UNIVERSIDADE ESTADUAL DE FEIRA DE SANTANA – UEFS
DEPARTAMENTO DE TECNOLOGIA
COLEGIADO DE ENGENHARIA DE COMPUTAÇÃO

JOÃO DE MATOS PEREIRA DE SOUZA NETO

PARALELIZAÇÃO DO ALGORITMO DE KRIGAGEM PARA A ARQUITETURA CUDA

FEIRA DE SANTANA
2012

JOÃO DE MATOS PEREIRA DE SOUZA NETO

PARALELIZAÇÃO DO ALGORITMO DE KRIGAGEM PARA A ARQUITETURA CUDA

Trabalho de Conclusão de Curso apresentado ao Departamento de Tecnologia, Curso de Engenharia de Computação da Universidade Estadual de Feira de Santana, como requisito parcial para obtenção do título de Engenheiro de Computação.

Orientador: Prof. Dr. Ângelo Amâncio Duarte

FEIRA DE SANTANA
2012

FOLHA DE JULGAMENTO

JOÃO DE MATOS PEREIRA DE SOUZA NETO

Monografia apresentada e julgada 11/01/2013 perante a banca examinadora:

____

Dr. Anfranserai Morais Dias

UNIVERSIDADE ESTADUAL DE FEIRA DE SANTANA

____

Dr. Ângelo Amâncio Duarte

UNIVERSIDADE ESTADUAL DE FEIRA DE SANTANA

____

Dr. Wagner Luiz Alves de Oliveira

UNIVERSIDADE ESTADUAL DE FEIRA DE SANTANA

AGRADECIMENTOS

Após chegar ao fim deste trabalho percebi que tenho muito a agradecer por ter chegado aqui. Primeiramente tenho que agradecer aos meus pais, Sandra e João, que sempre colocaram a boa educação como prioridade, mesmo nos momentos mais difíceis.

Agradeço também aos meus familiares e agregados, que sempre me deram apoio e compartilharam conquistas, em especial a meu tio Emanuel que esteve à disposição em tempo integral, tendo fundamental importância no meu aprendizado do inglês, sem o qual não poderia ter lido 80% das referências e minha tia Tânea e meus primos, com quem morei em Feira de Santana enquanto estudava.

Antes disso, dois amigos de infância tiveram grande importância nos rumos que me fizeram chegar aqui e, infelizmente não poderão acompanhar mais essa conquista: José Roque (in memoriam) o "amigo abastado" que me apresentou os computadores antes que qualquer outra pessoa da cidade tivesse um, me fazendo criar interesse por eles logo cedo e Marcos Souza (in memoriam) com quem estudava para o vestibular.

Na universidade, muitos contribuíram para a minha formação, portanto devo agradecimentos aos professores de Ecomp, em especial ao meu orientador, prof. Ângelo, cujas aulas me fizeram despertar interesse por programação paralela, e também a muitos colegas, em especial aos petianos por todos os problemas que passamos e superamos juntos e a Mateus e Lidiany minhas principais companhias e referências durante a graduação.

Outros que também contribuíram foram os amigos e conhecidos, em especial os que me cobravam insistentemente, chegando ao ponto de substituir saudações por "E aí, já está pronta? Falta muito?".

E, para não deixar ninguém de fora, agradeço também aos que me atrapalharam nesse período, pois todos eles foram bem intencionados, querendo apenas que me divertisse um pouco. De fato, na dúvida, acabo agradecendo por terem me ajudado a manter a sanidade nesse período, em especial, a Eduardo, Paulo Roberto, Gustavo, Tácio e a galera da EBADS (Escola Baiana de Dança de Salão) com os quais "desperdicei" uma parte do tempo que poderia ter me dedicado ao trabalho.

Por fim, novamente, agradeço a Mateus pela minuciosa revisão de última hora, e a Fernanda pela disponibilidade em tirar minhas dúvidas sobre a implementação que a mesma fez do método de krigagem.

“9 mães conseguem gerar um bebê em um mês”

Desciclopédia sobre programação paralela.

RESUMO

Esta monografia descreve a paralelização do algoritmo de interpolação por krigagem ordinária, desenvolvido em trabalho anterior, para a arquitetura de programação paralela da Nvidia – CUDA. O código original, orientado a objetos, foi reescrito no paradigma procedural, e o tempo de cada etapa de sua execução foi medido, possibilitando a escolha do que seria paralelizado. Escolheu-se a última etapa do algoritmo, pois a mesma tinha duração muito maior que as demais para grandes quantidades de pontos a serem estimados. A paralelização do código se deu explorando-se a hierarquia de memória diferenciada dos dispositivos da arquitetura, e também seu poder de processamento paralelo, respeitando a quantidade de memória disponível nos dispositivos, e ainda deixando o uso desses recursos configuráveis através de constantes no código. Após a codificação, foram feitos testes para verificar a exatidão do algoritmo e, em seguida, foram realizados testes para descobrir a melhor configuração do programa em cada um dos dispositivos disponíveis, para que o desempenho entre as placas pudesse ser comparado com a melhor configuração possível para desempenho. O programa foi bem sucedido nos testes, trazendo resultados precisos muito mais rapidamente que o programa serial e se escalando bem quando se aumenta a quantidade de hardware paralelo.

Palavras-chave: CUDA. Programação Paralela. Krigagem.

ABSTRACT

This text describes the parallelization of ordinary kriging algorithm for points interpolation, developed in a previous work, using the Nvidia parallel programming architecture – CUDA. The original object-oriented code was rewritten using the procedural paradigm, where its execution time was spitted into steps, making possible to choose part should be parallelized. The code parallelization took place using the device's memory hierarchy from the architecture and its parallel computing power, respecting its amount of memory. The use of resources in such a code is configurable by constants. After coding, the application was tested to verify the its accuracy, and to determine its best configuration for performance in three different Nvidia GPU boards. The results shown that accuracy was acceptable and that the execution time of the parallel version performed very well in terms of speedup and efficiency.

keywords: CUDA. Parallel Computing. Kriging

LISTA DE FIGURAS

Figura 1: Determinação do eixo X do semivariograma teórico..18

Figura 2: Comparação da distribuição do uso do chip entre as arquiteturas............................24

Figura 3: Visão geral sobre modelo de memória de dispositivo CUDA (KIRK;HWU, 2010) 26

Figura 4: Representação de dois multiprocessadores de fluxo...27

Figura 5: Chamada de função informando as dimensões do grid e dos blocos........................30

Figura 6: Exemplo de função do tipo kernel que soma dois vetores de forma paralela...........30

Figura 7: Legenda de cores...32

Figura 8: Trecho de arquivo de entrada do SAPoCaPT..33

Figura 9: Representação da matriz de distâncias..33

Figura 10: Distâncias divididas em Histogram_nbits intervalos...34

Figura 11: Representação do agrupamento dos pontos de acordo com as distâncias entre eles. 34

Figura 12: Determinação da grade de pontos a serem estimados, a partir dos pontos amostrados...35

Figura 13: Representação do laço para determinação dos pontos por krigagem, quando a matriz inversa já está calculada..36

Figura 14: Configurações do a) grid e b) blocos para o preenchimento dos eixos..................41

Figura 15: Preenchimento de um Eixo...42

Figura 16: Cálculo da coluna de covariâncias: a) versão serial, repetida INTERVALS² vezes, b) versão paralela executada apenas uma vez..44

Figura 17: Configurações a) do Grid e b) dos blocos..44

Figura 18: Dados necessários da memória global para a) calcular uma coluna de covariâncias e b) calcular quatro colunas...45

Figura 19: Representação de um bloco na fase inicial do cálculo das covariâncias, onde os threads obtêm as coordenadas dos pontos..46

Figura 20: Representação do término da primeira e início da segunda execução do laço que calcula as covariâncias..47

Figura 21: Substituição dos dados na memória compartilhada...48

Figura 22: Um bloco genérico na última iteração: a quantidade de pontos pode ser menor que nas anteriores...49

Figura 23: Início da execução do último bloco: ele pode ter menos colunas para calcular que os demais blocos....49
Figura 24: Combinação dos dois problemas apresentados nas figuras anteriores....50
Figura 25: Determinação dos pesos de forma serial....51
Figura 26: Determinação dos pesos de forma paralela....51
Figura 27: a) Grade e b) blocos configurados para o cálculo dos pesos....52
Figura 28: Acesso à memória para determinação de um ponto da matriz produto....53
Figura 29: Padrão de acessos repetidos à memória....54
Figura 30: Carga colaborativa dos primeiros elementos das matrizes....55
Figura 31: Segunda etapa da carga colaborativa da memória compartilhada....55
Figura 32: a) preenchimento da memória compartilhada quando não há elementos suficientes para o preenchimento total e b) finalização do cálculo dos pesos para o bloco em questão....56
Figura 33: Casos em que os blocos podem ter menos pesos para calcular que threads....57
Figura 34: Cálculos dos pontos, usando-se os valores amostrados e os pesos previamente calculados....58
Figura 35: a) Grade e b) blocos configurados para o cálculo dos pesos....59
Figura 36: Disposição dos blocos no grid para cálculo dos pontos....59

LISTA DE TABELAS

Tabela 1: Varáveis e constantes que configuram a krigagem serial..32
Tabela 2: Tempo de execução de cada etapa (s), para interpolação de 25 pontos...................38
Tabela 3: Tempos de execução do algoritmo serial (s), aumentando-se a quantidade de pontos interpolados..39
Tabela 4: Varáveis e constantes que configuram o sistema..40
Tabela 5: Dispositivos disponíveis para testar o programa...61
Tabela 6: Pontos estimados por diferentes versões do programa. 62
Tabela 7: Resposta no uso da memória e na variação do tempo de execução para valores de PARALLEL_POINTS...64
Tabela 8: Configuração do programa para cada uma das placas...68
Tabela 9: Tempo da etapa de estimação, variando-se apenas o número de amostras..............69
Tabela 10: Tempos de execução do programa, versões paralela e serial para 700 amostras....70
Tabela 11: Aumento relativo de núcleos e aceleração..73
Tabela 12: Eficiência da etapa paralela...73

LISTA DE GRÁFICOS

Gráfico 1: Semivariograma experimental e os parâmetros necessários ao algoritmo..............17
Gráfico 2: Modelo de semivariograma gerado com parâmetros obtidos do Gráfico 1.............17
Gráfico 3: Tempo de execução de cada etapa, para interpolação de 25 pontos com o algoritmo serial..38
Gráfico 4: Tempo de execução de cada etapa, para um número crescente de pontos interpolados..39
Gráfico 5: Alocação da memória compartilhada, de acordo com o número de threads...........65
Gráfico 6: Impacto do tamanho do bloco no tempo de execução na GeForce GT 240............66
Gráfico 7: Impacto do tamanho do bloco no tempo de execução na Tesla c2070....................67
Gráfico 8: Tempo de execução do programa completo variando-se apenas o número de amostras...70
Gráfico 9: Tempo de execução do programa em placas de vídeo diferentes............................72
Gráfico 10: Eficiência do programa como um todo...74

SUMÁRIO

1 INTRODUÇÃO....13
2 REVISÃO BIBLIOGRÁFICA....15
2.1 Krigagem 15
2.1.1 Krigagem Ordinária – Uma Abordagem Top-Down....15
2.2 Programação Paralela....18
2.2.1 Peculiaridades da Programação Paralela....19
2.2.2 Paralelismo em CPU....20
2.3 Programação Paralela em GPU 22
2.3.1 Arquiteturas de GPGPUs Modernas....23
2.3.2 Arquiteturas de GPGPU Recentes da NVIDIA....24
2.3.2.1 Hierarquia de Memória....25
2.3.2.2 Multiprocessadores de Fluxo....26
2.3.2.3 Escalonamento de Threads 27
2.3.2.4 Precisão da arquitetura CUDA....28
2.3.3 CUDA C....29
3 MATERIAIS E MÉTODOS....31
3.1 Algoritmo Serial de Interpolação por Krigagem....31
3.1.1 Implementação do Algoritmo de Krigagem....32
3.2 Estudo dos Tempos de Execução....37
3.3 Paralelização da Interpolação....40
3.3.1 Determinação da Grade de Pontos....40
3.3.2 O Cálculo das Colunas de Covariâncias....43
3.3.3 Determinação dos Coeficientes....50
3.3.4 A estimativa dos Pontos....57
3.4 Método Usado nos Testes....60
4 EXPERIMENTOS E RESULTADOS....61
4.1 Ambiente de Teste....61
4.2 Precisão e Exatidão....61
4.3 Configuração do Programa....63
4.3.1 Quantidade de Pontos Calculados por Iteração....63
4.3.2 Quantidade de Threads por Blocos....64
4.4 Desempenho....68

4.4.1 Comparação Entre a Versão Paralela e a Serial.....69
4.4.2 Eficiência no Uso do Hardware Paralelo.....71
5 CONCLUSÃO.....75
REFERÊNCIAS.....77

1 INTRODUÇÃO

Vem sendo desenvolvido na Universidade Estadual de Feira de Santana um Sistema de Auxílio ao Posicionamento de Poços em Campos de Petróleo Terrestre (SAPoCaPT). O SAPoCaPT permite que arquivos sejam lidos para obtenção dos pontos amostrados, possibilitando a geração dos mapas de propriedades, através de interpolação por médias móveis (CASTELO BRANCO, 2010). As propriedades utilizadas são permeabilidade, espessura porosa com óleo, pressão estática e BSW (*Basic Sediments and Water* – sedimentos básicos e água). Após o carregamento dessas quatro propriedades, pode-se gerar um mapa de qualidade, que será usado pelo usuário do programa para determinar o melhor local para perfuração.

Durante o desenvolvimento do SAPoCaPT observou-se que o método das médias móveis não apresentava o resultados suficientemente precisos e decidiu-se implementar o algoritmo de interpolação por krigagem, por ser um importante método geoestatístico amplamente usado em mineração, hidrogeologia, ciência ambiental dentre outros (CHENG et al, 2010).

O algoritmo de krigagem se mostrou mais preciso (CASTELO BRANCO, 2010), porém por ter maior complexidade computacional aumentou o tempo de execução do sistema. Esse cenário motivou a paralelização do algoritmo para aumento de desempenho, o que tem se mostrado uma tendência, já que Cheng (2010) e Guvendik et al (2012) também fazem uso de paralelização para ganhar desempenho na krigagem.

Uma tecnologia para paralelização recente é a arquitetura CUDA (*Compute Unified Device Architecture)* (SANDERS;KANDROT, 2010) da Nvidia, que permite aumentar o desempenho de determinadas aplicações usando recursos de GPU (*Graphics Processing Unit* - Unidade de Processamento Gráfico) de placas gráficas disponíveis no mercado. Esta abordagem, permite, de a cordo com o tipo de aplicação, alcançar ganho de desempenho a um custo significativamente menor que outras arquiteturas de programação paralela, pois o custo de uma placa gráfica é geralmente muito inferior ao de um cluster de computadores.

Isso posto, esse trabalho de conclusão de curso teve como objetivo principal paralelizar para a arquitetura CUDA, o algoritmo de krigagem originalmente desenvolvido

para o SAPoCaPT e, a partir daí, avaliar o desempenho e exatidão dos resultados do código desenvolvido, além de sua eficiência na exploração do hardware paralelo disponível, com o intuito de avaliar se o algoritmo se adequava à arquitetura CUDA. Os resultados obtidos mostraram que o programa paralelo desenvolvido executa muito mais rapidamente que o serial e que a porção paralelizada do código apresenta uma aceleração próxima do ideal teórico, quando há grande quantidade de cômputo.

2 REVISÃO BIBLIOGRÁFICA

Esta seção trata do conhecimento teórico necessário para compreensão da metodologia e resultados do trabalho. Primeiramente apresenta-se o método da krigagem e, em seguida, é feita uma breve contextualização do universo da programação paralela e, por fim, há um aprofundamento com foco em programação para GPU com arquitetura CUDA, utilizando-se CUDA C.

2.1 Krigagem

A krigagem é um método de interpolação geoestatístico usado para estimar pontos não amostrados no espaço, partindo do princípio que pontos mais próximos têm valores similares. Trata-se de um estimador linear, pois os dados estimados são combinações lineares dos amostrados, que faz a média dos erros ser aproximadamente nula; e apresenta uma variância mínima. Essas características fazem desse método o melhor estimador linear não viciado (GUERRA, 1988).

A estimação por krigagem, bem como seus tipos, é explicada em Guerra (1988) e Camargo et al (2012). Castelo Branco (2010) escolhe a Krigagem Ordinária para implementar no SaPoCaPT, apresentando os detalhes relativos à implementação do mesmo, que serão contextualizados nesta seção.

2.1.1 Krigagem Ordinária – Uma Abordagem *Top-Down*

Como já mencionado, a krigagem é um estimador linear, portanto a determinação do valor em um ponto não amostrado se dá através da multiplicação de cada valor amostrado por um peso previamente calculado. A equação 1 descreve como se calcula o ponto não amostrado de coordenadas ($\mathbf{x},\mathbf{y}$), em que $\mathbf{w_i}$ é o peso calculado, $\mathbf{f_i}$ o valor amostrado no ponto $\mathbf{i}$, e $\mathbf{n}$ o número de pontos amostrados.

$$F(x,y)=\sum_{i=1}^{n}(w_i f_i) \quad (1)$$

Pela equação 1, nota-se que todos os valores amostrados contribuirão com o ponto estimado. Portanto, para que valores diferentes possam ser estimados, esses pesos precisam ser recalculados a cada estimativa.

Essa etapa de cálculo dos pesos é basicamente o que diferencia os tipos de krigagem. No caso da krigagem ordinária, esse processo é feito resolvendo-se o sistema de equações 2, sendo que **C(h$_{ij}$)** é o valor da covariância de **h$_{ij}$** – em que **h$_{ij}$** é a distância do ponto **i** ao ponto **j** – , **X** uma variável temporária e **p** o ponto para o qual se quer estimar um valor. A resolução desse sistema de equações é a parte mais custosa do algoritmo de krigagem, já que o mesmo deve ser resolvido para cada ponto a ser estimado.

$$\begin{cases} w_1 C(h_{11})+w_2 C(h_{12})+...+w_n C(h_{1n})+X=C(h_{1p}) \\ w_1 C(h_{21})+w_2 C(h_{22})+...+w_n C(h_{2n})+X=C(h_{2p}) \\ ... \\ w_1 C(h_{n1})+w_2 C(h_{n2})+...+w_n C(h_{nn})+X=C(h_{np}) \\ w_1+w_2+...+w_n=1 \end{cases} \quad (2)$$

Na krigagem, é natural que as distâncias entre os pontos tenha valor crucial na determinação dos pesos, pois se trata de um método geoestatístico. Por isso, o sistema de equações se vale do conceito de covariância, que mede a interdependência espacial entre duas variáveis – no caso, dois pontos no plano. As covariâncias são calculadas pela equação 3,

$$C(d_{ij})=\begin{cases} 0 & , se\, d_{ij}=0 \\ C_0+C_1[Sph(h_{ij})] & , se\, 0<a<d_{ij} \\ C_0+C_1 & , se\, d_{ij}>a \end{cases} \quad (3)$$

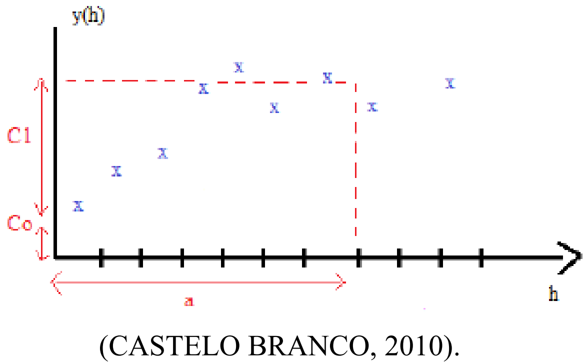
na qual **Sph(h$_{ij}$)** é um modelo matemático de um semivariograma (função da covariância entre pontos em relação a distância), **C$_0$**, **C$_1$** e **a** são parâmetros obtidos através do semivariograma experimental – Gráfico 1. O modelo utilizado na implementação de **Sph(h$_{ij}$)** é o esférico, que pode ser observado equação 4.

$$Sph(h_{ij})=1,5(\frac{h_{ij}}{a})-0,5(\frac{h_{ij}^3}{a}) \quad (4)$$

O semivariograma experimental, assim como o teórico, é um gráfico das semi-variâncias em função das distâncias, porém ele é discreto e tem seus valores obtidos através das amostras. Tem a configuração mostrada no Gráfico 1 e tem como utilidade auxiliar o especialista usuário do programa a escolher os parâmetros para o cálculo do semivariograma

teórico. Este deve apresentar uma curva com as mesas características semivariograma esperimental do experimental, como ilustrado no Gráfico 2 porém apresentando continuidade. Esses parâmetros estão representados no Gráfico 2 – $\mathbf{C_0}$, $\mathbf{C_1}$ e **a** – e são os mesmos utilizados para a determinação das equações 3 e 4.

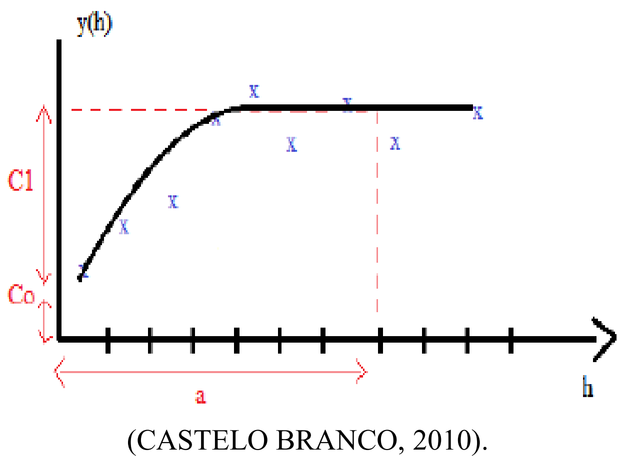
Gráfico 1: Semivariograma experimental e os parâmetros necessários ao algoritmo



(CASTELO BRANCO, 2010).

Usando-se o modelo esférico, espera-se obter uma curva similar ao Gráfico 2. Vale ressaltar que a escolha desses parâmetros tem influência direta no resultado das aproximações, porém não interfere no algoritmo, nem na quantidade de cômputo, apenas no resultado dos pontos estimados.

Gráfico 2: Modelo de semivariograma gerado com parâmetros obtidos do Gráfico 1



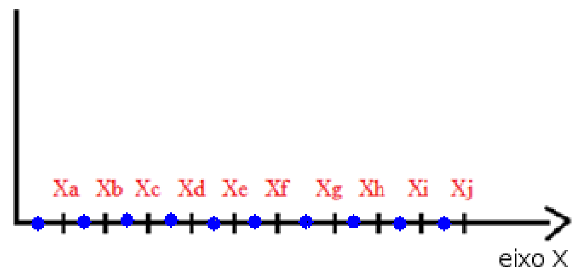
(CASTELO BRANCO, 2010).

