ANÁLISE COMPARATIVA DA IMPLEMENTAÇÃO DE BEAMFORMING UTILIZANDO PROGRAMAÇÃO CONVENCIONAL E PARALELA

Fernando de Souza Pereira Monteiro*, Fabricio de Abreu Bozzi*, Fábio Oliveira Baptista da Silva*

*Rua Ipiru, n 02,Cacuia, Ilha do Governador, Rio de Janeiro

Emails: fernando.monteiro@marinha.mil.br, bozzi@ipqm.mar.mil.br, fabio.oliveira@marinha.mil.br

Abstract— This study aims to analyze the implementation of the beamforming algorithm using conventional programming, primitives and parallel programming. The time-domain delay-and-sum algorithm, known as conventional beamforming, is used with typical sonar configurations. In this work, conventional programming refers to implementation in C / C ++ languages, being executed with its own operators. Intel's primitives are analyzed as they promise optimization in signal processing. Finally, the implementation via GPU is compared using CUDA.

Keywords— Sonar, Beamforming, Computer Programing, CUDA.

Resumo— Este estudo visa analisar a implementação do algoritmo de formação de feixes utilizando programação convencional, primitivas e programação paralela. O algoritmo atraso e soma no domínio do tempo, conhecido como *beamforming* convencional, é utilizado com configurações típicas de sonares. Neste trabalho, a programação convencional se refere a implementação nas linguagens C/C++, sendo executada com seus operadores próprios. As primitivas da Intel são analisadas, já que prometem otimização em processamentos de sinais. Por último, a implementação via GPU é comparada, utilizando CUDA.

Palavras-chave— Sonar, Conformação de feixes, Programação de Computadores, CUDA

1 Introdução

O desenvolvimento de sistemas de sonares, quando referindo-se a parte seca, sempre exigiu otimizações por lidar com um volume grande de sinais a serem processados. Com o avanço da eletrônica e a acessibilidade a computadores de alto desempenho, os antigos sistemas sonares, volumosos e com pouco poder de processamento, vem sendo substituídos por hardwares pequenos e computadores que podem executar de maneira distribuída e otimizada o processamento dos sinais.

Apesar do aumento de capacidade de memória e processadores disponíveis, o desejável é sempre otimizar o processamento de sinais, gerando "economia" de processamento, o que possibilita que diversas análises sejam feitas simultaneamente. Neste sentido, algumas linguagens de programação, como o C e o C++, tendem a serem rápidas ao realizar cálculos matemáticos, apesar de operarem sequencialmente em um elemento [1].

Uma opção de incremento do processamento é dado por chamadas de primitivas da CPU, como a opção dada pela fabricante Intel. A *Intel Integrated Performance Primitives* (Intel IPP) é uma arquitetura de software que otimiza as chamadas ao processador [2] [3], diminuindo assim o tempo gasto ao realizar cálculos. Outra escolha que pode ser adotada é a utilização das placas gráficas para realização do processamento. Neste caso, devido a grande quantidade de processadores, pode, dentro de certos limites ser vantajoso [4] [5].

Desta forma, o estudo em lide, compara uma aplicação de processamento de sinais utilizada em sonares, implementada em C/C++, IPP e CUDA.

2 Formador de Feixes Atraso e Soma

A formação de feixes, *Beamforming* em inglês, é a parte central do processamento dos sinais de arranjos de sensores. