possui acesso à um tamanho B. A partir das especificações técnicas retiradas da MIC e GPU temos que a MIC possui 32KB de cache L1 e 512KB de cache L2 dividido entre as 4 *hardware threads* de cada processador. Já a GPU possui um cache de até 64KB (cache variável, depende da memória compartilhada) do L1 e 1.5MB para o cache L2. Logo, o que se precisa fazer é escolher um valor de $largura_x$ e $largura_y$ para que o valor de B chegue o mais próximo do cache L1.

A seguir serão apresentados os resultados obtidos a partir da variação da carga de trabalho do número de amostras entre cada uma das duas arquiteturas seguidas de uma análise de como os parâmetros escolhidos impactaram no desempenho da aplicação.

5. Resultados

As execuções nos dois ambientes de testes foram feitos de tal forma que se pudesse dividir o número de amostras a serem calculadas entre a MIC/GPU e seus respectivos HOSTs. A variação da carga de trabalho foi feita manualmente onde cada linha dos gráficos montados abaixo significa uma execução diferente.

Lembrando que foram comparados apenas as execuções realizadas na mesma

máquina, visto que especificações, arquiteturas e variáveis do ambiente (GHz, RAM, número de processadores, versão sistema operacional, marcas diferentes de processadores, versão de compiladores) tornam inviável uma comparação direta a não ser pelo tempo de execução e o quão bem a aplicação escalou comparado com sua versão serial.

Levando-se em conta a corretude dos dados, em todas as execuções, o erro calculado pela versão paralela comparando-se à versão sequencial, o resultado da taxa de erro não ultrapassou 1×10^{-15}.

5.1. Intel Xeon Phi

Figura 8. Tempos de execução da versão paralela no HOST (MIC com 0%) e da versão híbrida (MIC com 10% a 100%).

O código paralelo no $HOST_{MIC}$ foi executado com 16 *threads* e durou 67.27 segundos como pode ser visto na figura 8 quando a carga de trabalho realizada na MIC é 0%, ou seja, aproximadamente três vezes mais rápida que a versão serial de 185.12s. Essa solução paralela ocupou $RAM_{utilizada} = 16 * 4096^2 * 2 * 8 = 4$Gigabytes de RAM

Já na versão híbrida o melhor tempo de execução foi obtido quando dividido 50% da carga das amostras entre a MIC e $HOST_{MIC}$. Tal execução foi encontrada após se definir o número de cópias da malha em $nThreads = 22$, ou seja, o número máximo de amostras a serem executadas na MIC e $largura_x = 1$ e $largura_y = 10$. Logo, o número total de *threads* geradas foi $TotalThreads = nThreads * largura_x * largura_y = 220$ threads geradas.

Logo, comparando-se o melhor tempo obtido na versão híbrida de 29.32 com a versão serial temos um ganho de 6 vezes e 2 vezes quando comparada com a execução do $HOST_{MIC}$.

5.2. Nvida Tesla K20x GPU

Para se avaliar a implementação híbrida na GPU, primeiro implementou-se a versão paralela no $HOST_{GPU}$, que, utilizando 16 processadores foi capaz de reduzir o tempo de

363.12 segundos obtido na execução serial para 202.93 segundos, ou seja, um desempenho de aproximadamente 2 vezes da solução serial.

Já para a versão híbrida, sendo possível dividir a carga de amostras entre GPU e $HOST_{GPU}$ o melhor tempo de execução obtido foi quando 90% da carga de trabalho das amostras foi realizado na GPU atingindo um tempo de execução de 30.05 segundos, ou seja, um desempenho 12 vezes mais rápido quando comparada à versão serial e aproximadamente 7 vezes mais rápido que a solução paralela implementada na $HOST_{GPU}$. As demais variações de carga podem ser vistas no gráfico da figura 9.