Os pontos do semivariograma experimental (ex. Gráfico 1) são determinados da seguinte forma: primeiro calculam-se as distâncias de cada ponto em relação a todos os demais, usando a fórmula da distância entre dois pontos, expressa na equação 5.

$$h=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2} \tag{5}$$

Após obtidas todas as distâncias, já é possível se determinar o eixo das abscissas do semivariograma experimental, através da divisão em intervalos iguais, de zero a maior distância entre os pontos. Essa quantidade de divisões deve ser especificada pelo usuário. O ponto médio entre esses intervalos determina as abcissas. Na Figura 1 é possível observar a representação da divisão em intervalos, com as pequenas barras verticais no eixo, bem como o ponto médio dos intervalos, representados pelos pontos.

Figura 1: Determinação do eixo X do semivariograma teórico.



O eixo Y do semivariograma teórico é calculado pela equação 6, equação do semivariograma. Antes de usá-la, porém, é necessário agrupar as distâncias de acordo com os intervalos determinados no passo anterior. Assim, para cada Y calculado serão usados apenas os pontos daquele intervalo específico.

$$y(h)=\frac{1}{2\mathrm{n}(h)}\sum_{i=1}^{n}[z(x_i)-z(x_i+h)] \tag{6}$$

Na equação, y(h) é o ponto calculado, h é o ponto do eixo x (a distância), z(x) é o valor amostrado para aquele ponto, e z(x + h) é o valor amostrado do ponto imediatamente mais próximo a ele, no conjunto de pontos daquele intervalo.

Esta seção resumiu o necessário para o entendimento da implementação do algoritmo da krigagem presente no SAPoCaPT. Para aprofundamento de detalhes, tanto de implementação como teóricos ver Castelo Branco (2010).

2.2 Programação Paralela

Durante três décadas, o aumento de desempenho dos sistemas computacionais se deu através da melhoria no processador (aumento da frequência *clock*, melhoria na arquitetura e no acesso à memória), porém desde 2002 os fatores físicos para utilização dessas técnicas, como alto consumo de energia e superaquecimento, acabaram por limitar bastante essas melhorias (PACHECO, 2011). Uma alternativa para aumentar o desempenho é utilizar programação paralela, na qual o computo é dividido entre várias unidades de processamento para se obter o resultado mais rápido.

Programar para arquiteturas paralelas é mais complexo que para arquiteturas tradicionais, pois naquelas o programador tem que se preocupar com comunicação e sincronização entre diferentes tarefas, além do próprio trabalho de particionar o problema em tarefas.

A programação paralela está intimamente ligada ao hardware para o qual se quer programar, que pode ser, por exemplo, um computador com processador de múltiplos núcleos, uma grade de computadores, um *cluster*, uma GPGPU (*General Purpose GPU* – GPU de propósito geral) ou até mesmo uma combinação destes.

2.2.1 Peculiaridades da Programação Paralela

A maioria dos algoritmos tem etapas que são estritamente sequenciais, por isso a paralelização dos mesmos não consegue alcançar um nível linear de ganho de desempenho quando se aumenta o número de unidades de processamento. A lei de Amdahl (AMDAHL, 1967) enuncia que o aumento de velocidade de execução de um programa paralelo se dá de acordo com a equação 7,

$$S=\frac{1}{(1-P)} \tag{7}$$

em que P é a parte paralelizável do algoritmo e S, a aceleração (*speedup*).

Outra forma de se avaliar o código paralelo é quanto à eficiência do mesmo em explorar os recursos paralelos disponíveis, a qual é a medida relativa entre a aceleração e o

número de núcleos, sendo seu valor ideal 1. Um gráfico que plota a eficiência do programa em função de um número crescente de núcleos de processamento disponíveis mostra a escalabilidade do programa, ou seja, o quanto o programa responde bem ao aumento do número de unidades de processamento.

Um programa paralelo é dividido em tarefas, que são distribuídas entre os processadores. Porém, para que o resultado seja coerente, essas tarefas precisam estar devidamente sincronizadas.

As tarefas podem apresentar interdependência. Neste caso, o programador deve se preocupar em garantir que uma tarefa **B**, que dependa do resultado de outra tarefa **A**, só comece a executar após o término de **A**. Além disso, tarefas que executam em paralelo realizam computo em subconjuntos diferentes dos dados. É também responsabilidade do programador delimitar a fatia do dado que cada tarefa vai processar. (PACHECO, 2011)

2.2.2 Paralelismo em CPU

O modelo básico de arquitetura de computadores proposto por Von Neumann em 1945, totalmente serial, definiu que uma unidade de processamento buscasse instruções e dados na memória principal, com a comunicação entre CPU e memória ocorrendo através de um barramento de dados. Desde então, várias melhorias foram adicionadas à arquitetura, como a memória cache, e diversas maneiras diferentes de se incluir paralelismo em vários níveis.

Primeiramente surgiu o paralelismo em nível de instrução, com a introdução de processadores com suporte a pipeline. No pipeline, cada instrução é quebrada em etapas, executadas por unidades funcionais diferentes e, à medida que a primeira etapa de uma instrução termina de ser executada, passa-se para a segunda, ao passo que a instrução seguinte executaria a sua primeira etapa. Dessa forma várias são executadas em paralelo, cada uma em uma etapa diferente, em condições ideais.

Ainda no nível de instrução, surgiram os processadores superescalares, que aumentam a vazão de instruções permitindo a execução de mais de uma instrução ao mesmo tempo. Isso é possível graças a um número extra de unidades funcionais que permite executar mais de uma instrução por *clock*. Porém, como os programas são escritos para ser sequenciais, é necessário fazer a verificação de dependência entre as instruções, pois se uma depender do resultado da outra elas não poderão ser executadas em paralelo. Essa verificação pode ser executada pelo processador em tempo de execução ou pelo compilador, dependendo da abordagem.

Essas técnicas de paralelismo em nível de instrução são interessantes pois aumentam a vazão de instruções sem necessidade de alteração do código, porém pelo fato desse fluxo de instruções ser sequencial, essas técnicas acabam sendo limitadas e complexas de implementar. Isso criou a necessidade de se buscar outras técnicas de aumento de desempenho, que incluíam sistemas com mais de um processador.

Nesse contexto, Flynn (1972) propõe o agrupamento das arquiteturas de computadores em quatro classes que são: SISD (*Single Instruction, Single Data stream* – fluxo de instrução único, fluxo de dados único), SIMD (*Single Instruction, Multiple Data stream* – fluxo de instrução único, vários fluxos de dados), MISD (*Multiple Instruction, Single Data stream* – vários fluxos de instruções, fluxo de dados único) e, MIMD (*Multiple Instruction, Multiple Data stream* – vários fluxos de instruções, vários fluxos de dados).

A arquitetura SISD caracteriza computadores seriais, em que a um dado momento, existe um processo utilizando apenas um fluxo de dados.

Por sua vez, na arquitetura SIMD o mesmo conjunto de instruções é aplicado a vários fluxos de dados. Para atingir essa premissa, são necessárias várias unidades de processamento que devem operar o mesmo conjunto de instruções em paralelo. Essa arquitetura é particularmente útil quando há uma grande quantidade de dados para os quais a mesma operação deve ser executada, aumentando-se a vazão de dados.

Na MISD, um fluxo de dados é executado paralelamente por fluxos diferentes de instruções, para, por exemplo, aumentar a tolerância a falha, enquanto na MIMD, vários processadores funcionam de forma independente, dessa forma, a um dado momento esses processadores podem estar executando programas diferentes em dados distintos.

Nas arquiteturas com múltiplos processadores, a memória pode ser compartilhada, podendo ser acessada por todos com apenas um espaço de endereçamento, ou distribuída, em

que cada unidade de processamento tem a sua memória, com seu endereçamento, e a comunicação entre eles acontece através de, por exemplo, uma rede. A memória pode ainda ser distribuída compartilhada, onde o esquema de memória distribuída pode ser acessado através de um mesmo espaço de endereçamento, ainda que elas estejam fisicamente separadas.

Essas arquiteturas e esses modelos de memória resumem as possibilidades em programação paralela. Tratando-se de GPU, não há grandes diferenças nos modelos básicos de arquitetura, como ficará mais claro na próxima seção.

2.3 Programação Paralela em GPU

Com o surgimento de sistemas operacionais gráficos no final dos anos 80 e início dos 90 abriu-se oportunidade para o mercado de placas gráficas, que começou com as aceleradoras 2D, oferecendo suporte a operações de *bitmap* em hardware. Em seguida foram lançados os primeiros jogos de tiro em primeira pessoa, que tornaram as placas aceleradoras 3D mais populares.

Os jogos foram evoluindo, se popularizando e, consequentemente, criando um mercado bastante lucrativo para fabricantes de placas gráficas, que se propunham a oferecer os recursos de processamento necessários aos jogos, fazendo assim, hardwares cada vez mais velozes. A evolução dessas placas chegou a tal ponto que começou a chamar a atenção de pesquisadores interessados em aumentar a velocidade de execução de suas aplicações, porém não havia linguagens de propósito geral para as mesmas.

Nesse contexto, para aproveitar os recursos computacionais de uma placa de vídeo, era necessário desenvolver os programas usando as API (*Application Programming Interface* – Interface de Programação de Aplicativos) gráficas, convertendo o problema em operações gráficas nativas das API. Esse processo era mais trabalhoso que simplesmente usar uma linguagem de propósito geral, porém o esforço era feito pelo programador pois o desempenho de aplicações que podem explorar o processamento vetorial das placas gráficas era significativamente maior. Atualmente já se pode contar com linguagens de programação de

propósito geral para GPU, como a CUDA C (KIRK;HWU, 2010) ou OpenCL (Tsuchiyama et al, 2012).

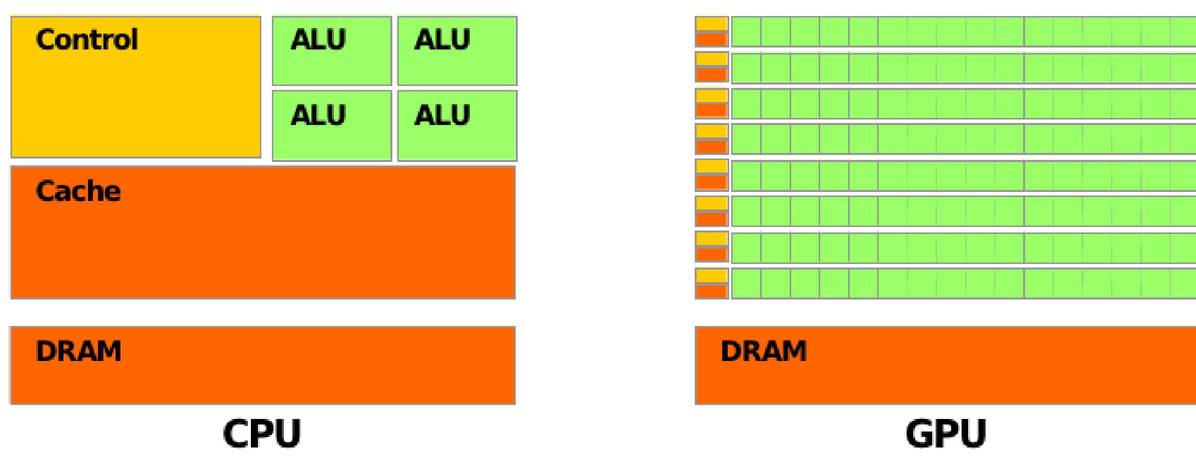
A *framework* OpenCL é um padrão aberto que viabiliza a programação paralela em sistemas heterogêneos, como as placas gráficas recentes da AMD, Nvidia, Intel e alguns dispositivos embarcados. A linguagem CUDA C tem o mesmo propósito da OpenCL, porém é exclusiva da Nvidia, e compatível apenas em placas da mesma. Ambas definem uma maneira de programar dispositivos escrevendo métodos específicos para os mesmos, além de oferecer API, que incluem funções para gerenciar cópias de dados de e para os dispositivos, além de estruturas que permitem criar e sincronizar as tarefas executadas paralelamente nesses dispositivos.

2.3.1 Arquiteturas de GPGPUs Modernas

As CPUs (*Central Processing Unit* – Unidade Central de Processamento) modernas são projetadas para executar códigos sequenciais o mais rápido possível, fazendo uso de complexas unidades de controle, que permitem execução em paralelo de instruções de um único fluxo ou até mesmo fora de ordem, trazendo o mesmo resultado da execução sequencial. Os processadores também fazem mais cache de memória para diminuir a latência da memória principal. Isso torna a execução de programas mais rápida, porém não aumenta a quantidade máxima de operações por segundo – o cômputo. Outra desvantagem das CPUs é precisar manter compatibilidade com sistemas operacionais, aplicativos legados, o que dificulta bastante o aumento da largura de banda da memória (KIRK;HWU, 2010).

As GPUs, por sua vez, não herdam tantas limitações e usam modelos de memória mais simples, por isso mantêm uma largura de banda consideravelmente maior entre sua memória e suas unidades de processamento (KIRK;HWU, 2010). Ao contrário das CPUs, que fazem uso massivo de cache, as GPUs usam grande quantidade de ULA (Unidade Lógica e Aritmética), visando uma vazão maior de cálculos, que operam paralelamente, superando bastante a capacidade de cômputo das CPUs. A Figura 2 mostra dois diagramas de blocos que comparam a distribuição do uso de ULA (ALU, na figura), unidades de controles e memória cache entre uma CPU moderna e uma GPU moderna.

Figura 2: Comparação da distribuição do uso do chip entre as arquiteturas.



(NVIDIA,2012)

O processamento nas GPGPU se dá através dos SP (*Stream Processors* – Processadores de Fluxo), que usam o paradigma de processamento de *stream*, que está relacionado à arquitetura SIMD, permitindo a alguns tipos de aplicações o uso do paralelismo que pode ser oferecido pelas placas gráficas. Esse paradigma funciona da seguinte forma: a um dado conjunto de dados (fluxo) é aplicada uma série de operações (*kernel funtions*). Como as operações são as mesmas para todo o conjunto de dados, elas são divididas entre os vários SP disponíveis na GPGPU.

As tarefas que são divididas entre os SP são comumente agrupadas em *grids* bidimensionais, pois esse esquema se encaixa perfeitamente com os modelos de renderização das GPU. Cada tarefa criada tem uma coordenada no *grid* e, portanto, como elas têm que executar o mesmo cômputo numa porção diferente do dado, essas posições são utilizadas para mapear esses dados.

2.3.2 Arquiteturas de GPGPU Recentes da NVIDIA

A arquitetura CUDA foi a primeira a introduzir o conceito de computação de propósito geral e, embora tenha a desvantagem de ser compatível apenas com placas da Nvidia, está continuamente lançando novas versões, sendo a mais atual a 5.0. Uma comparação entre CUDA e OpenCL é feita por Kamran et al (2011), que chega à conclusão da superioridade de desempenho da primeira em relação à segunda. Por essas razões, a mesma foi escolhida e será detalhada nesta seção.

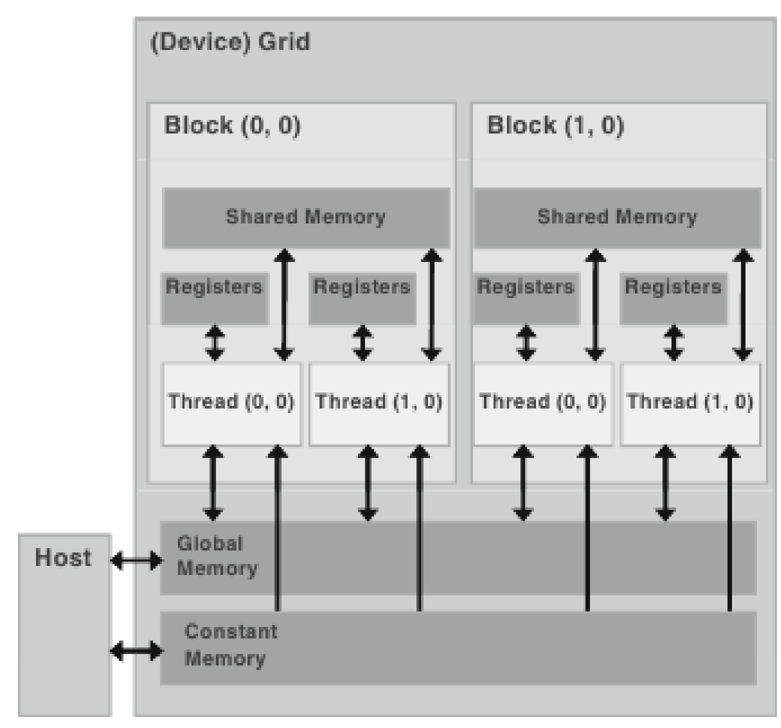
2.3.2.1 Hierarquia de Memória

As placas gráficas da Nvidia têm sua própria memória, independente da memória principal do computador, e com uma hierarquia diferenciada, visando suprir necessidades provenientes de execução de código paralelo.

Como pode ser observado na Figura 3, existem a Memória Global (*Global Memory*), a Memória Constante (*Constant Memory*), a Memória Compartilhada (*Shared Memory*) e os Registradores (*Registers*). As duas primeiras memórias se comunicam diretamente com a memória principal do *host*, de onde os dados são copiados e para onde o resultado do cômputo é retornado. A memória constante é somente de leitura e limitada a 64kB, enquanto a global permite escrita e tem tamanho geralmente superior a 512MB, dependendo do modelo do hardware. A memória global tem uma latência muito grande, da ordem de centenas de ciclos de *clock*, o que é compensado com outras especificidades da arquitetura, que serão definidas adiante.

A memória compartilhada é significativamente menor que a global, porém mais rápida, e recebe este nome por ser compartilhada por determinados grupos de *threads*, como ficará mais claro a seguir. Por fim, os registradores são as memórias menores, mais rápidas, e não são compartilhados por *threads* diferentes.

Figura 3: Visão geral sobre modelo de memória de dispositivo CUDA

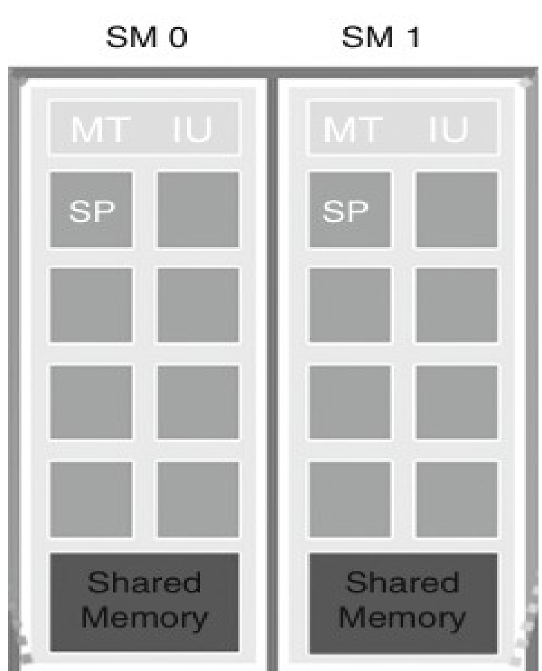


(KIRK;HWU, 2010)

2.3.2.2 Multiprocessadores de Fluxo

O processamento nas placas Nvidia se dá nos SM (S*tream Multiprocessor* – multiprocessadores de fluxo), que são conjuntos de *CUDA cores (ou SP - Stream Processor* – Processador de fluxo). Nas primeiras gerações da arquitetura, cada multiprocessador tinha oito *cores*, como o representado na Figura 4, enquanto atualmente existem placas com 32 *CUDA cores* por multiprocessador. É interessante notar que existe uma memória compartilhada para cada SM.

Figura 4: Representação de dois multiprocessadores de fluxo



(KIRK;HWU, 2010)

Cada *CUDA core* tem uma unidade de processamento de inteiros e outra de ponto flutuante, que não podem ser usadas ao mesmo tempo, e pode executar até duas operações por *clock*.

2.3.2.3 Escalonamento de *Threads*

O software escrito para a arquitetura CUDA divide suas tarefas em *threads*, agrupados em blocos que formam um *grid*. Blocos de *threads* são atribuídos a SM, sendo que todos os *threads* de um bloco compartilham a memória compartilhada do SM. Os *threads* de um mesmo bloco são escalonados, de forma que enquanto alguns estiverem esperando dados da memória global, outros estejam executando cômputo. Isso é feito de maneira transparente pelo hardware e, para que haja um bom aproveitamento do hardware, é necessário apenas que em um bloco existam mais *threads* que *CUDA cores* por SM. Porém esse número não pode ser demasiado grande, pois os recursos de memória compartilhada e registradores são divididos entre eles. Outro fator limitante é a própria arquitetura, que nas suas primeiras

versões restringia o número de *threads* por bloco a um máximo de 512, chegando atualmente a 1024.

Uma vez definido o número de *threads* por blocos e o número de blocos, o escalonamento de *threads* é feito de forma transparente pelo hardware, que tenta ao máximo manter os processadores ocupados, mudando *threads* sempre que há necessidade de se esperar algum dado da memória.

2.3.2.4 Precisão da arquitetura CUDA

A arquitetura CUDA implementa o padrão IEEE 754 para pontos flutuantes de precisão simples, e nas placas mais recentes, com arquitetura de versão igual ou superior a 2.0, também há suporte para precisão dupla.

Em 2008 o padrão foi revisado, adicionado-se suporte a FMA (Fused Multiply-Add) (WHITEHEAD; FIT-FLOREA, 2011). Com essa operação, um produto seguindo de uma soma (A*B +C) é arredondado apenas uma vez, diferindo da implementação anterior na qual havia dois arredondamentos, um após a multiplicação e outro após a soma.

Essa revisão do padrão só se refletiu na arquitetura CUDA em suas versões a partir da 2.0. Dessa forma, com o aumento de precisão em dispositivos de gerações diferentes, é esperado se obter resultados ligeiramente diferentes.

Whitehead (2011) explica e demonstra que bibliotecas matemáticas diferentes, ainda que executem no mesmo processador, podem retornar resultados desiguais, embora próximos, pois a ordem em que as operações são executadas influencia nos resultados, devido à finitude da precisão. Assim sendo, quando se programa para uma arquitetura diferente, já é esperado se obter resultados ligeiramente diferentes.

2.3.3 CUDA C

Para tornar acessível os recursos da arquitetura CUDA, a NVIDIA disponibilizou a linguagem CUDA C - de propósito geral - e o seu compilador *nvcc*. A linguagem é uma extensão do C, em que o desenvolvedor pode escrever código tanto para o processador como para a GPU, ter acesso direto à memória do(s) dispositivo(s) de vídeo e fazer a comunicação e sincronização CPU/GPU.

A linguagem CUDA C introduz algumas extensões às linguagens C/C++, dentre elas: os qualificadores de tipo de função; qualificadores de tipo de variável; sintaxe diferenciada para chamada de funções executadas na GPU; e variáveis internas para identificação de *threads* e blocos.

Os qualificadores de tipo de função são __global__, __device__ e __host__. Os dois iniciais definem funções que são executadas na GPU, porém o primeiro é para funções que são chamadas apenas pelo *host* – funções do tipo *kernel* – e o segundo apenas por código executado na GPU. Isso implica que uma função __global__ tem o código que será executado por todos os threads, enquanto as __device__, por serem chamadas por *threads*, têm o código que será executado por apenas um deles.

O qualificador __host__ é para funções executadas na CPU, é o padrão e pode ser omitido. Uma função pode ter os qualificadores __device__ e __host__ combinados, o que resulta na compilação de duas versões para o método, uma para a GPU e outra para a CPU.

Os qualificadores de tipo de variável são __device__, __constant__ e __shared__. Nos dois primeiros as variáveis têm o tempo de vida da aplicação e são visíveis por todos os *threads*, diferenciando-se apenas pelo fato do primeiro permitir escrita. Variáveis do tipo __constant__ não podem ser editadas, porém o acesso a elas é mais rápido, pois a arquitetura se aproveita dessa característica para fazer cache do seu conteúdo. Por último, variáveis do tipo __shared__ têm o escopo de um bloco e são visíveis e compartilhadas por todos os *threads* do mesmo.

Para que o programa executado na CPU consiga chamar uma função do tipo __global__, é necessário informar as dimensões que terão o *grid* e cada bloco. Isso é feito através de uma nova sintaxe para chamada de métodos, que informa as dimensões entre os delimitadores <<< e >>>, como pode ser visualizado na Figura 5, na qual dim_5 e dim_6 são variáveis com as dimensões previamente inicializadas.

Figura 5: Chamada de função informando as dimensões do grid e dos blocos.

```
702        D_media_ponderada<<<dim_5,dim_6>>>(d_estimated_points,
703                                           d_color,
704                                           number_of_lines,
705                                           INTERVALOS,
706                                           d weightMatriz);
```

A última extensão é a definição de variáveis para a identificação de *threads* e blocos. A identificação de um bloco pode ser acessada por um *kernel* através das variáveis *blockIdx.x, blockIdx.y e blockIdx.z*. Similarmente, a identificação dos threads pode ser obtida pelas variáveis, *threadIdx.x, threadIdx.y,* e *threadIdx.z.* A combinação dessas seis variáveis identifica unicamente cada *thread* na execução de determinada função.

A Figura 6 exemplifica uma função do tipo __global__, que executa a soma de dois vetores, na qual se usa o identificador do bloco para determinar qual elemento do vetor será acessado por cada *thread*. Por essa configuração pode-se concluir que o programa executará corretamente se forem criados N blocos com um *thread* em cada, sendo N a dimensão dos vetores, já que é usado apenas o identificador do bloco para indexar estes vetores.

Figura 6: Exemplo de função do tipo kernel que soma dois vetores de forma paralela.

```
7  __global__ void add ( float *a, float *b, float *c ) {
8
9          int i = blockIdx.x;
10         c[i] = a[i] + b[i];
11 }
```

Esses conceitos de krigagem e programação paralela em CUDA C já são suficientes para o entendimento da metodologia utilizada e das análises sobre os resultados obtidos, tratadas nos capítulos a seguir.

3 MATERIAIS E MÉTODOS

Após a familiarização com os conceitos de programação em GPU usando a linguagem CUDA C, deu-se início ao estudo da implementação do algoritmo de krigagem feita para monografia de Castelo Branco (2010). Na seção 3.1 será apresentada a implementação do algoritmo serial da krigagem, assim como as modificações que foram necessárias, tanto para finalizar sua codificação como para facilitar o processo de paralelização. A seção 3.2 descreve a análise feita para se decidir quais fases do algoritmo valeriam o esforço de paralelização, enquanto a seção 3.3 detalha a implementação paralela das partes do algoritmo selecionadas na seção anterior. Na 3.4 são descritos os testes a serem feitos para medir o desempenho do código.

3.1 Algoritmo Serial de Interpolação por Krigagem

Ao se analisar o código, percebeu-se que o mesmo usava orientação a objetos em todas as suas funcionalidades. Além do sistema fazer uso de várias classes para interface gráfica e geração de gráficos, a estrutura dos objetos facilitava a compreensão do funcionamento do mesmo. Porém alguns dos vetores e matrizes associados aos cálculos eram armazenados em forma de listas de objetos o que certamente torna o cálculo mais lento que vetores ou matrizes, devido ao padrão de acesso à memória dessas estruturas. Dessa forma, achou-se por bem reescrever a parte do código referente à krigagem no paradigma procedural, em linguagem C, usando-se vetores unidimensionais para armazenar os dados, padrão amplamente difundido em programação paralela.

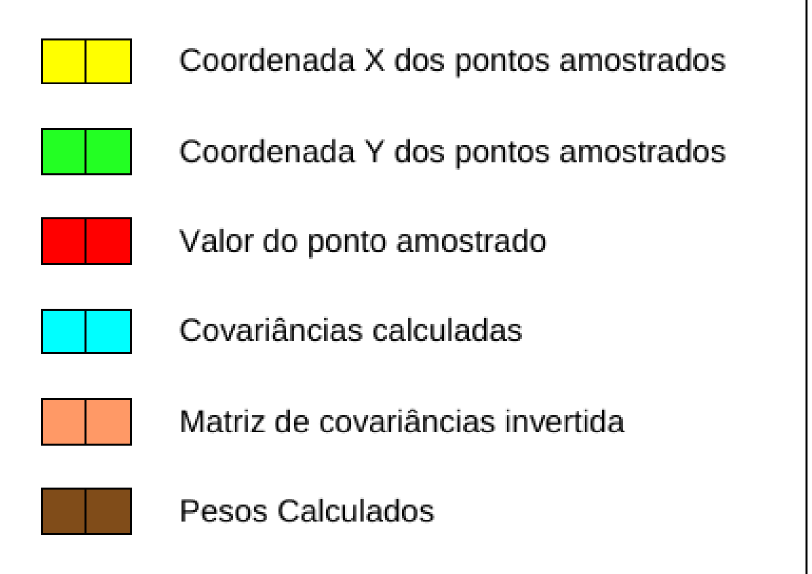
A seção a seguir detalha o algoritmo, já no paradigma procedural, pontuando as diferenças entre o código original e o reescrito. Entretanto, para facilitar a compreensão do mesma, a Tabela 1 descreve as variáveis e constantes mais importantes para o entendimento das etapas do algoritmo.

Tabela 1: Varáveis e constantes que configuram a krigagem serial

Variável/Constante	Descrição
N	Variável determinada pela quantidade de pontos amostrados.
Histogram_nbits	Variável que determina a quantidade de subdivisões nas abscissas do semivariograma experimental
INTERVALS	Constante que determina a quantidade de pontos que terá um lado da grade quadrada de pontos que será gerada. O quadrado desta constante representa o total de pontos a ser estimado.

Durante a explicação de determinadas etapas do algoritmo, tanto na próxima seção como em outras posteriores, são utilizadas representações de matrizes e vetores, em que cada cor representa um dado específico. Essas cores e suas correspondências estão na Figura 7.

Figura 7: Legenda de cores



3.1.1 Implementação do Algoritmo de Krigagem

O arquivo de entrada que alimenta o programa deve estar no formato texto e possuir três colunas, sendo as duas primeiras as coordenadas do ponto e a terceira o valor amostrado

naquele ponto, configurando um ponto por linha. Um trecho desses arquivos pode ser observado na Figura 8.

Figura 8: Trecho de arquivo de entrada do SAPoCaPT

```
8681538	622033,56	0
8681538	622076,81	0
8681541	622117,38	0
8681542	622159,13	0
8681543	621993	0
8681543	622172,88	0
8681550	621952,69	0
8681551	622009,19	2
8681551	622051,94	2
8681552	622094,19	2
8681559	622134,25	2
```

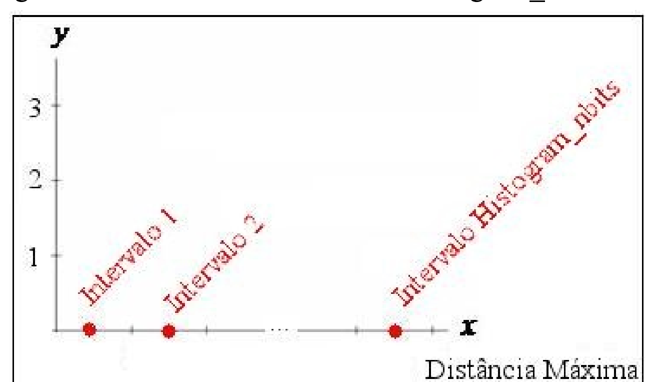
O trecho de arquivo da Figura 8 é convertido em vetores, em que **N** é o número de pontos do arquivo. O passo seguinte do algoritmo é o cálculo da distância de cada ponto a todos os demais, resultando numa matriz de distâncias, que está representada na Figura 9. Como a distância de todo ponto até ele mesmo é nula, a diagonal dessa matriz é nula. A distância entre dois pontos aparece na matriz duas vezes, pois é possível acessar a distância de P1 a P2, e também a de P2 a P1. Apesar dessa representação, a distância entre dois pontos é calculada apenas uma vez.

Figura 9: Representação da matriz de distâncias.

Essas distâncias têm importância fundamental para determinação do semivariograma. Primeiramente define-se o número de abscissas que o histograma deve ter, através do parâmetro **Histogram_nbits**, como ilustrado na Figura 10. Dessa forma, o eixo X do histograma, que abrange de zero à maior distância, tem esse intervalo divido em

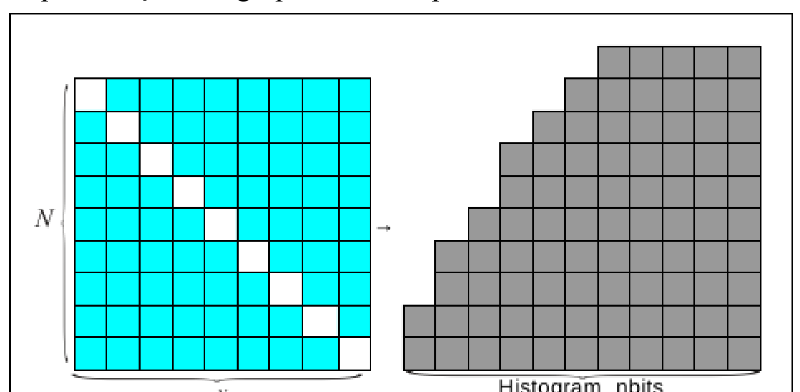
Histogram_nbits intervalos. O valor da abscissa presente no gráfico é o ponto médio de cada um desses intervalos.

Figura 10: Distâncias divididas em Histogram_nbits intervalos



Em seguida, esses pontos são agrupados em **histogram_nbits** grupos, de acordo com as distâncias. Esse processo é representado na Figura 11, na qual o quadrado à esquerda, que representa a matriz de distâncias, é reagrupado no polígono à direita, onde os mesmos pontos estão agrupados por distância, sendo os mais a esquerda mais próximos entre si. Com os pontos agrupados, podemos usar a equação do semivariograma para determinar o eixo das ordenadas.

Figura 11: Representação do agrupamento dos pontos de acordo com as distâncias entre eles.



A obtenção do semivariograma experimental é a primeira etapa do algoritmo e tem como única função auxiliar o usuário do programa a escolher os parâmetros adequados para determinação do semivariograma sintético.

O semivariograma sintético é a simples codificação de uma equação – o modelo matemático esférico de um semivariograma. Quando o usuário sugere os parâmetros para a

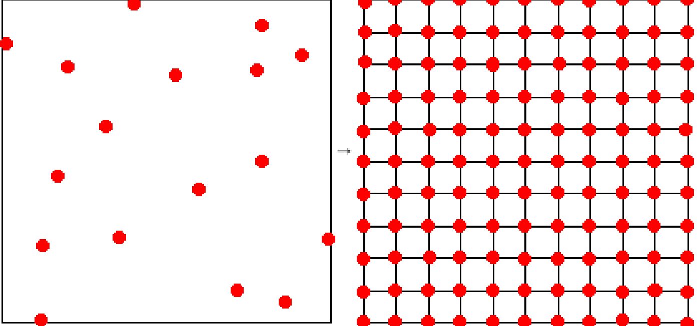
determinação do semivariograma sintético, esse deve ser plotado para que seja analisada a proximidade do modelo teórico com as amostras. O usuário poderá trocar os parâmetros, especificando assim um novo semivariograma, até que esteja satisfeito com o modelo obtido.

Tão logo o semivariograma sintético seja definido, já haverá dados suficientes para o cálculo das covariâncias. Todas as covariâncias calculadas ficam armazenadas numa matriz quadrada da ordem **N**. Porém, para satisfazer o sistema do algoritmo da krigagem ordinária, é necessário acrescentar uma coluna e uma linha, ambas preenchidas com o valor 1, para representar a variável temporária e o somatório dos pesos, respectivamente, que são requisitos para a krigagem ordinária.

Em seguida, para a resolução do sistema, verifica-se se é possível inverter a matriz de covariância. Isso é feito calculando-se o determinante da mesma. Caso seja possível, faz-se a inversão da matriz de covariâncias, que serve como base para a resolução do sistema da krigagem. Caso não, o programa é finalizado.

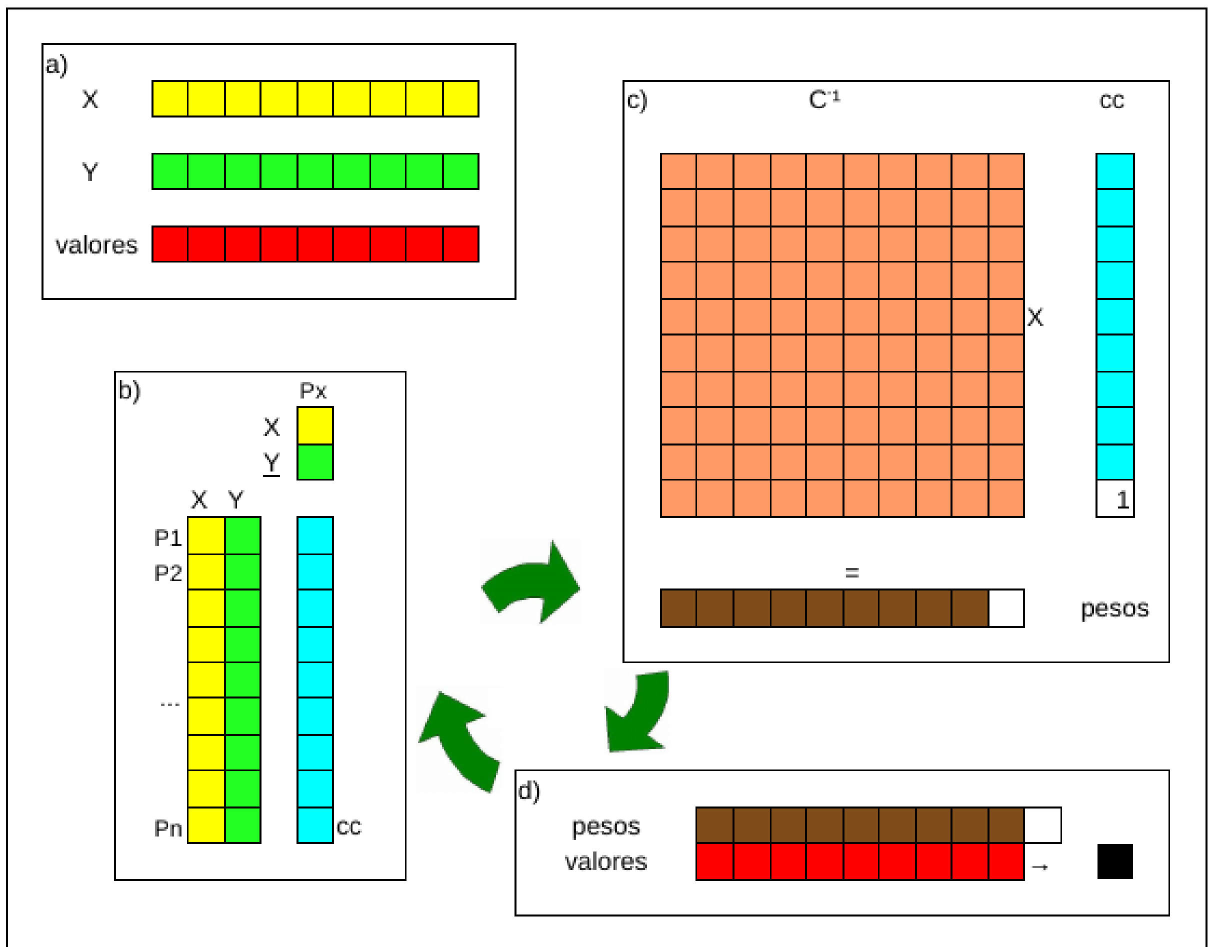
Todos os passos do algoritmo descritos até aqui são executados apenas uma vez e o volume de cômputo depende apenas de **N**. O cômputo realizado nas próximas etapas estão em um laço que depende da quantidade de pontos a ser estimada. Na Figura 12 é possível observar como o processo é feito: faz-se um quadrado a partir das maiores e menores coordenadas dos pontos em cada eixo e divide-se cada eixo em **INTERVALS** partes iguais, formando uma grade de pontos em que se conhece as coordenadas dos pontos e se quer estimar o valor para os mesmos.

Figura 12: Determinação da grade de pontos a serem estimados, a partir dos pontos amostrados.



Com as coordenadas de todos os pontos para os quais se quer estimar valores, basta calcular um por vez em um laço, representado pela Figura 13.

Figura 13: Representação do laço para determinação dos pontos por krigagem, quando a matriz inversa já está calculada.



A Figura 13a representa as coordenadas e valores dos pontos amostrados, que serão utilizados no laço representado pelo restante da figura.

Na Figura 13b é representada a parte em que se calcula as covariâncias do ponto que se quer estimar – Px – em relação aos amostrados. O cálculo é o mesmo usado para determinar a matriz de covariâncias, porém, como se trata de apenas um ponto em relação aos demais, o resultado, ao invés de uma matriz quadrada da ordem dos pontos amostrados, é apenas uma matriz coluna. O produto da matriz de covariâncias invertida, previamente calculada, por essa matriz coluna determina os pesos que serão utilizados para determinação dos pontos. Essa etapa, a resolução do sistema de krigagem, está representada na Figura 13c, que, como mencionado anteriormente, precisa ser resolvido para cada ponto.

Finalmente, a Figura 13d representa a estimação do ponto, que é o produto de cada peso calculado por um dos valores amostrados, gerando assim uma soma ponderada, que resulta no ponto estimado.

Todo esse processo – Figura 13 b, c e d – é repetido para a determinação de cada um dos pontos. Agora que já se conhece a forma como os cálculos são feitos, é interessante observar que não é necessário guardar as coordenadas de todos os pontos que serão estimados. Basta apenas dois vetores, um com as posições do eixo X e outro com as do eixo Y. Dessa forma, usam-se dois laços aninhados, de forma que cada valor de X seja combinado com cada valor de Y, gerando assim toda a grade de pontos.

É interessante observar que o código original do SAPoCaPT calculava apenas um ponto, portanto, além de portar o código para o paradigma procedural concluiu-se a implementação do algoritmo da krigagem para tornar possível a paralelização do mesmo.

Cada etapa da transcrição do código foi testada, usando-se o programa original para garantir a fidelidade dos resultados. Fazer isso era necessário, pois o programa original foi validado por Castelo Branco (2010) utilizando scripts para krigagem no Matlab.

Após a conclusão da codificação, foi necessário um estudo dos tempos de execução de cada etapa do algoritmo, para determinar quais deveriam ter prioridade no processo de paralelização, assunto da próxima seção.

3.2 Estudo dos Tempos de Execução

Com a implementação concluída, pôde-se medir o tempo de execução de cada etapa do código para diversos valores de entrada e avaliar quais etapas valeriam o esforço de paralelizar.

O algoritmo foi dividido em cinco etapas para medição do tempo de execução: cálculo dos semivariogramas, cálculo da matriz de covariância, cálculo do determinante da matriz de covariância, inversão da matriz de covariância e interpolação dos pontos.

Cada um dos testes foi repetido três vezes, com o intuito de minimizar interferências externas no tempo de execução do programa. A média dos valores obtidos é utilizada nos gráficos e tabelas.

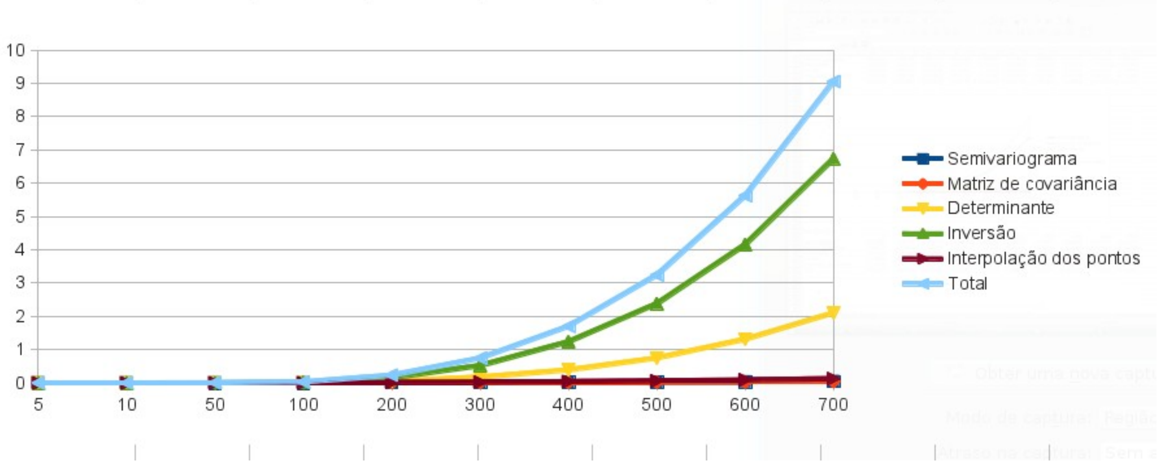
Primeiramente, foram feitos testes com o número constante de pontos a serem interpolados, variando-se apenas o número de amostras analisadas. Na Tabela 2 pode-se visualizar o tempo de execução de cada uma dessas etapas, e também presentes no Gráfico 3,

para deixar a visualização mais clara. O número de pontos amostrados varia de 5 a 700, para um número de pontos interpolados sempre igual a 25.