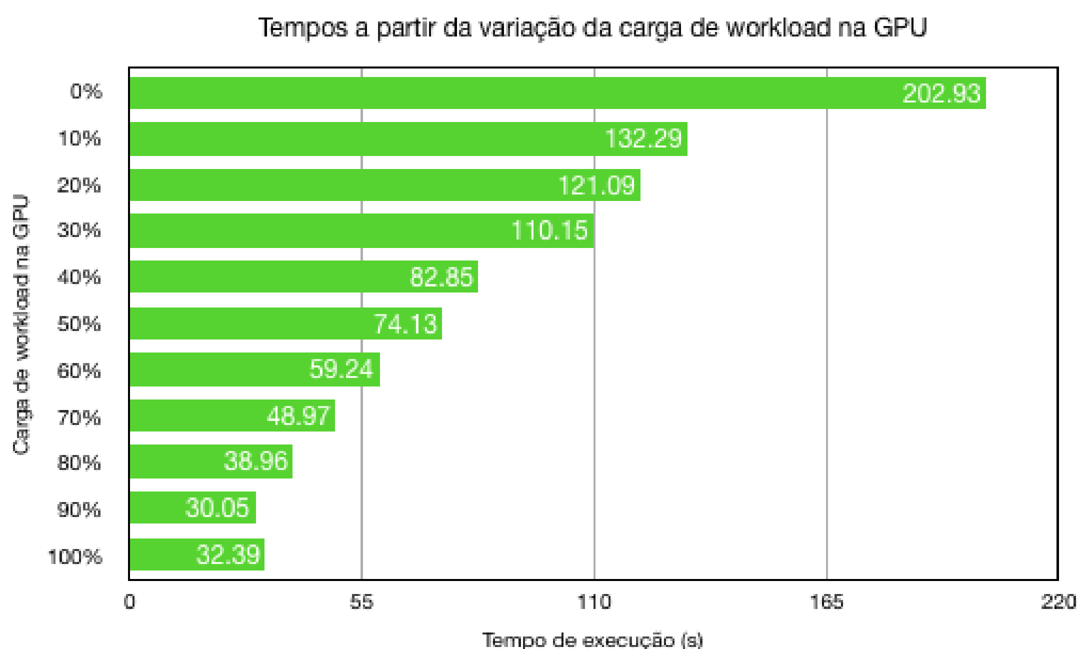

Figura 9. Gráfico do tempo de execução variando-se a carga executada na GPU.

Tal performance na GPU foi obtida quando definido o número de cópias da malha em 18, ou seja, $nThreads = 18$ que também reflete ao máximo de amostras sendo executadas na GPU. Os parâmetros que resultaram no melhor tempo de execução quando 90% da carga de trabalho foi executada na GPU foram $largura_x = 5$ e $largura_y = 129$, ou seja, sendo o tamanho da matriz de convolução 129x129 cada *thread* fica responsável por calcular um vetor de tamanho $[\frac{129}{largura_x}][\frac{129}{largura_y}] = [\frac{129}{5}][\frac{129}{129}] = [\approx 25][1]$ na malha. O número total de *threads* criadas ficou, então, $TotalTreads = nThreads * largura_x * largura_y = 11.610$ threads ou 645 threads por $threadBlock$. Executando-se o Nvidia Visual Profiler [2] foi possível observar que cada *thread* utilizou uma quantidade de 31 registros por *thread* com uma taxa de ocupação de 98.4%.

6. Considerações Finais

O presente artigo mostra uma análise de desempenho e consumo de memória de diferentes implementações paralelas do algoritmo de gridding executadas em diferentes arquiteturas de computadores, no caso HOSTs, MIC e GPU.

Uma análise preliminar do desempenho mostrou a existência de contenção de dados no acesso à malha. Propomos uma cópia temporária de dados para uso local por cada

[2] `https://developer.nvidia.com/nvidia-visual-profiler`

thread que possibilitou a execução de n amostras em paralelo. No entanto, tal abordagem gerou um grande consumo de memória, que se mostraria inviável em sistemas que possuem pouco espaço em memória, como é o caso de placas aceleradoras. Propusemos, por isso, um novo arranjo no acesso à memória que permitiu cada *thread* acessar somente uma área da memória, o que possibilitou o uso eficiente das aceleradoras.