Tabela 2: Tempo de execução de cada etapa (s), para interpolação de 25 pontos

Etapa do Algoritmo	Quantidade de Amostras							
	5	100	200	300	400	500	600	700
Semivariograma	0,00001	0,001199	0,004755	0,009686	0,014356	0,024573	0,032052	0,042891
Matriz de cov.	0,000002	0,001499	0,007387	0,010416	0,015584	0,022322	0,029708	0,042224
Determinante	0,000003	0,006542	0,052479	0,177893	0,391283	0,749152	1,312578	2,103335
Inversão	0,000007	0,020422	0,15759	0,522089	1,236935	2,379221	4,152882	6,733554
Interpolação	0,00004	0,003116	0,011435	0,024433	0,042281	0,065398	0,094761	0,129238
Total	0,000062	0,032778	0,233646	0,744517	1,700439	3,240666	5,621981	9,051242

Gráfico 3: Tempo de execução de cada etapa, para interpolação de 25 pontos com o algoritmo serial.



Da análise da Tabela 2, pode-se concluir que o tempo de execução de qualquer etapa aumenta quando há aumento do número de amostras. No Gráfico 3, é possível observar que a etapa de inversão da matriz e a de cálculo do determinante são as que têm maior duração com o aumento do número de amostras, portanto, são boas candidatas para se estudar a paralelização.

O Gráfico 3 apresenta uma visão sobre a variação do tempo de execução do algoritmo para um número crescente de amostras e sem alteração no número de pontos interpolados. Porém, para casos reais, o número de pontos interpolados será sempre muito superior ao número de pontos amostrados. Dessa forma, uma análise mais detalhada sobre os tempos de execução de cada etapa requer o aumento do número de pontos interpolados.

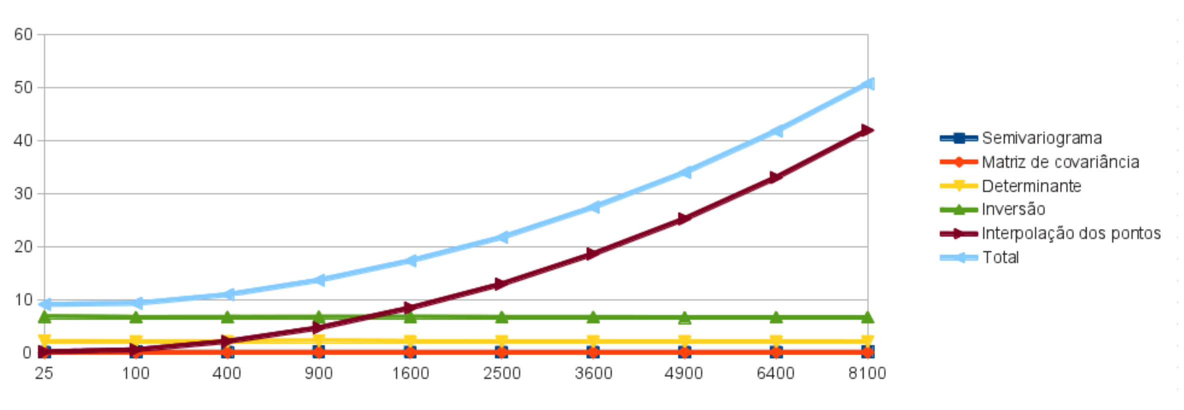
Para atender a esse requisito, fizeram-se novas medições de tempo, mantendo-se o número de amostras constante em 700, e variando-se o número de pontos em função

quadrática, já que a grade gerada é quadrada. O resultado das medições pode ser observado na Tabela 3 e Gráfico 4.

Tabela 3: Tempos de execução do algoritmo serial (s), aumentando-se a quantidade de pontos interpolados

Etapa do Algoritmo	Quantidade de Pontos Interpolados							
	100	400	900	1600	2500	3600	4900	6400
Semivariograma	0,04614	0,04301	0,04495	0,043531	0,043376	0,04261	0,044309	0,043108
Matriz de cov.	0,03968	0,04080	0,04126	0,042224	0,040112	0,04054	0,03987	0,039651
Determinante	2,05718	2,06938	2,19691	2,088102	2,063299	2,06092	2,069395	2,059734
Inversão	6,6118	6,64338	6,68666	6,707801	6,64233	6,63018	6,586428	6,602248
Interpolação	0,51077	2,09014	4,64053	8,396112	12,899044	18,5850	25,17855	32,96159
Total	9,26560	10,8867	13,6103	17,27777	21,688161	27,3592	33,91855	41,70633

Gráfico 4: Tempo de execução de cada etapa, para um número crescente de pontos interpolados.



Da análise da Tabela 3 e do Gráfico 4, pode-se observar que mesmo se aumentando bastante o número de pontos interpolados para o mesmo número de amostras, o tempo de execução das quatro primeiras etapas não sofre alteração. É possível também observar que com o acréscimo do número de pontos interpolados, a etapa de interpolação passa a consumir a maior parte do tempo de cômputo, chegando a ser superior a 80% no caso de se interpolar 8100 pontos para 700 amostras. A curva do gráfico sugere que esse número tende aumentar muito mais para um número maior de pontos interpolados.

Através da análise desses dados, pôde-se observar que o cálculo do determinante e a inversão da matriz têm contribuição importante no tempo de cômputo, porém o tempo gasto

com a interpolação é bastante superior a ambos, nos casos em que há mais de 1600 pontos. Portanto, decidiu-se trabalhar na paralelização dessa etapa prioritariamente.

3.3 Paralelização da Interpolação

A interpolação no programa serial acontece num laço que executa três passos repetidas vezes. No programa paralelo, descrito nessa sessão, o número de execuções desse laço foi reduzido e os três passos foram substituídos por três tarefas paralelas executadas sequencialmente.

Antes do laço com a repetição de passos é feita a determinação das coordenadas da grade de pontos que será calculada. As três tarefas executadas em laço são: 1) o cálculo das colunas de covariâncias, para cada um dos pontos determinados na etapa anterior, 2) o produto da matriz inversa, obtida na etapa serial, pelas colunas de covariância, resultando nos pesos e 3) o uso desses pesos calculados para determinar os pontos estimados.

Antes do detalhamento dessas etapas, é válido ressaltar que o desenvolvimento das mesmas foi feito pensando-se nos testes que seriam executados. Dessa forma, algumas constantes e variáveis foram acrescentadas às já definidas na Tabela 1 para configurar o comportamento do sistema, visando definir, por exemplo, as dimensões dos blocos para a chamada de uma função do tipo *kernel*. Elas estão detalhadas na Tabela 4.

Tabela 4: Varáveis e constantes que configuram o sistema

Variável/Constante	Descrição
TILEWIDTH	Constante relacionada com a dimensão dos blocos
PARALLEL_POINTS	Constante que determina a quantidade de pontos que será calculada por iteração do laço

3.3.1 Determinação da Grade de Pontos

Para que a krigagem seja realizada, é necessário primeiramente se ter as coordenadas dos pontos para os quais se querem estimar valores, no caso, a já referida grade de pontos.

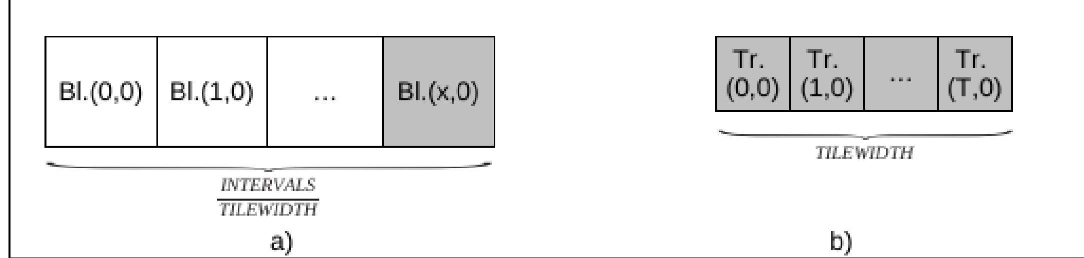
Analisando as operações da interpolação no algoritmo serial, pode-se observar que existe primeiramente a determinação dos pontos a serem interpolados, em que os eixos X e Y do mapa são gerados, criando-se intervalos igualmente espaçados entre os valores máximos e mínimos de X e Y respectivamente.

A principio não existe qualquer motivação para a paralelização desse trecho do código, pois se trata apenas de uma pequena quantidade de operações simples. No entanto, todas as operações a partir dessa etapa serão realizadas na GPU e, assim sendo, têm-se duas opções: montar o vetor no *host* e transferi-lo para a GPU, ou montar o vetor na própria GPU. Nesta última opção, seria necessária apenas a cópia do valor máximo e mínimo, para o preenchimento dessa estrutura que é um método bastante simples de implementar. Baseado nisso, optou-se por criar uma função em CUDA para preencher os eixos da grade de pontos.

A Figura 15 ilustra o *grid* e os blocos que são configurados para se preencher um eixo da grade de pontos. Cada eixo tem **INTERVALS** pontos, portanto é necessário apenas esse mesmo número de *threads* para que cada um deles preencha um valor no vetor alocado. O *grid* tem apenas uma dimensão, e a quantidade de blocos é determinada pela constante **TILEWIDTH**. A quantidade de *threads* **t** em um bloco é determinada pela equação 8:

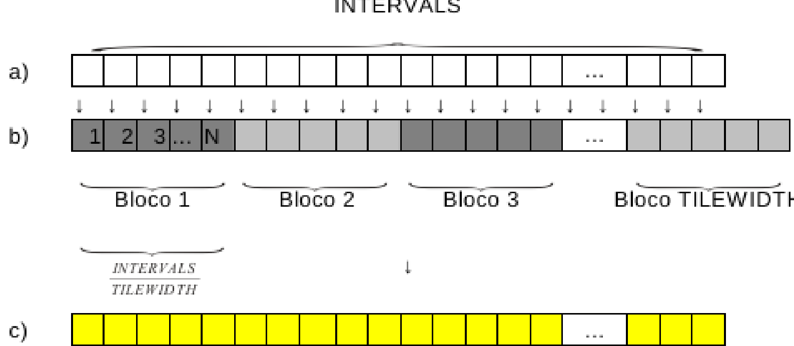
$$t = \frac{INTERVALS}{TILEWIDTH} \tag{8}$$

Figura 14: Configurações do a) grid e b) blocos para o preenchimento dos eixos



Primeiramente aloca-se no dispositivo o vetor unidimensional vazio onde os pontos de um eixo serão preenchidos. A Figura 15a) ilustra esse vetor, cuja quantidade de pontos é determinada pela constante **INTERVALS**.

Figura 15: Preenchimento de um Eixo



A Figura 15b representa o *grid* configurado para esse método de preenchimento do vetor da Figura 15a. O *grid* tem apenas uma dimensão, e a quantidade de blocos é determinada pela constante **TILEWIDTH**. Como ambas as constantes podem assumir valores inteiros diferentes, não se pode contar com uma divisão exata. Nesse caso, **t**, da equação 8, assumirá o valor do maior inteiro mais próximo e existirão *threads* no último bloco que não executarão nenhuma operação, fato também ilustrado na Figura 15b, onde os dois últimos *threads* do último bloco ficam ociosos.

Com a configuração do *grid* estabelecida, a partição dos dados se torna trivial, já que cada *thread* deve simplesmente escrever um número em uma posição do vetor previamente alocado. Dessa forma, a posição do vetor que será modificada é determinada através da combinação do número e dimensão do bloco com o número do *thread*, como representado pela equação 9:

$$pos = BlockIdx.x * BlockDim.x + ThreadIdx.x \quad (9)$$

Com a posição da memória a ser usada já definida, basta-se agora executar o cômputo na mesma, que é o cálculo do valor que vai ser gravado na posição especificada do vetor. Esse valor é calculado através da soma da menor coordenada do eixo com o passo multiplicado pelo índice do vetor, como na equação 10. Isso feito, o vetor encontra-se preenchido, como representado pela Figura 15c).

$$coordenada = coordenadaInicial + passo * pos \quad (10)$$

Esse método preenche apenas um eixo. Portanto, como a krigagem está sendo feita no plano, esse método precisa ser chamado duas vezes. Além do paralelismo inerente à implementação descrita, as duas vezes que esse método precisa ser chamado podem acontecer em paralelo.

Com os pontos determinados, segue-se para a etapa seguinte, o cálculo das colunas de covariância. Esse cálculo depende das coordenadas de todos os pontos, portanto precisa esperar até que os eixos sejam determinados para que possa ser executado corretamente.

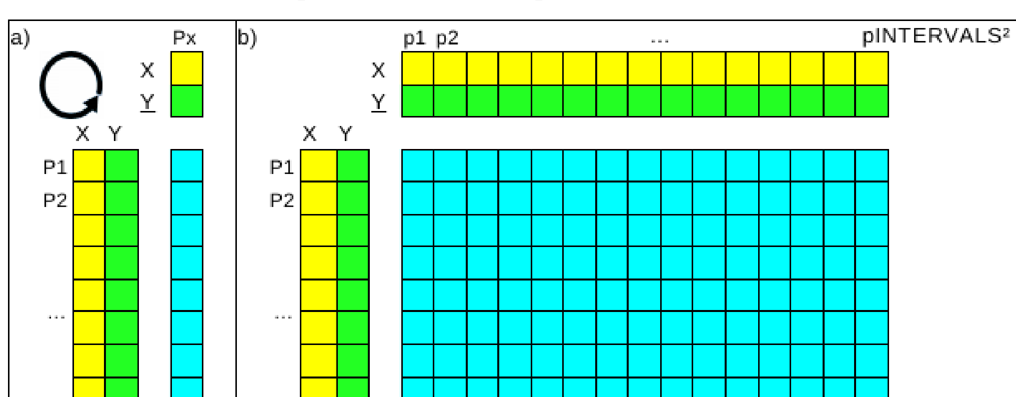
A determinação dos eixos só é feita uma vez na execução da krigagem, portanto, fica fora do laço que executa as demais etapas. Antes de entrar neste laço, além de já ter as coordenadas desses pontos a serem estimados no dispositivo, é necessário copiar outras estruturas obtidas nas etapas seriais do programa, como a matriz invertida das covariâncias, as coordenadas e valores dos pontos amostrados.

3.3.2 O Cálculo das Colunas de Covariâncias

No algoritmo serial, a cada iteração do laço é calculada a covariância do ponto a que se quer estimar em relação a todos os pontos amostrados, e esse resultado é armazenado em uma matriz coluna. Para aumentar a eficiência do paralelismo, optou-se por calcular todas as colunas de covariância de uma só vez. A diferença das implementações é retratada na Figura 16, em que a Figura 16a) mostra o laço que é executado **INTERVALS²** vezes – número total de pontos a serem estimados –, enquanto a Figura 16b) representa o processamento de todas as **INTERVALS²** colunas, usando paralelismo.

O número de pontos a serem gerados é especificado pelo usuário, portanto, como o valor pode ser arbitrário, é possível que ele seja grande o suficiente para que não haja memória no dispositivo capaz de armazenar todas as estruturas relacionadas com os cálculos. Para isso foi definida a constante **PARALLEL_POINTS**, que limita o número máximo de colunas que pode ser processada em uma iteração, permitindo que não haja implicação no consumo de memória quando se aumenta o total de pontos – **INTERVALS²**.

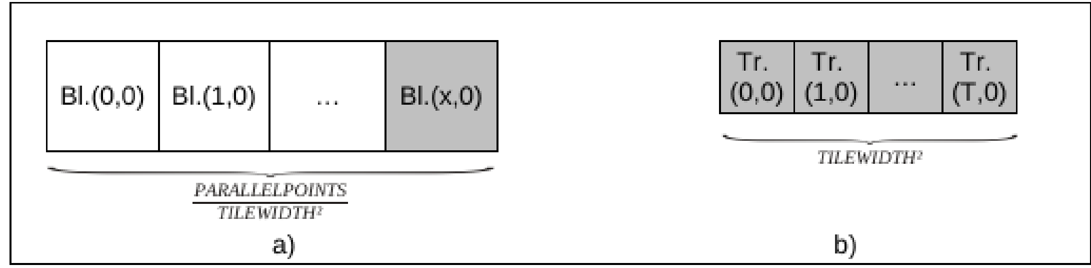
Figura 16: Cálculo da coluna de covariâncias: a) versão serial, repetida INTERVALS² vezes, b) versão paralela executada apenas uma vez



Na Figura 16 está representado o agrupamento das colunas de covariâncias que formam uma matriz de ordem **N** x **INTERVALS**², em que **N**, como já mencionado, é o numero de pontos amostrados. Essa configuração só ocorre quando **INTERVALS**² é menor que **PARALLEL_POINTS**. Quando isso não acontece, a matriz da figura é particionada em matrizes de ordem **N** x **PARALLEL_POINTS**.

A configuração do *grid* especificado para a resolução do problema pode ser observada na Figura 17a). Trata-se de um *grid* unidimensional, cuja coordenada X varia de zero a $\frac{PARALLELPOINTS}{TILEWIDTH}-1$. A Figura 17b) representa a configuração de cada bloco desse *grid*, também unidimensional, com as coordenadas limitadas pela constante **TILEWIDTH²**.

Figura 17: Configurações a) do *Grid* e b) dos blocos.

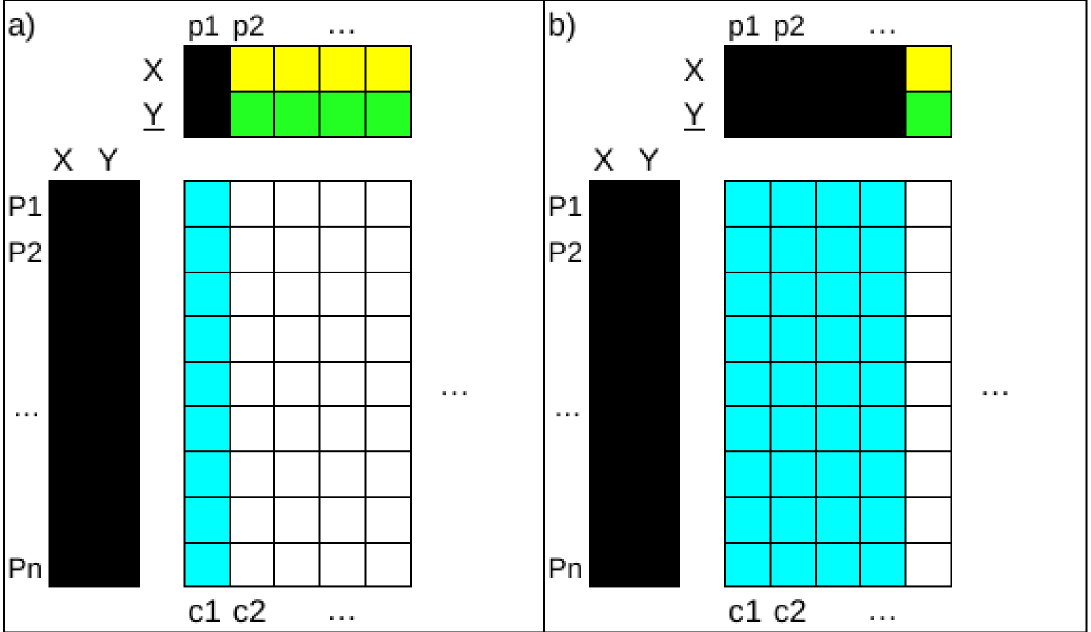


Com essa configuração, tem-se um total de **PARALLEL_POINTS** *threads* que é exatamente o número de pontos que está sendo estimado na iteração corrente. Dessa forma, cada *thread* calcula todos os elementos de uma coluna de covariâncias.

Para se chegar à covariância é necessário calcular a distância entre os pontos. Dessa forma, para se determinar a covariância em relação a cada ponto é preciso ter acesso às coordenadas dos mesmos. Assim, como um *thread* irá calcular todas as covariâncias do ponto

para o qual se quer estimar um valor em relação aos pontos amostrados, o mesmo vai precisar acessar as coordenadas de todos os pontos amostrados. A Figura 18a destaca com a cor preta os dados da memoria global que um *thread* precisa acessar para calcular uma coluna de covariâncias.

Figura 18: Dados necessários da memória global para a) calcular uma coluna de covariâncias e b) calcular quatro colunas



A Figura 18b destaca dos dados que serão necessários para o cálculo de quatro colunas de covariâncias. Uma comparação entre as Figuras 18 a e b é possível concluir que as coordenadas dos pontos amostrados serão requisitados por todos os *threads*, portanto, se cada um deles carregar um ponto e compartilhar na memória compartilhada, os acessos à memória global serão reduzidos.

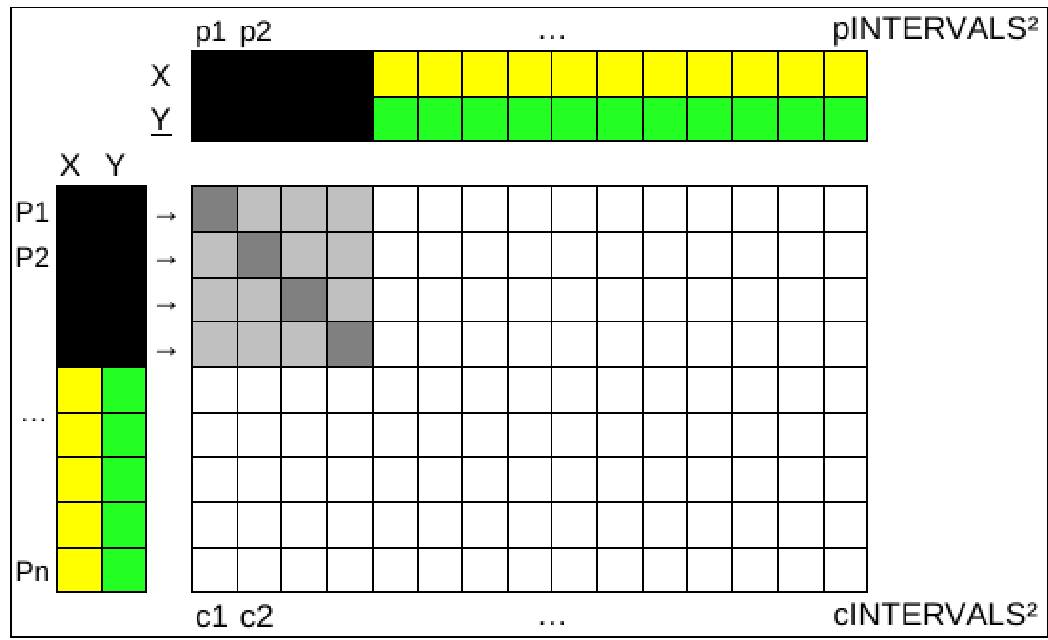
Outro fato que a Figura 18 deixa mais claro é que as coordenadas dos pontos que serão estimados (localizadas acima na figura), são utilizadas em todos os cálculos de covariâncias da coluna. Dessa fora é interessante que cada *thread* armazene esse valor em um registrador, para não precise fazer um acesso à memória global pra cada covariância a ser determinada.

Para fazer uso da memória compartilhada, o algoritmo foi dividido em duas partes: a cópia dos dados para a mesma, seguida pelo cálculo das covariâncias. Como a memória compartilhada é limitada e a quantidade de pontos amostrados é variável, é possível que seja necessário copiar os dados para a memória compartilhada várias vezes, até que se chegue ao final das colunas de covariâncias calculadas por aquele bloco.

A carga dos dados é bastante simples. São criados dois vetores com o número de elementos igual ao número de *threads* num bloco – **TILEWIDTH²** – um para armazenar as abscissas e outro, as ordenadas dos pontos amostrados. Dessa forma, cada *thread* carrega colaborativamente as coordenadas de um ponto, sendo necessária sincronização para garantir que todos os *threads* já tenham carregado suas respectivas coordenadas. Sincronizar os threads após a etapa de carga de dados na memória compartilhada garante que os mesmos possam passar para a próxima etapa tendo, na referida memória, todos os dados de que vão precisar na iteração.

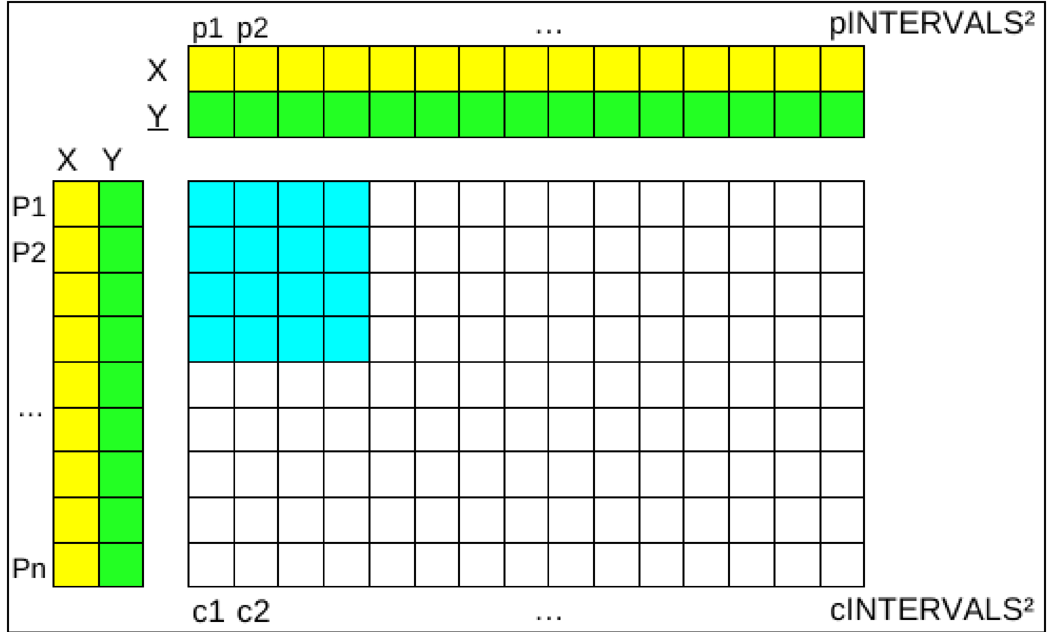
Essa etapa inicial é representa na Figura 19, na qual **TILEWIDTH** é igual a dois, ou seja, há quatro *threads* carregando as coordenadas dos pontos amostrados, como indicado pelas setas, e nenhuma covariância ainda foi calculada. Na figura, que representa apenas a execução de um bloco para simplificar a compreensão, 16 covariâncias vão ser calculadas, porém, ao invés de cada *thread* carregar as coordenadas de quatro pontos para calcular as quatro covariâncias, cada um deles carrega apenas uma, e compartilha com os demais. Isso pode ser visualizado na Figura 19, levando-se em consideração que os quadrados cinza mais escuros representam as coordenadas trazidas por aquele *thread*, enquanto os mais claros representam as coordenadas que serão aproveitadas da memória compartilhada, pois já foi carregada por outro *thread*.

Figura 19: Representação de um bloco na fase inicial do cálculo das covariâncias, onde os *threads* obtêm as coordenadas dos pontos.



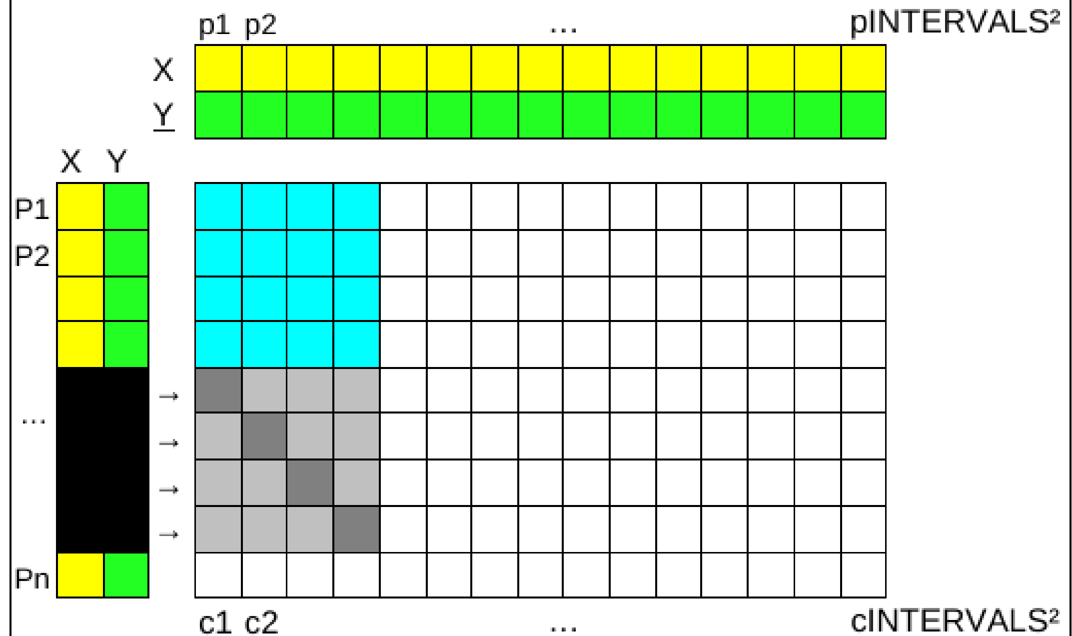
Assim que cada um dos *threads* carregou as coordenadas de seu respectivo ponto na memória compartilhada, o bloco de *threads* já tem dados suficientes para calcular, no exemplo, 16 covariâncias, como pode ser observado na Figura 20, em que, cada um dos quatro *threads* calcula quatro elementos da sua coluna de covariâncias. Assim que os pontos são calculados, há novamente necessidade de sincronização para garantir que todos os valores presentes na memória compartilhada já tenham sido utilizados antes que os mesmos sejam substituídos. A figura também ilustra que nessa parte do cálculo não há acessos a memória global do dispositivo, já que os dados necessários foram carregados na memória compartilhada e nos registradores, como já descrito.

Figura 20: Representação do término da primeira e início da segunda execução do laço que calcula as covariâncias.



A substituição dos valores para a segunda etapa do laço é representada na Figura 21. Nela, é possível observar que não há mais necessidade de se carregar as coordenadas dos pontos que estão sendo estimados (acima), pois elas já estão armazenadas nos registradores, e serão úteis até o término do calculo da coluna de covariâncias.

Figura 21: Substituição dos dados na memória compartilhada



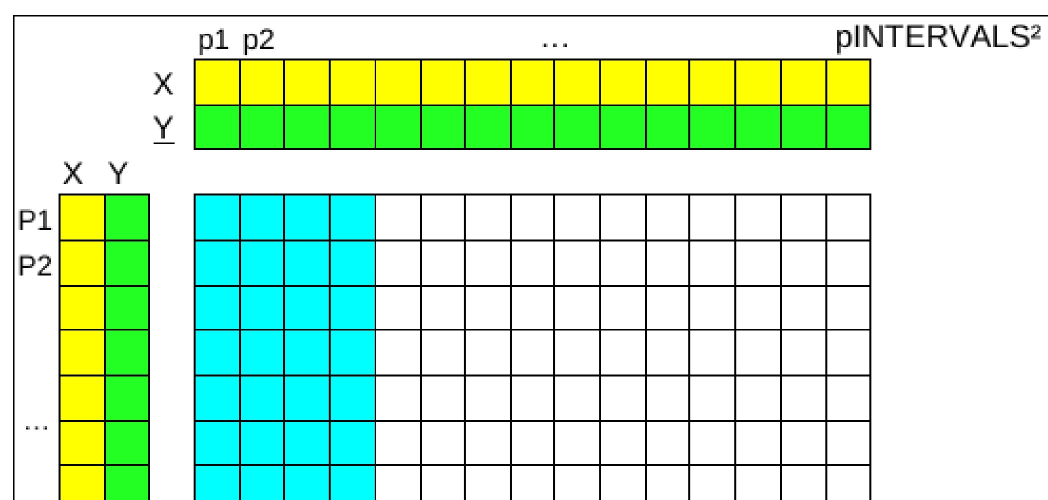
Fazer uso da memória compartilhada introduz complexidade ao código, pois se cada *thread* simplesmente lesse as coordenadas dos pontos da memória principal, não haveria necessidade de se dividir o trabalho em etapas, nem de se sincronizar as mesmas. Porém, como a memória global tem uma latência alta, o esforço tende a ser recompensado. Nesse exemplo, são feitos quatro acessos a memória para o cálculo de 16 covariâncias, reduzindo-se os acessos de 16 para 4. Generalizando-se, o acesso à memória global do dispositivo será reduzido a

$$\frac{1}{TILEWIDTH^2}$$

do total, desde que a memória compartilhada seja grande o suficiente para manter toda essa informação, que também aumenta com **TILEWIDTH**.

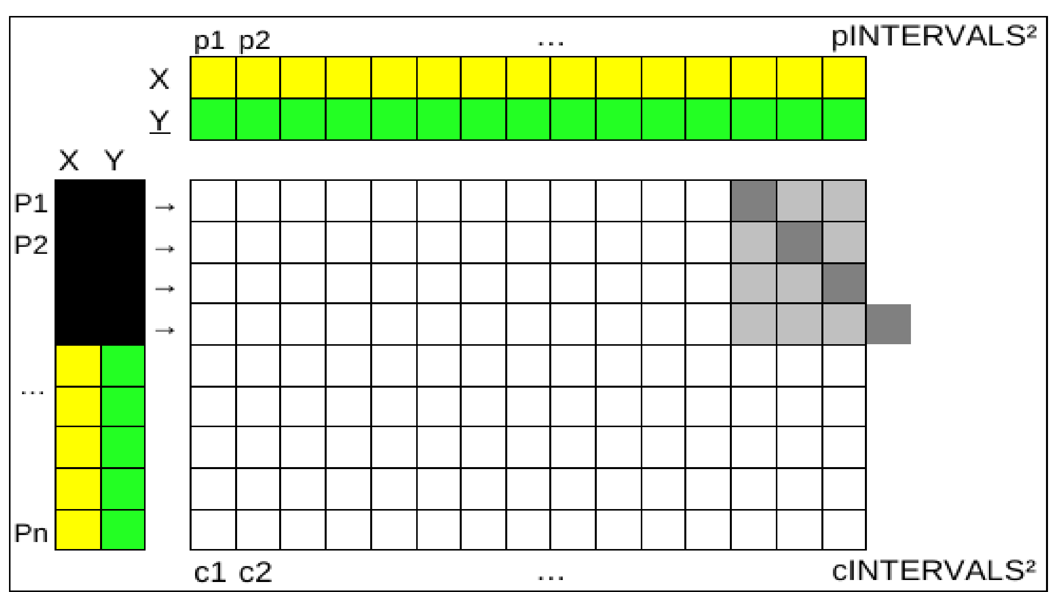
Além de se preocupar com o tamanho da memória compartilhada alocada, é necessário também prever os casos especiais que podem acontecer nas bordas da matriz de dados resultante, que nem sempre terá o mesmo tamanho do *grid*. Na Figura 22, por exemplo, pode-se observar que, para a configuração em questão, a última execução do laço que calcula as covariâncias deve prever que não será necessário carregar quatro pontos para o cálculo de 16 covariâncias, pois só existem quatro covariâncias a serem calculadas e apenas um ponto cujas coordenadas precisam ser carregadas.

Figura 22: Um bloco genérico na última iteração: a quantidade de pontos pode ser menor que nas anteriores.



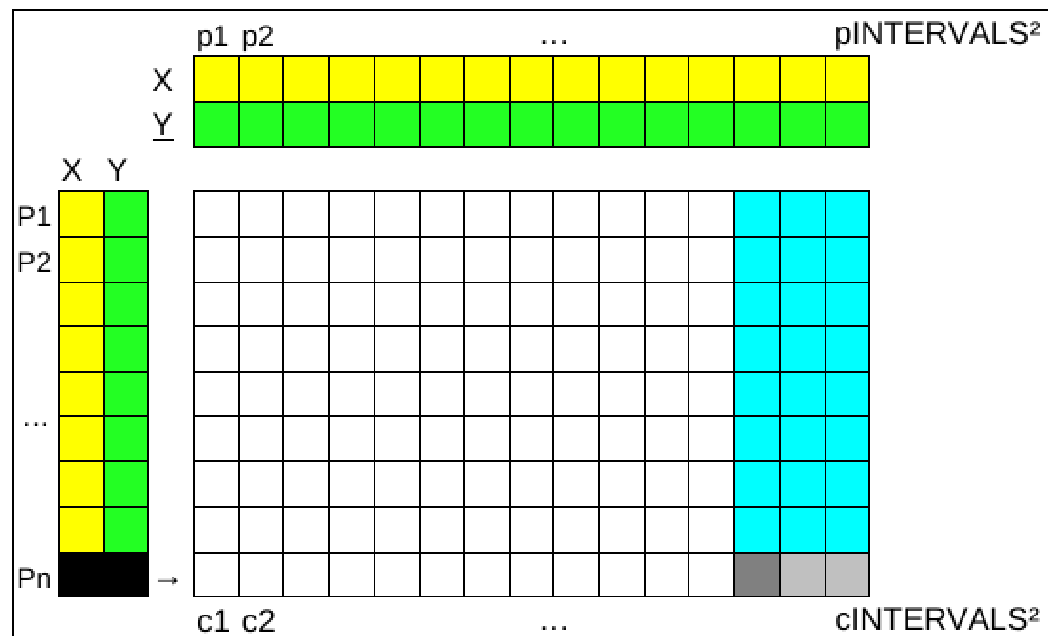
Um problema similar acontece com o último bloco, como representado pela Figura 23, em que pode não haver colunas para todos os *threads* calcularem. Assim, podem existir *threads* que não executem nenhum cômputo, porém elas ainda precisam carregar a memória compartilhada de forma colaborativa para que os demais *threads* tenham os dados necessários aos seus respectivos cômputos.

Figura 23: Início da execução do último bloco: ele pode ter menos colunas para calcular que os demais blocos.



Por último, é necessário prever quando esses dois problemas descritos se sobrepõem, como ilustrado pela Figura 24, em que é possível ter *threads* que não executem cômputo nem carga de dados.

Figura 24: Combinação dos dois problemas apresentados nas figuras anteriores.



Por fim, cada *thread* que está calculando uma coluna acrescenta o valor 1 ao final da mesma para completar o sistema de equações da krigagem ordinária, que inclui a soma dos pesos.

Com todas as colunas de covariâncias calculadas, já se têm os dados necessários para a próxima etapa do algoritmo, que é o cálculo dos coeficientes.

3.3.3 Determinação dos Coeficientes

A paralelização do cálculo dos coeficientes se deu de forma similar à do cálculo das covariâncias, pois antes esse cálculo também era feito para cada ponto a ser estimado e passou a ser realizado em conjunto. A determinação dos coeficientes é simplesmente a resolução do

sistema da krigagem ordinária, isto é, a multiplicação da matriz de covariâncias invertida pela coluna de covariâncias, como mostra Figura 25, numa abordagem serial.

Analisando as operações, optou-se por agrupar as colunas de covariâncias lado a lado e realizar apenas um produto matricial, ao invés de vários, como pode se observado na Figura 26, que mostra uma configuração similar ao produto da Figura 25, em que a segunda matriz representa várias colunas de covariâncias agrupadas, e a terceira, vários pesos agrupados. Dessa forma, a determinação dos coeficientes foi reduzida a uma multiplicação de matrizes.

Figura 25: Determinação dos pesos de forma serial

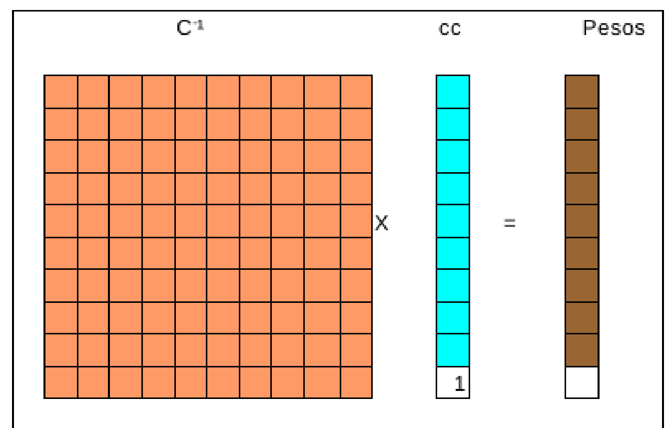
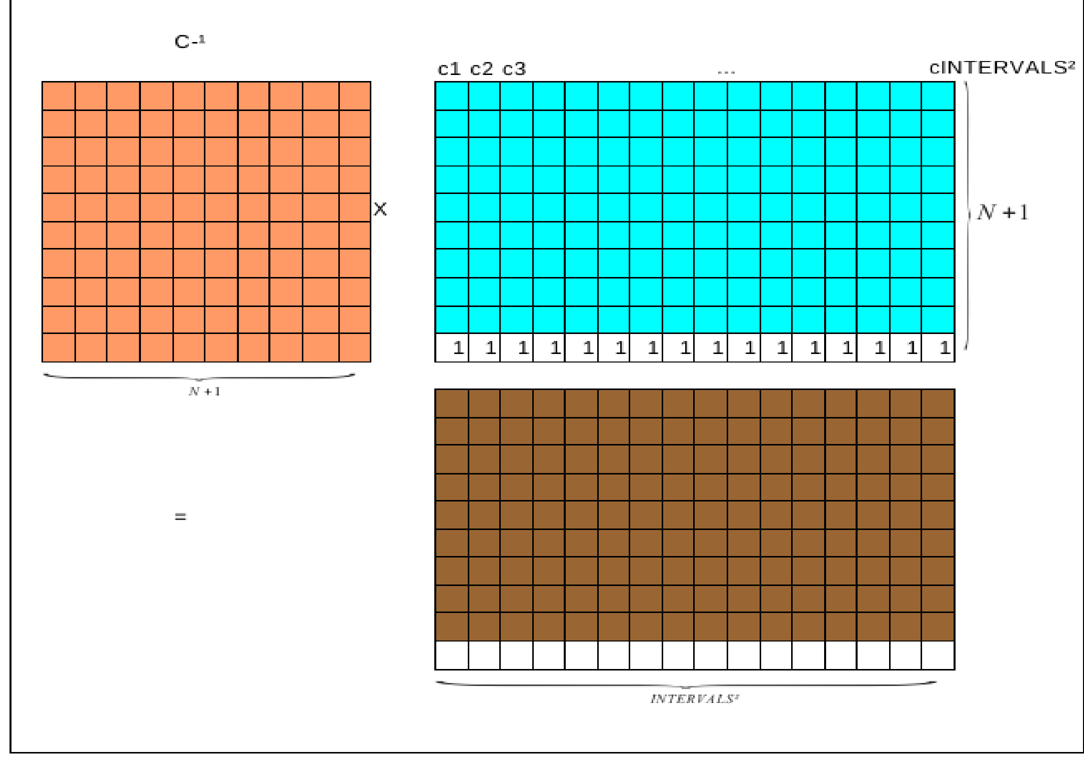


Figura 26: Determinação dos pesos de forma paralela.



Com todos os dados necessários na memória do dispositivo, basta chamar-se a função do tipo *kernel*, especificando as dimensões do *grid* e dos blocos. Na Figura 27 estão representados o *grid* e o bloco em, respectivamente, a e b. Como pode ser observado, os blocos são quadrados, com dimensão determinada pela constante **TILEWIDTH**, porém o *grid* tem suas dimensões proporcionais às dimensões da matriz resultante do produto, para cada *thread* calcule um ponto.