Nossa análise experimental mostrou que ser possível obter um *speedup* de 12 vezes ao se executar 90% da carga de trabalho nos cores da placa GPU. Já na arquitetura Xeon Phi, obtivemos um *speedup* de 6 vezes quando 50% da carga de trabalho é executada na MIC, enquanto o restante é executado no HOST.

A análise do desempenho realizada confirmou nossa hipótese inicial de que o gargalo de desempenho estava relacionado à forma como o algoritmo realiza os acessos à memória. A avaliação do desempenho forneceu indicadores importantes que nos levaram a concluir que o algoritmo paralelo proposto é eficiente para o problema, e que a utilização bem sucedida da memória *cache* foi fundamental para os bons resultados obtidos.

Referências

[A. Eklund and LaConte 2013] A. Eklund, P. Dufort, D. F. and LaConte, S. (2013). Medical image processing on the GPU: Past, present and future. *Med. Image Anal.*, vol. 17 pp. 1073-1094.

[Beatty and Pauly 2005] Beatty, Philip J., D. G. N. and Pauly, J. M. (2005). Rapid gridding reconstruction with a minimal oversampling ratio. *Medical Imaging, IEEE Transactions on 24.6*, 799-808.

[Braam and Wortmann 2016] Braam, P. and Wortmann, P. (2016). Kernel Prototyping SOW. Technical Report SKA-TEL-SDP-0000083, Science Data Processor Consortium.

[Cornwell 2004] Cornwell, T. J., G. K. . B. S. (2004). W projection: A new algorithm for wide field imaging with radio synthesis arrays. *Astronomical Data Analysis Software and Systems XIV ASP Conference Series*, v. 347.

[Cornwell 2006] Cornwell, T. (2006). Computing costs of imaging for the xNTD. Technical Report ASKAP Memo Series 001. *ANTF*.

[Dewdney and et. al 2010] Dewdney, P. E. and et. al (2010). SKA phase 1: Preliminary system description. SKA memo. 130.

[Dewdney et al. 2009] Dewdney, P. E., Hall, P. J., Schilizzi, R. T., and Lazio, T. J. L. W. (2009). The square kilometre array. *Proceedings of the IEEE*, 97(8):1482–1496.

[Fang et al. 2013] Fang, J., Varbanescu, A. L., Sips, H., Zhang, L., Che, Y., and Xu, C. (2013). An empirical study of Intel Xeon Phi. *arXiv preprint arXiv:1310.5842*.

[Frigo and Johnson 2005] Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93: 216–231.

[Huang et al. 2008] Huang, Q., Huang, Z., Werstein, P., and Purvis, M. K. (2008). GPU as a general purpose computing resource. In *PDCAT*.

[Humphreys and Cornwell 2011] Humphreys, B. and Cornwell, T. (2011). Analysis of convolutional resampling algorithm performance. *Memo*, 132.

[Humphreys et al. 2013] Humphreys et al., B. (2013). SKA GitHub. `https://github.com/ATNF/askap-benchmarks`. Acesso em 17/08/2017.

[Jacobs 2005] Jacobs, D. (2005). Correlation and convolution. *Class Notes for CMSC 426*.

[Muscat 2014] Muscat, D. (2014). High-performance image synthesis for radio interferometry.

[O'Sullivan 1985] O'Sullivan, J. D. (1985). A fast sinc function gridding algorithm for fourier inversion in computer tomography. *IEEE Trans. Med. Imag.*, vol. 4 pp. 200-207.

[SDP 2018] SDP (2018). SKA Website Science Data Processor. `https://www.skatelescope.org/sdp/`. Acesso em 23/03/2018.

[Sørensen et al. 2008] Sørensen, T. S., Schaeffter, T., Noe, K. Ø., and Hansen, M. S. (2008). Accelerating the nonequispaced fast fourier transform on commodity graphics hardware. *IEEE Transactions on Medical Imaging*, 27(4):538–547.

[T.J. Cornwell 2012] T.J. Cornwell, M.A. Voronkov, B. H. (2012). Correlation and convolution. *Wide field imaging for the Square Kilometre Array*.