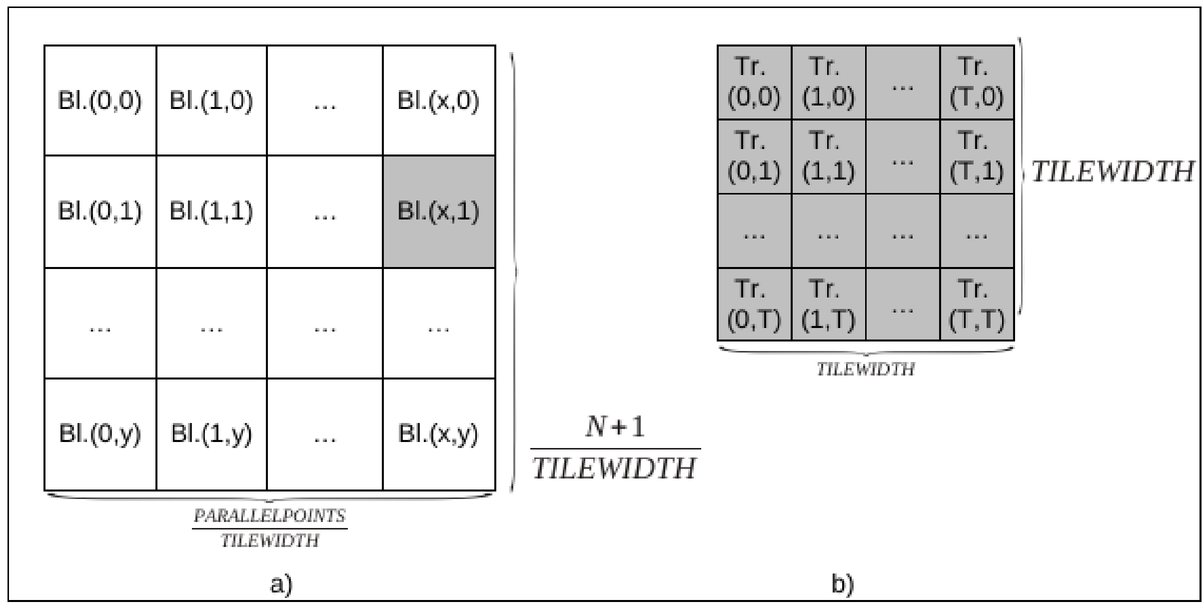
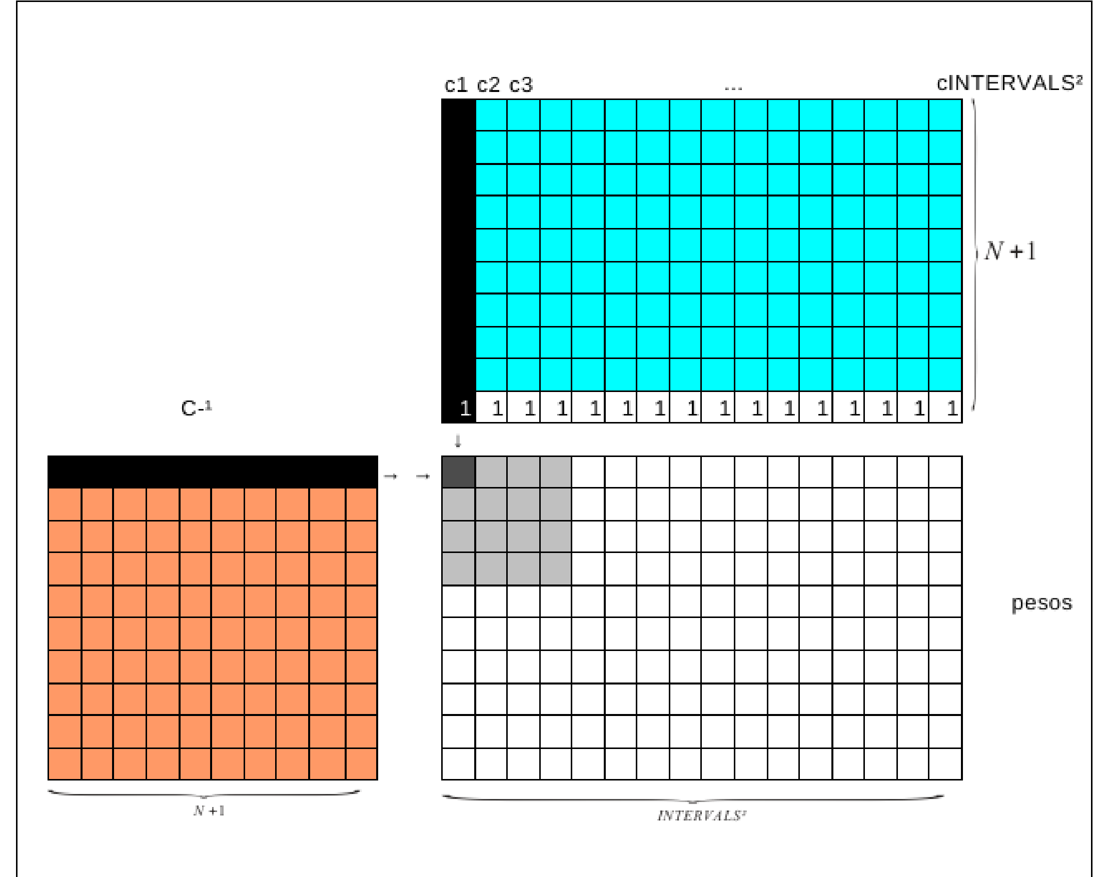


Figura 27: a) Grade e b) blocos configurados para o cálculo dos pesos.

Para se calcular cada elemento da matriz produto, precisa-se acessar uma linha inteira da primeira matriz e uma coluna inteira da segunda matriz. O valor deste elemento, no caso um peso, é a soma de todos os produtos de cada um dos elementos da linha da primeira pelo seu correspondente na coluna da segunda, como representado na Figura 28.

A Figura 28 apresenta a matriz de pesos vazia (sem preenchimento) com um bloco em destaque no início da mesma, que representa um bloco de *threads* e, como pode ser observado, dispõe os threads em um quadrado. Cada thread calcula um ponto, sendo que a figura destaca o primeiro *thread* do bloco (mais escuro) e os elementos das matrizes que precisam ser acessados para o cálculo desse peso.

Figura 28: Acesso à memória para determinação de um ponto da matriz produto.



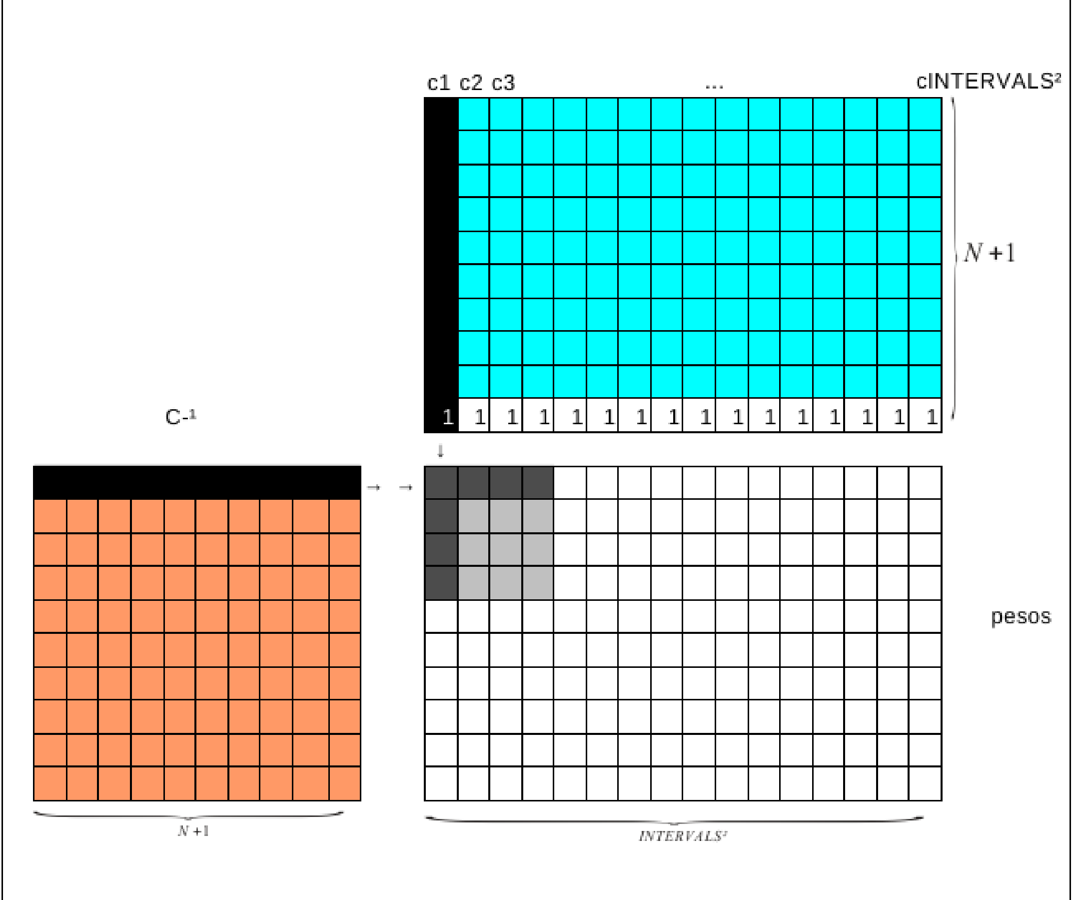
A Figura 29 destaca outros *threads* para os quais seriam úteis os mesmos dados da memória global utilizados pelo primeiro, ou seja, das matrizes que estão sendo multiplicadas. Através da análise desta figura, nota-se que cada linha carregada poderá ser usada por **TILEWIDTH** *threads* de um bloco e cada coluna por também **TILEWIDTH** *threads* do mesmo bloco.

Já que cada elemento carregado das matrizes terá utilidade para **TILEWIDTH** *threads*, optou-se por carregar estes elementos das matrizes na memória compartilhada, de forma colaborativa. No esquema escolhido, cada *thread* carrega um ponto de cada matriz e, ao final desse processo, têm-se duas submatrizes na memória compartilhada. O produto dessas submatrizes não requer acessos a memória global, e resulta em um resultado parcial dos pesos que se deseja obter. A repetição desses produtos de submatrizes iterativamente resulta no produto final das matrizes completas.

A Figura 30 deixa a estratégia adotada mais clara, quando destaca os elementos das matrizes que estão sendo carregados na memória compartilhada. Esses quadrados têm a mesma dimensão do bloco, pois cada *thread*, como mencionado, armazena apenas um elemento de cada matriz por vez. É necessário sincronizar os threads nessa etapa, para

garantir que todos os pontos estejam carregados durante a etapa seguinte, que é a soma parcial dos produtos escalares, ou seja, multiplicação das submatrizes.

Figura 29: Padrão de acessos repetidos à memória.



Para garantir que cada *thread* tenha realizado a soma parcial dos produtos escalares é necessário mais uma vez sincronizá-los. A partir deste pondo, há uma nova carga da memória compartilhada, em que os elementos carregados anteriormente são substituídos pelos seguintes, como esquematizado na Figura 31. É possível observar que esses elementos são os **TILEWIDTH** próximos das mesmas **TILEWIDTH** linhas, no caso da primeira matriz, e os **TILEWIDTH** próximos das mesmas **TILEWIDTH** colunas na segunda matriz.

Esse processo é repetido até que se chegue ao final das linhas e colunas em que se está trabalhando. Porém, na última iteração, o número de elementos em cada linha e coluna podem ser menor que **TILEWIDTH**, portanto haverá sobra de espaço na memória compartilhada alocada. Para que não haja necessidade de modificação nas operações matemáticas realizadas, optou-se por preencher os espaços excedentes com zero, como representado na Figura 32a). Dessa forma, a soma final não será alterada, já que os produtos escalares serão zero.

Figura 30: Carga colaborativa dos primeiros elementos das matrizes

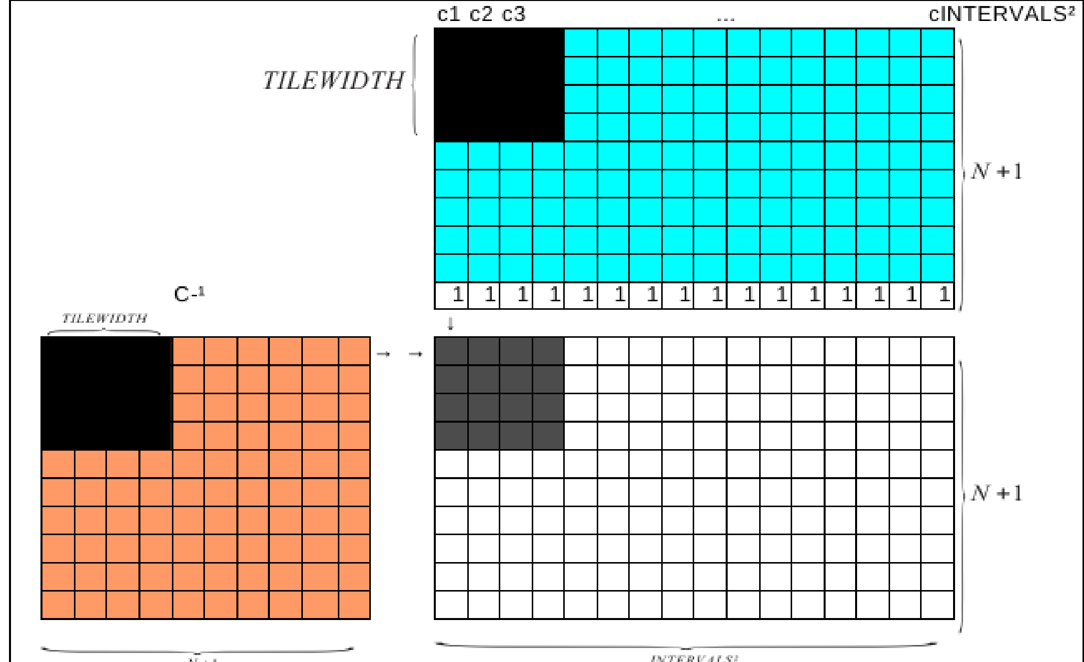
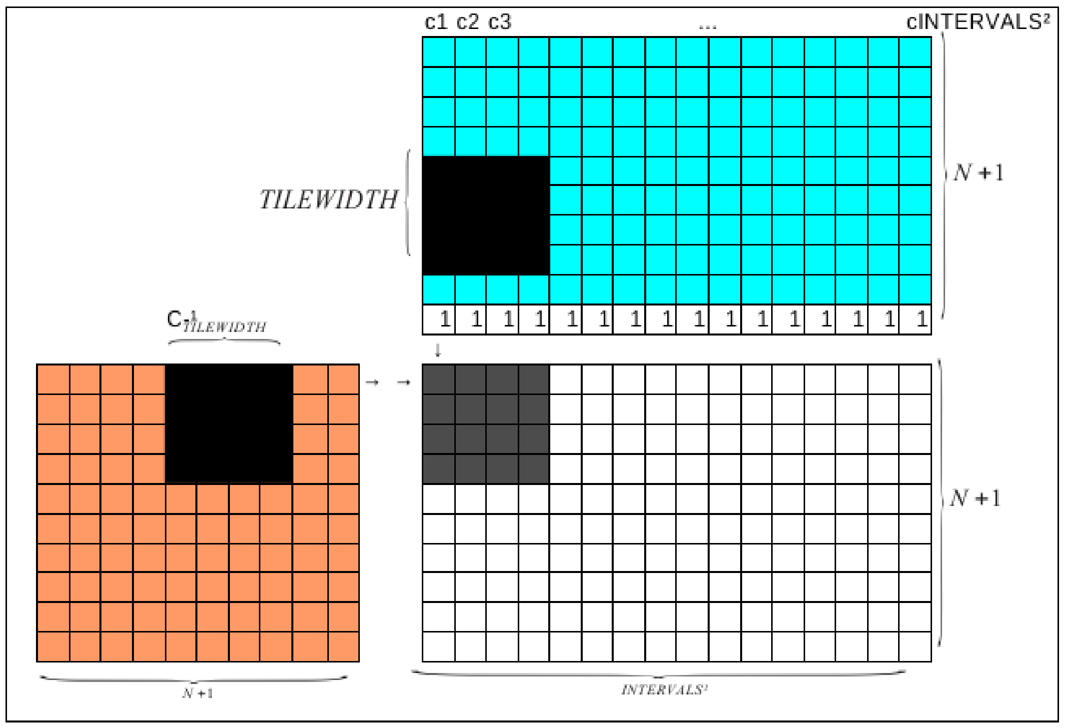
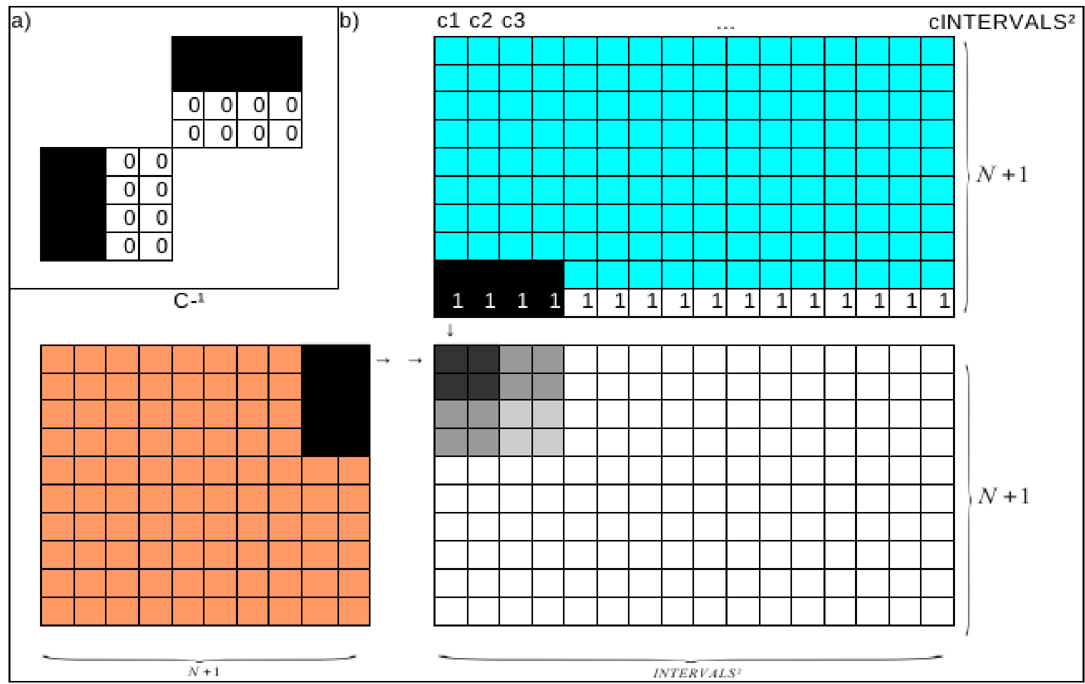


Figura 31: Segunda etapa da carga colaborativa da memória compartilhada.



A Figura 32b) representa a execução da última iteração de cada *thread*, onde os últimos elementos das linhas e colunas são carregados. No exemplo, por conta das linhas restantes não serem suficientemente grandes para preencher a variável alocada na memória, o acesso à memória global é reduzido. Alguns *threads* carregam elementos das duas matrizes, outros, apenas de uma delas e outros não realizam acessos. Isso é representado pelos tons de cinza na figura, onde os mais escuros realizam acesso às duas matrizes e os mais claros a nenhuma, apenas preenchendo com zero a sua parcela da memória compartilhada.

Figura 32: a) preenchimento da memória compartilhada quando não há elementos suficientes para o preenchimento total e b) finalização do cálculo dos pesos para o bloco em questão.

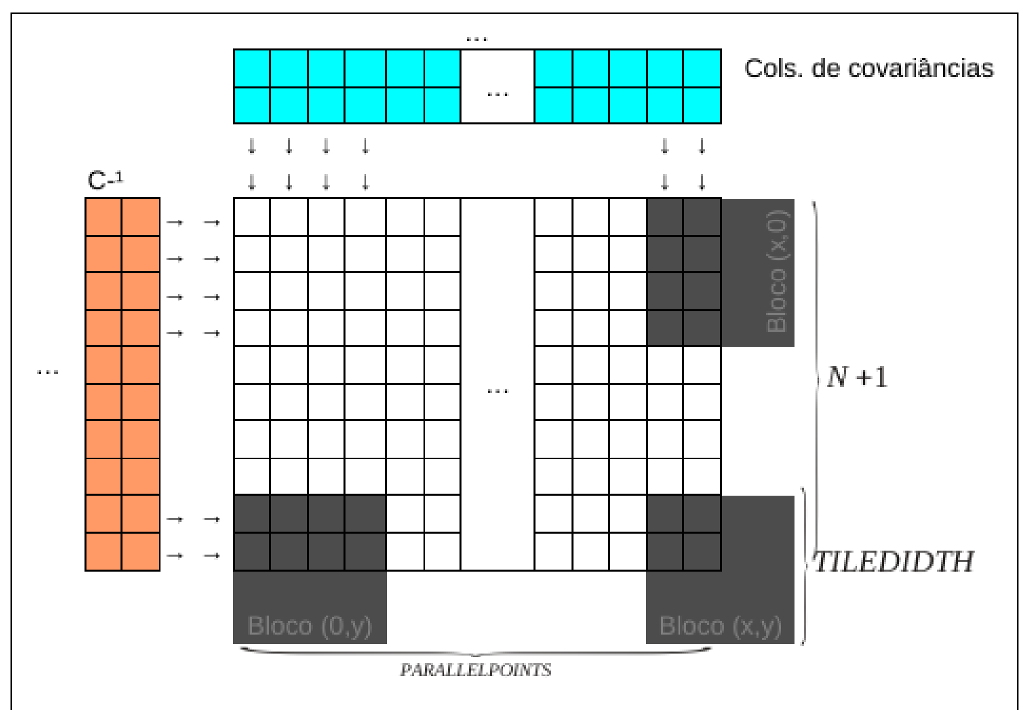


Nas bordas da matriz que está sendo calculada, alguns blocos podem ter um número menor de pontos para calcular que de *threads*. É mais fácil visualizar o problema, que só existe quando **PARALLEL_POINTS** ou **N** não são divisíveis por **TILEWIDTH**, observando-se os três blocos em destaque na Figura 33.

Todos os *threads* que estão dentro das dimensões da matriz produto vão realizar cômputo para determinar o ponto pelo qual é responsável. Os demais não realizarão cômputo, porém devem participar do preenchimento da memória compartilhada. Como é possível observar ainda na Figura 33, no Bloco (0,y), todo os threads carregam dados das colunas de covariâncias – como indicado pelas setas, porém apenas metade deles carregam dados das linhas da matriz de covariâncias invertida. Exatamente o oposto ocorre no Bloco (x,0), no

qual não há colunas suficientes para que todos os threads carreguem, mas há linhas. Uma combinação dessas duas configurações acontece com o Bloco (x,y), no qual existem threads que além de não realizarem cômputo, não fazem acessos à memória, apenas preenchendo os valores com zero na memória compartilhada.

Figura 33: Casos em que os blocos podem ter menos pesos para calcular que threads.



Com a execução dos passos descritos e respeitando-se as restrições, o *grid* configurado inicialmente calcula todos os pesos que serão usados para determinar **PARALLEL_POINTS** pontos.

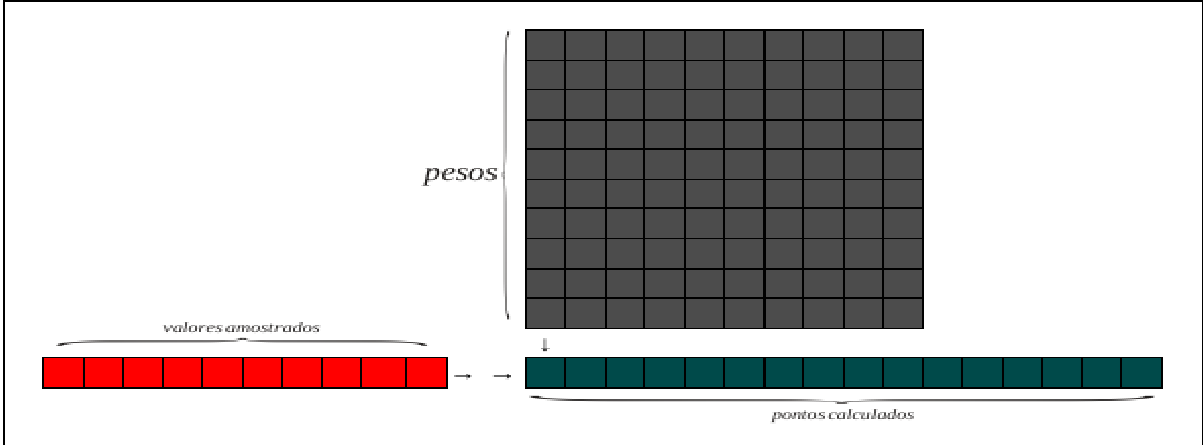
3.3.4 A estimativa dos Pontos

Com todos os pesos calculados, determinar os pontos é tarefa simples: basta fazer a média ponderada de todos os valores amostrados, usando os pesos obtidos como os ponderadores.

Como ilustrado na Figura 34, a determinação dos pontos, ponderando-se os valores amostrados com os pesos calculados, pode ser interpretado também como um produto de matrizes. Nesse produto, como representado na figura, a primeira matriz terá apenas uma linha, com os pontos amostrados, e a segunda matriz terá em suas colunas os pesos calculados para a determinação de cada ponto.

É interessante observar que o vetor resultante possui todos os pontos estimados, independente da variável **PARALLEL_POINTS** e, portanto, deve mapear o resultado dos produtos de acordo com a iteração do laço. O código foi projetado dessa maneira para que não houvesse nenhuma transferência de dados entre o dispositivo e a memória principal do computador dentro do laço que realiza o cômputo. Dessa forma, depois que todo cômputo é realizado, os pontos são transferidos da memória do dispositivo de vídeo para o *host*.

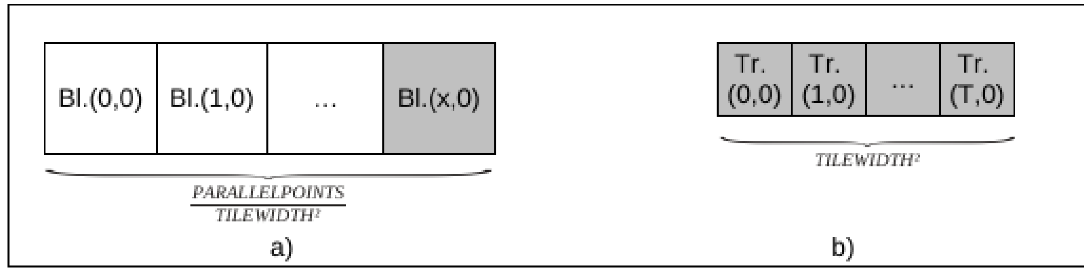
Figura 34: Cálculos dos pontos, usando-se os valores amostrados e os pesos previamente calculados.



Por se tratar de um caso particular de multiplicação de matrizes, no qual a primeira matriz do produto é sempre uma matriz linha, preferiu-se criar um método alternativo em relação ao descrito anteriormente, que pudesse explorar melhor essa particularidade na disposição dos dados, pois, para qualquer ponto calculado do produto será necessário ler toda a linha da primeira matriz e cada coluna da segunda matriz só será lida uma vez, sendo necessário apenas se fazer cache da primeira matriz.

A configuração do *grid* usada neste método é idêntica à usada na determinação das colunas de covariâncias, como reproduz a Figura 35.

Figura 35: a) Grade e b) blocos configurados para o cálculo dos pesos.

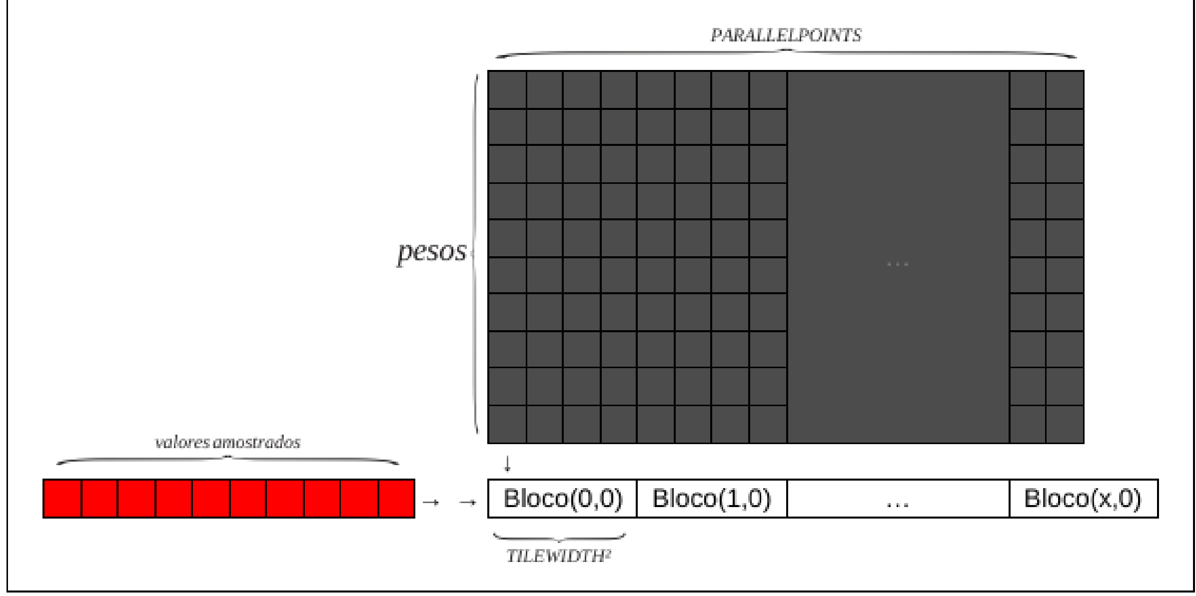


Em um mesmo bloco, os *threads* fazem carga colaborativa da matriz linha, para reduzir acessos à memória global. A Erro: Origem da referência não encontrada ilustra como esses blocos estão agrupados no *grid* em relação aos dados necessários ao cômputo desta etapa.

Outros detalhes são bastante similares aos já explicados na função de multiplicação de matrizes descrita anteriormente, portanto, serão omitidos

.

Figura 36: Disposição dos blocos no *grid* para cálculo dos pontos.



3.4 Método Usado nos Testes

O programa foi exaustivamente testado para se verificar a correção dos resultados e a eficiência do mesmo na exploração dos recursos de processamento disponíveis nos dispositivos de vídeo.

Com o objetivo de facilitar os testes foi criado um método para gravar os resultados de execução do programa em um arquivo. Além das coordenadas dos pontos estimados e seus respectivos valores, o arquivo apresenta informações de tempo de execução das etapas do programa, característica da GPU do sistema, e as constantes que configuram o mesmo.

A parte serial do programa já contou com as medições de tempo das seis etapas do programa, a saber:1) determinação do semivariograma experimental; 2) determinação do semivariograma sintético; 3) cálculo da matriz de covariância; 4) cálculo do determinante da matriz de covariâncias; 5) inversão da matriz de covariâncias; e 6) cálculo dos pontos estimados.

De posse desses recursos, deve-se avaliar a correção do algoritmo, comparado-se o resultado do programa serial com o paralelo, usando o primeiro como base para se calcular o erro do segundo.

Em seguida, deve-se determinar uma boa configuração para as constantes **TILE_WIDTH**, **PARALLEL_POINTS**, que estão diretamente ligadas ao tamanho dos blocos e grids, de forma que o hardware possa ser bem aproveitado quando se queira estimar um grande número de pontos.

Conhecendo-se boas configurações para o programa, deve-se executá-lo em hardware com diferentes características, para avaliar a eficiência com que o mesmo usa o hardware paralelo, e até mesmo comparar os tempos de execução do programa serial com o paralelo.

4 EXPERIMENTOS E RESULTADOS

4.1 Ambiente de Teste

O programa foi desenvolvido em C/C++/CUDA C usando o IDE (*Integrated Development Environment* – Ambiente Integrado para Desenvolvimento) Qt-Creator. Embora o programa resultante possa ser compilado para diversas plataformas, todos os testes foram realizados em ambiente Linux.

Três placas gráficas estiveram disponíveis para testes, as quais são listadas na Tabela 5, com algumas de suas características.

Tabela 5: Dispositivos disponíveis para testar o programa

Placa	Versão da arquitetura CUDA	Quantidade de CUDA *cores*	Memória de vídeo
GeForce 310	1.2	16	512 MB
GeForce GT 240	1.2	96	512 MB
Tesla C2070	2.0	448	6 GB

Para aumentar a precisão dos resultados, decidiu-se fazer a medição do tempo três vezes em cada um dos testes realizados, e obter a média. Essas medições estão relacionadas a trechos específicos do código que realizam cômputo, portanto tempos de entrada e saída, como a leitura do arquivo de entrada ou a escrita do arquivo com o resultado dos testes, não são computados.

Foram selecionados dez arquivos com amostras de propriedades de algumas zonas de um poço de petróleo, que possuem números diferentes de amostras.

4.2 Precisão e Exatidão

Foi realizado um primeiro teste com 412 amostras para se estimar 10 mil pontos, utilizando-se a versão serial do programa e a versão paralela sendo executada em dispositivos com versões diferentes da arquitetura CUDA. Alguns dos pontos calculados estão dispostos na Tabela 6, que mostra os valores obtidos na diferentes execuções.

Os valores estimados são bem próximos, coincidido em alguns casos. Algumas pequenas diferenças já eram esperadas, devido às limitações de representação dos números de ponto flutuante. Nesse teste, tomando-se como referência a execução do programa serial, o erro médio para a arquitetura 1.2 foi de $5,36\times10^{-8}\%$ e, $3,85\times10^{-8}\%$ para a arquitetura 2.0.

Tabela 6: Pontos estimados por diferentes versões do programa.

Coordenadas		Pontos estimados		
x	y	Programa serial	CUDA 1.2	CUDA 2.0
621051	8681325	1,55761397	1,55761397	1,55761385
621051	8681853	7,67889595	7,678895	7,678895
621159,75	8681560	4,13051081	4,13050795	4,13051081
621181,5	8681423	0,99228656	0,99228585	0,9922874
621268,5	8681833	6,07785463	6,07785273	6,07785177
621290,25	8681403	0,25077489	0,25077412	0,25077429

Não se pode dizer qual das três execuções do programa tem o valor mais próximo do exato, pois todas elas trabalham com precisão finita, o que ocasiona acúmulo de erros ao longo da série de operações que são realizadas para se calcular um valor, como já discutido neste trabalho.

Ainda assim, o fato dos valores estarem próximos entre si sugere que todos estejam próximos do número exato e, independente disso, todos os resultados obtidos estão suficientemente próximos da execução serial do programa, que é a referência.

Foram feitos vários outros testes com amostras diferentes e em diferente número e os resultados foram similares.

4.3 Configuração do Programa

Após se analisar a precisão do programa fez-se necessário determinar qual a melhor configuração das variáveis **PARALLEL_POINTS** e **TILE_WIDTH** para os dispositivos disponíveis, antes da realização de testes de desempenho.

4.3.1 Quantidade de Pontos Calculados por Iteração

A constante **PARALLEL_POINTS** determina quantos pontos serão calculados paralelamente, portanto, tem influência direta na velocidade de execução do programa, pois quanto mais pontos forem calculados paralelamente, melhor o aproveitamento das unidades de processamento disponíveis. Porém, se essa variável tiver um valor muito alto, o uso de memória no dispositivo será muito alto, ou até mesmo o programa pode parar por não conseguir alocar memória suficiente.

Já que o valor dessa constante impacta diretamente no consumo de memória e na utilização dos processadores, e deseja-se analisar o comportamento do software para valores crescentes da mesma, decidiu-se usar para esse teste da Tesla C2070 por possuir mais memória e mais unidades de processamento que as demais placas disponíveis para testes.

O cálculo do número total de pontos estimados é dividido em iterações de **PARALLEL_POINTS** pontos por vez. Já que se quer variar a quantidade de pontos que serão calculados por iteração, é importante que se tenha uma grande quantidade de pontos a serem calculado, para que a medida que o valor de **PARALLEL_POINTS** cresça, ainda tenha um número grande de cômputo para particionar.

Para isso, escolheu-se o maior arquivo disponível, com 2450 amostras, desejando-se estimar 1 milhão de pontos, com 256 *threads* por bloco.

A Tabela 7 mostra a duração e a quantidade de memória alocada da etapa paralela do programa para alguns valores de **PARALLEL_POINTS**. Da análise da mesma, pode-se concluir que não há ganho de desempenho significativo para valores da constante superiores a 1.000. Em outras palavras, o cálculo de 1.000 pontos por iteração consegue ocupar os 448 processadores da placa tão bem quanto o cálculo de 50.000 pontos por iteração.

Vale ressaltar que a quantidade de memória presente na Tabela 7 se trata apenas da memória alocada explicitamente para armazenar as matrizes necessárias aos cálculos. Deve-se considerar que a memória real necessária no dispositivo será maior, devido à chamada de

funções e as próprias camadas de software da arquitetura. Ainda assim, pode-se observar que, quando **PARALLEL_POINTS** assume valores acima de 1.000, há um grande aumento na memória necessária para armazenar essas estruturas. Usar, por exemplo o valor 50.000 exige quase 1GB de memória alocada, quantidade não disponível nas duas outras placas.

Com base nesses resultados, optou-se por usar em **PARALLEL_POINTS** o valor 1.000 nas duas placas menos poderosas e 10.000 na outra, alocando, respectivamente, 45 MB e 214 MB em suas memórias de vídeo. Com estes valores fixados, o uso da memória varia apenas em função da quantidade de amostras, o que não é um problema, dado que o impacto desta na memória é pequeno, além do fato de ter sido usado o maior arquivo disponível para para se obter esses números.

Tabela 7: Resposta no uso da memória e na variação do tempo de execução para valores de **PARALLEL_POINTS**

PARALLEL_POINTS	Memória alocada no dispositivo (MB)	Tempo (s)
10	26,95	629,6594
50	27,7	377,0795
100	28,63	325,5127
1.000	45,46	279,8477
10.000	213,76	274,6500
50.000	961,74	273,4495

4.3.2 Quantidade de *Threads* por Blocos

A quantidade de *threads* por blocos afeta o sistema de diferentes formas, pois um bloco com muitos *threads* pode:

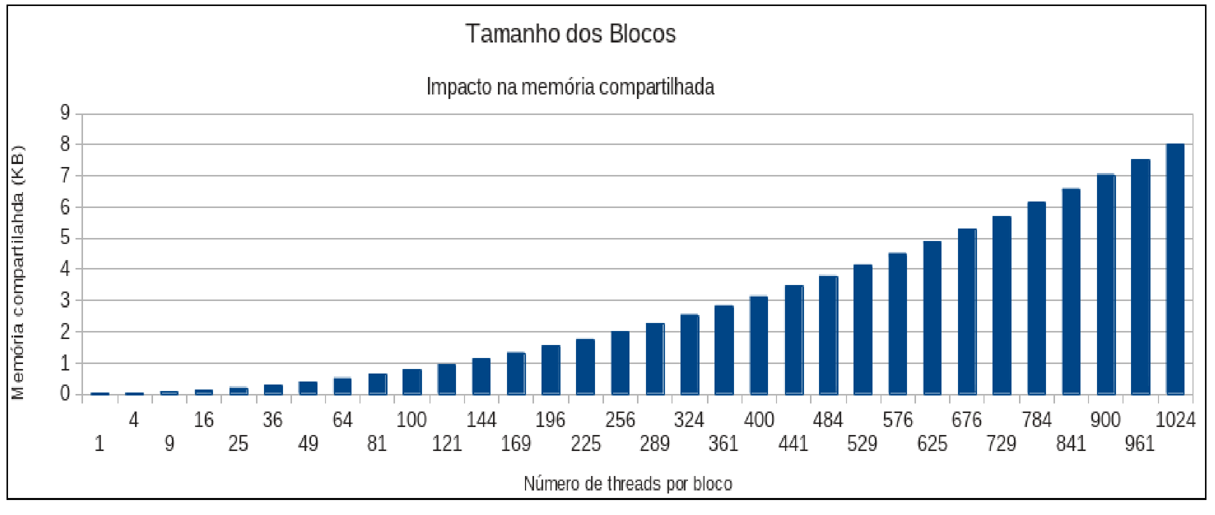
1. requerer mais memória compartilhada que o disponível;
2. fazer com que o *grid* tenha poucos blocos grandes, podendo deixar alguns SM ociosos caso o dispositivo tenha muitos;

3. aumentar o tempo gasto com comunicação entre os muitos *threads* do bloco.

Ainda assim é importante ter um bloco com um número razoável de *threads* para que se possa manter ocupados os *cores* do SM que executa o mesmo. Outro fator que deve ser levando em consideração é o tamanho do *warp*, pois, caso o número de *threads* não seja múltiplo do *warp* , o último *warp* terá um número pequeno de *threads*, podendo deixar alguns núcleos ociosos.

No sistema, a quantidade de *threads* por blocos varia com o quadrado da constante **TILE_WIDTH**. A quantidade de memória compartilhada alocada também é determinada por essa constante, pois cada *thread* aloca uma quantidade fixa dessa memória, assim, quanto mais *threads*, mais se aloca da memória compartilhada. A multiplicação de matrizes, que é a etapa mais custosa em relação a consumo de memória compartilhada, aloca duas matrizes de ponto flutuante de precisão simples com ordem **TILE_WIDTH²** elementos. Dessa forma, pode-se calcular a quantidade de memória compartilhada utilizada em função da quantidade de *threads* por bloco. O resultado desses cálculos estão plotados no Gráfico 5.

Gráfico 5: Alocação da memória compartilhada, de acordo com o número de threads.

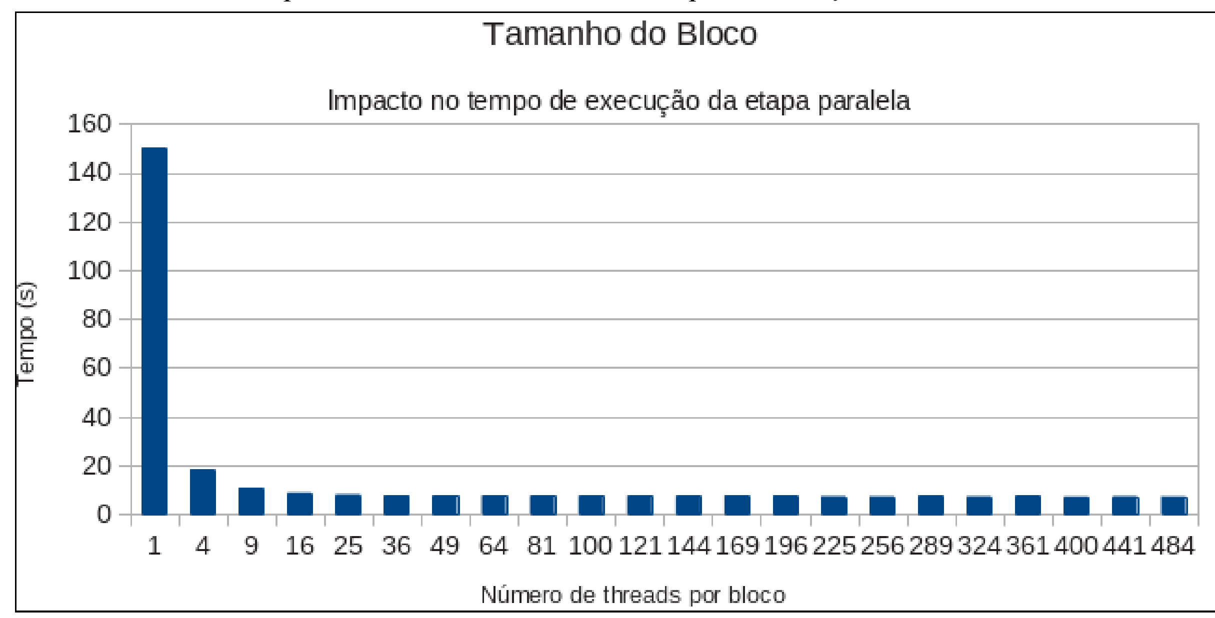


O uso da memória cresce com o número de *threads* por bloco, porém, como ilustra o Gráfico 5, não passa dos 8KB para 1024 cores, que é o maior número de *threads* possível na versão 2.0 da arquitetura.

O Gráfico 6 mostra a duração da etapa paralela do programa para cada uma das possíveis escolhas de número de threads, usando-se o arquivo de entrada com 700 amostras e

estimando-se 10.000 pontos na placa GeForce GT 240. É possível observar, através da análise do mesmo, que o tempo de execução da etapa se estabiliza com um número de threads superior a 16. Isso acontece pois o número de CUDA cores por SM em placas dessa geração é 8, portanto 16 *threads* – o dobro de núcleos – já os mantém ocupados.

Gráfico 6: Impacto do tamanho do bloco no tempo de execução na GeForce GT 240



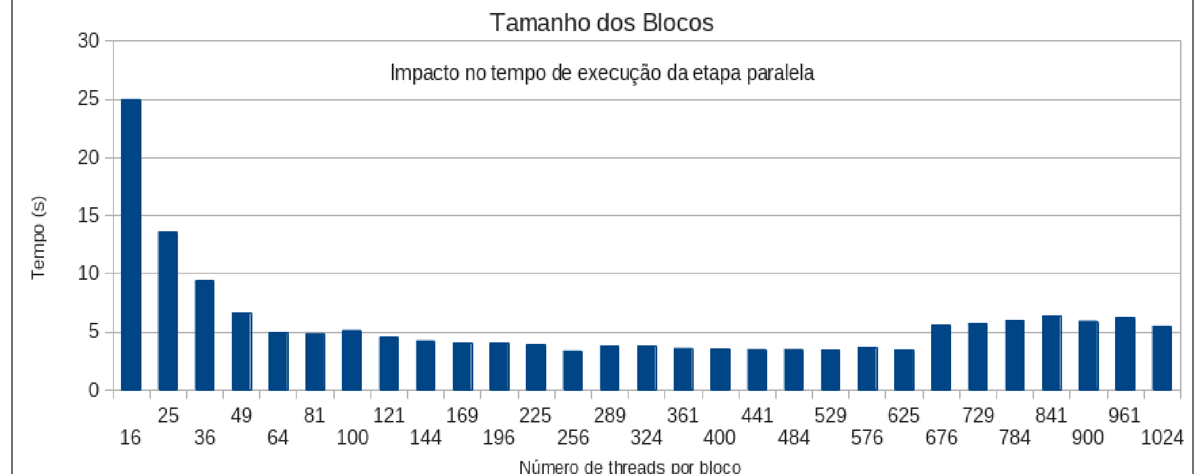
No intervalo de 16 a 484 *threads* o tempo de execução se estabilizou por volta dos 7s, porém os três últmos tiveram um resultado ligeiramente melhor, provavelmente, por causa da maior utilização da memória compartilhada. Desta forma, estabeleceu-se para a constante **TILE_WIDTH** o valor 22, que configura os blocos do programa com 484 *threads*, o maior valor possível para o programa em placas dessa versão arquitetura.

Ainda no Gráfico 6, pode-se observar o impacto sobre o desempenho do sistema uma escolha de valores baixos para a variável em questão. Com um *thread* por bloco – a pior configuração – o SM, que possui oito cores, terá sete deles ociosos enquanto o outro executa. Neste mesmo cenário, a memória compartilhada não será compartilhada com nenhum outro *thread* já que só existe um por bloco, o que significa que todos os acessos terão o mesmo efeito de acessos à memória global. Graças a isso, o tempo de execução para esse pior cenário foi cerca de 21 vezes mais lento que o melhor.

Uma análise similar foi realizada com a GeForce 310, e o resultado foi bastante parecido, algo esperado, já que essas placas são da mesma versão da arquitetura CUDA, portanto, respondem às configurações de maneira similar.

O Gráfico 7 é similar ao 6, porém com dados relativos à 412 amostras e estimação de 1 milhão de pontos na Tesla c2070.

Gráfico 7: Impacto do tamanho do bloco no tempo de execução na Tesla c2070



Esta placa, por ser da versão 2.0 da arquitetura, tem seus SMs agrupando 32 cores. Dessa forma, se um bloco possuir apenas um *thread* – o pior caso – deixa 31 cores ociosos, tendo um efeito muito mais negativo no tempo de execução do programa. Exatamente por isso, os três primeiros valores foram omitidos do gráfico para facilitar a leitura do mesmo, pois o pior caso demorou mil segundos para executar, sendo 33 vezes mais lento que a melhor configuração.

É possível observar que, para essa placa, o tempo de execução do programa só se estabilizou que depois o número de *threads* ultrapassou os 64, que é exatamente o dobro de CUDA cores por SM, assim como nos outros dispositivos, que possuíam 8 cores por SM e o programa se estabilizou após os 16 *threads* por blocos.

Ainda se analisando o Gráfico 7, é possível notar que, quando o número de *threads* ultrapassa 625, tempo de execução do trecho aumenta cerca de 60%. Isso pode acontecer devido a dois motivos principais: 1) o grande número de *threads* no bloco pode ter gerado uma sobrecarga na sincronização dos mesmos; 2) a arquitetura CUDA na versão 2.0 consegue manter até 16 blocos simultaneamente em um SM. Portanto, como cada bloco com mais de 625 threads usa pelo menos de 5KB de memória compartilhada (vide Gráfico 5), manter até

mesmo 10 deles simultaneamente ultrapassaria a quantidade de memória compartilhada disponível.

O resultado dos experimentos mostrou que o melhor valor de **TILE_WITDH** para a execução do programa na Tesla c2070 é 16, portanto, com blocos de 256 *threads*, que é múltiplo do número de *threads* por *warp* e usa apenas 2KB de memória compartilhada. Essa configuração garante um uso máximo de 32K dos 48K disponíveis na memória compartilahda, mesmo se mantendo 16 blocos por SM. A arquitetura dessa placa permite gerenciar *grids* muito maiores que nas placas de arquiteturas mais antigas (vide Tabela 1), portanto não há necessidade de preocupação com blocos relativamente pequenos, com apenas 256 *threads,* pois o *grid* pode comportar um grande número desses blocos.

4.4 Desempenho

Após determinadas as melhores configurações, resumidas na Tabela 8, foram realizadas comparações entre as execuções do programa, com o objetivo de verificar o ganho de desempenho obtido com a paralelização.

Primeiramente comparou-se a diferença de desempenho entre o programa serial e o paralelo, em seguida, do programa paralelo para diferentes números de *cores* disponíveis, avaliando-se assim a eficiência do programa.

Tabela 8: Configuração do programa para cada uma das placas

Placa	PARALLEL_POINTS	TILE_WIDTH
GeForce 310	1.000	22
GeForce GT 240	1.000	22
Tesla C2070	10.000	16

4.4.1 Comparação Entre a Versão Paralela e a Serial

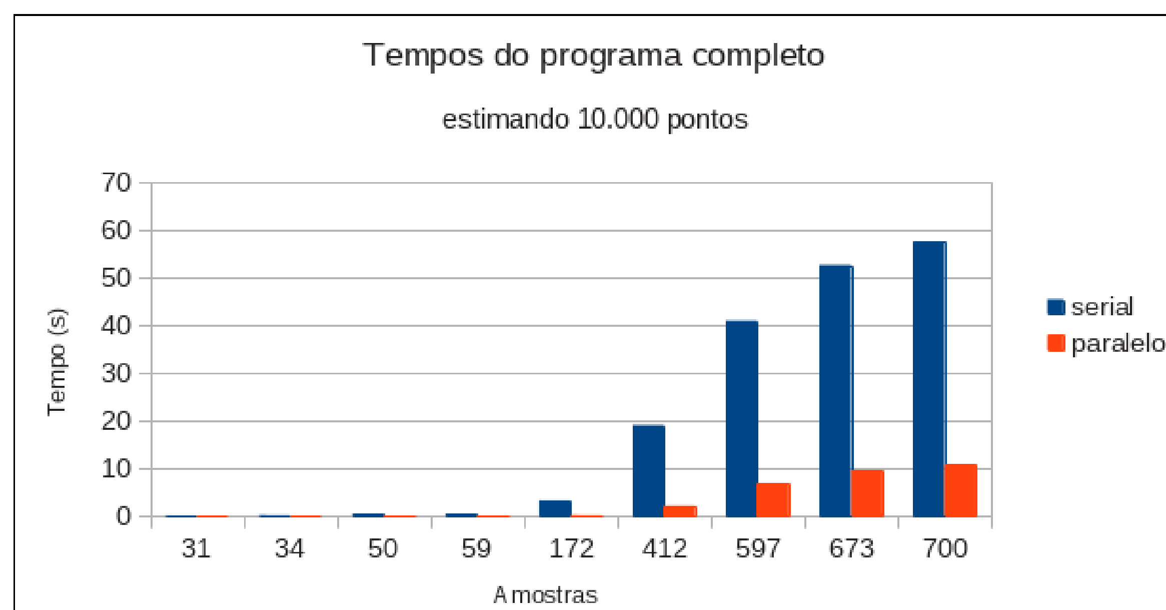
Primeiramente escolheu-se variar o número de pontos amostrados e deixar constante a quantidade de pontos estimada, no caso, em 10.000 pontos. A Tabela 9 apresenta os tempos apenas para a etapa do programa que foi paralelizada, desconsiderando todo o restante. A comparação é feita entre um Intel Core 2 Quad e a Nvidia Tesla C2070. Pode-se observar que o programa paralelo é sempre mais rápido que o serial e o tempo de execução do mesmo aumenta bem menos para acréscimos no número de amostras.

Tabela 9: Tempo da etapa de estimação, variando-se apenas o número de amostras.

Amostras	Tempo Serial	Tempo Paralelo
31	0,143401	0,003077
34	0,165686	0,002548
50	0,367005	0,003989
59	0,435176	0,005187
172	3,244046	0,075207
412	17,806857	0,946374
597	37,125209	2,935469
673	47,083245	4,205198
700	51,421919	4,668532

O Gráfico 8 mostra o tempo de execução do programa completo (excluindo-se E/S).

Gráfico 8: Tempo de execução do programa completo variando-se apenas o número de amostras.



É importante também comparar os tempos de execução variando-se o número dos pontos estimados, pois a etapa do programa que foi paralelizada está diretamente relacionada com a estimação dos pontos. Essa comparação pode ser observada na Tabela 10, que mostra os tempos para execução do programa completo, tanto da versão paralela como da serial, para 700 pontos amostrados.

Tabela 10: Tempos de execução do programa, versões paralela e serial para 700 amostras

Estimações	Paralelo	Serial
25	10,86 s	6,13 s
100	10,78 s	6,40 s
1.024	10,83 s	9,60 s
10.000	10,83 s	57,47 s
102.400	11,5 s	6 min e 2 s
1.000000	17,06 s	58 min e 3 s
10.004.569	73,84 s	14 h 7 min e 46 s

É possível observar que, para os três primeiros valores obtidos, o programa paralelo demorou mais para chegar ao resultado que o serial. Isso é explicado pelo fato do número de pontos estimados ser muito pequeno, dessa forma, o tempo necessário para configurar o dispositivo não é compensado pelo pouco cômputo que é realizado no mesmo. Além disso, a pequena quantidade de pontos não permite exploração do hardware disponível, deixando alguns núcleos ociosos.

O programa paralelo demora praticamente o mesmo tempo para calcular 25 ou 10.000 pontos, cerca de 11 segundos, enquanto o serial já apresenta um grande aumento, demorando pouco mais de seis segundos para calcular 25 e praticamente um minuto para calcular os 10.000.

A partir de 10.000 pontos nota-se uma grande diferença no tempo de execução dos programas.

O programa paralelo é significativamente mais rápido que o serial, porém ainda é preciso saber se o mesmo escala bem de acordo com a quantidade de unidades de processamento disponíveis.

4.4.2 Eficiência no Uso do Hardware Paralelo

Para avaliar o quão eficientemente um software usa os recursos paralelos, mede-se como o tempo de execução do programa se comporta com o aumento de número de processadores. Algumas arquiteturas paralelas permitem que essa avaliação possa ser bem detalhada, à medida que permitem adicionar ao sistema, por exemplo, uma unidade de processamento por teste. Na arquitetura CUDA não temos esse controle para aumentar o número de núcleos utilizados, já que o programa é escrito para tirar o máximo proveito possível de todos os núcleos disponíveis.

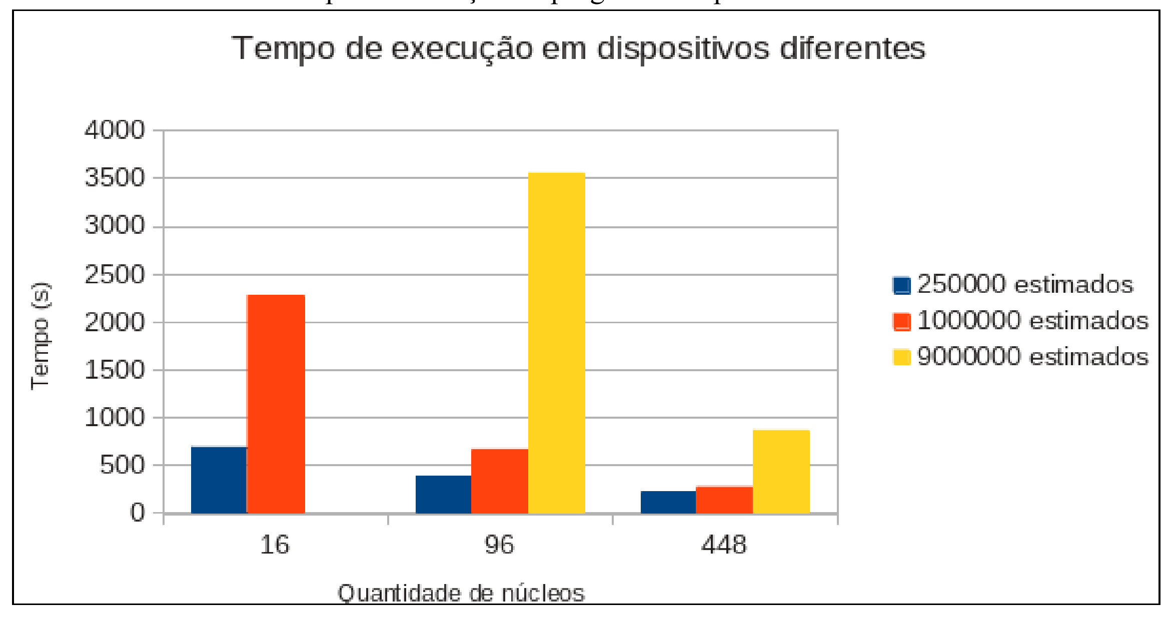
Dessa forma podemos executar o programa em placas de vídeo diferentes, que tenham um número diferente núcleos. Embora seja possível executar o programa variando-se o

número de núcleos, essa variação vai depender das placas disponíveis para teste, e nunca vai ser unitária, já que os SM's têm no mínimo oito núcleos, a depender da versão da arquitetura.

Outro fator relevante é o fato de esses dispositivos terem características arquiteturais diferentes, como memória ou núcleos com frequências diferentes umas das outras, maior largura de banda do barramento ou variação no número de núcleos no SM por exemplo.

O Gráfico 9 apresenta três conjuntos de barras, que representam o tempo de execução do programa nas três placas disponíveis para o experimento, indicando em seu eixo da abscissas o número de núcleos de cada uma delas. As medidas de tempo são feitas para execuções do programa com entrada de 2450 amostras para, respectivamente, 250 mil, 1 milhão e 9 milhões de pontos calculados. O cálculo de 9 milhões de pontos com apenas 16 núcleos demorou quase 20.000 segundos, portanto foi omitido para facilitar a leitura do gráfico.

Gráfico 9: Tempo de execução do programa em placas de vídeo diferentes



Observa-se no gráfico que, para cada valor de pontos estimados, o tempo de execução do programa é menor à medida que se aumenta o número de núcleos, como esperado. Porém ainda é necessário analisar se esse aumento é proporcional ao acréscimo do número de núcleos, informação constante na Tabela 11, que mostra os valores para a estimação de nove milhões de pontos.

Tabela 11: Aumento relativo de núcleos e aceleração

Quantidade de núcleos	Quantidade relativa de núcleos	Aceleração
16	1 vez – valor base	1x – valor base
96	6 vezes	5,39 vezes
448	28 vezes	22,06 vezes

A Tabela 12 apresenta o valor da eficiência do programa para os tempos do Gráfico 9. É notável que a eficiência do programa é maior com o aumento do cômputo, que é um comportamento interessante, já que para problemas reais é comum se ter grande quantidade de dados para ser computado.

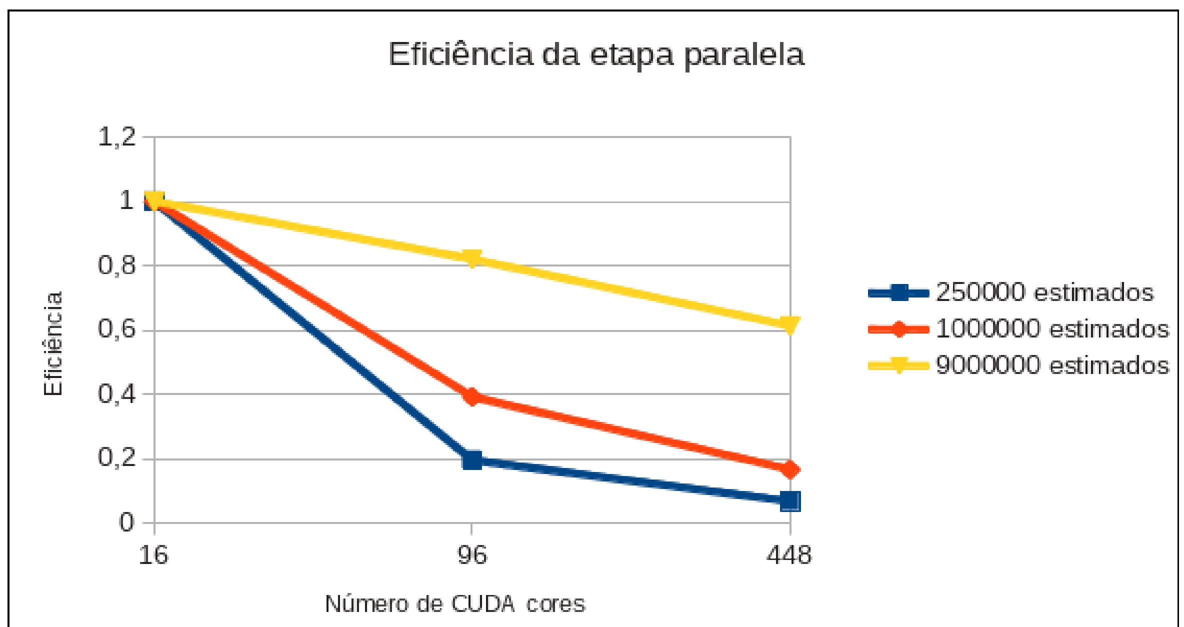
Tabela 12: Eficiência da etapa paralela

Pontos estimados	Número de núcleos		
	16	96	448
250 mil	1	0,1955	0,0670
1 milhão	1	0,3920	0,1659
9 milhões	1	0,8203	0,6128

No melhor resultado, a eficiência permaneceu sempre bem próxima do ideal teórico, valor 1. É necessário também fazer uma análise similar no programa como um todo, porém o cálculo pode não refletir os valores reais, pois a etapa serial do código é executada em processadores diferentes e, portanto, a variação do tempo não depende apenas do quão eficiente o programa como um todo utiliza o hardware paralelo, pois o processador mais eficiente vai contribuir para a redução do tempo de execução. Com essas ressalvas, o Gráfico 10 mostra a eficiência para o programa completo.

Como pode ser observado, na comparação da Tabela 12 com o Gráfico 10, a eficiência diminui um pouco, porém a configuração ainda é muito parecida com o anterior, onde a eficiência ainda é boa para grandes volumes de cômputo – aproveitamento de mais de 60% para 28 vezes mais núcleos disponíveis.

Gráfico 10: Eficiência do programa como um todo.



5 CONCLUSÃO

Esta monografia descreve uma implementação serial do algoritmo de krigagem e a paralelização da mesma para a arquitetura CUDA, visando obter ganho de desempenho aproveitando os recursos computacionais de GPU da Nvidia.

Os testes realizados com o algoritmo desenvolvido mostraram que o programa paralelo é suficientemente preciso, já que a saída gerada pelo mesmo apresenta valores muito próximos aos obtidos pelo programa serial, com erro máximo da ordem de 10^{-8}%. Uma pequena diferença entre os resultados já era prevista, já que a precisão finita de ponto flutuante utilizada está sujeita a acúmulos de pequenos erros, que podem trazer resultados diferentes quando operações aritméticas são realizadas em ordem diferente num código em relação a outro – como é o caso de um código serial e um paralelo - ou executa em arquiteturas diferentes – como por exemplo em Intel x86 ou Nvidia CUDA.

Através dos testes, pôde-se concluir que o programa paralelo precisa estar adequadamente configurado para extrair o máximo de desempenho da GPU, cabendo configurações diferentes para diferentes hardwares com diferentes capacidade de processamento. Configurações inapropriadas, como mostrado nos testes de configuração, podem fazer com que o programa utilize muita memória do dispositivo gráfico ou mesmo não aproveite o poder processamento disponível, podendo resultar, respectivamente, em finalização do mesmo por falta de memória, ou grande aumento no tempo de execução por mal aproveitamento do hardware.

Os testes também mostraram que os temos de execução do programa na arquitetura CUDA aumentam menos quando se aumenta o número de pontos interpolados, em comparação com a execução na arquitetura x86. Para mil pontos, o tempo de execução dos dois programas nos testes realizados é de aproximadamente 10s, para 10 mil pontos não há alteração no tempo de execução do paralelo, enquanto o serial demora cerca de um minuto para executar. A medida que se aumenta o número de pontos interpolados a diferença cresce. O maior teste realizado, estimando-se 10 milhões de pontos o programa paralelo demorou 74s enquanto o serial 14h. Isso posto, pode-se afirmar que o esforço desprendido para explorar os recursos de processamento da GPGPU foi compensado com a aceleração da execução do programa.

Na arquitetura CUDA o programa escalou bem com o aumento de processadores, porém, para atingir índices mais próximos aos teóricos, é necessário se estimar pelo menos 10 milhões de pontos. Para números inferiores a 250 mil a eficiência do programa cai para valores próximos a 10% enquanto para valores altos fica acima dos 80% , nos testes realizados.

Com base nos resultados obtidos é possível concluir que o algoritmo da krigagem se adéqua bem a arquitetura CUDA. Ainda assim, é válido salientar que apenas a etapa de interpolação dos pontos foi paralelizada no algoritmo. Essa etapa foi escolhida por tratar-se da mais custosa computacionalmente, porém existem outras etapas que podem ser paralelizadas em trabalhos futuros, como o cálculo do determinante da matriz de covariâncias e a inversão da mesma.

O algoritmo paralelo também poderá ser aperfeiçoado para poder detectar automaticamente a melhor configuração para explorar o processamento disponível na GPU sem exceder a memória disponível no chip, pois atualmente são necessários vários testes para determinar uma configuração ideal. Para cada um desses testes o programador deve compilar o código, já que esses parâmetros são definidos através de constantes, e executar o programa para avaliar a resposta às configurações escolhidas no tempo de execução do mesmo.

REFERÊNCIAS

CASTELO BRANCO, F. *Desenvolvimento do Algoritmo de krigagem em um Sistema para Gerenciamento de Reservatórios de Petróleo.* 2010. 58 f. Monografia (Graduação em Engenharia de Computação) - Universidade Estadual de Feira de Santana, Feira de Santana, 2010.

CHENG T., LI D., WANG Q. *On Parallelizing Universal Kriging Interpolation Based on OpenMP* Distributed Computing and Applications to Business Engineering and Science (DCABES) 2010, fl. 36-39

GUVENDIK, C.; GENC, A.E.; TAMER, O.; NIL, M. *Improving the performance of Kriging based interpolation application with parallel processors* Signal Processing and Communications Applications Conference (SIU) 2012, fl. 1-4

SANDERS, J.; KANDROT, E. *CUDA by Example:* An Introduction to General-Purpose GPU Programming. Upper Saddle River, NJ: Addison-Wesley, 2010.

GUERRA, P.A.G. *Geoestatística Operacional.* Brasília: Departamento Nacional da Produção Mineral. 1988.

CAMARGO, E.C.G. et al. *Integração de Geoestatística e Sistemas de Informação Geográfica:* Uma necessidade. São José dos Campos. 2012. Disponível em: http://www.dpi.inpe.br/geopro/trabalhos/gisbrasil99/geoest_gis/. Acessado em: 08 ago. 2012.

PACHECO, P. S. *An Introduction to Parallel Programming.* Burlington, MA, USA. Morgan KAUFMANN, 2011.

AMDAHL G.M., *Validity of the Single-Processor Approachto Achieving Large-Scale Computing Capabilities* Federation of Information Processing Societies Conf., AFIPS Press 1967, fl. 483-485

FLYNN, M. J. *Some Computer Organizations and Their Effectiveness* Department of Computer Science, The Johns Hopkins University, Baltimore1972

KIRK, D. B.; HWU, W. *Programming Massively Parallel Processors:* A Hands-on Approach. Burlington, MA. Morgan Kaufmann Publishers, 2010.

TSUCHIYAMA R.; NAKAMURA T.; LIZUKA T.; ASAHARA A.; SON J.; MIKI S. *The OpenCL Programming Book.* Fixstars, 2012

NVIDIA. *CUDA Programming Guide v4.2.* 2012. Disponível em: http://www.nvidia.com/content/cuda/cuda-documentation.html. Acessado em: Ago. de 2012.

KAMRAN K., NEIL G. D., FIRAS H. *A Performance Comparison of CUDA and OpenCL.* British Columbia, Canada. 2011.

WHITEHEAD, N; FIT-FLOREA, A. *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs.* 2011. Disponível em: https://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus. Acessado em: dez. de 2012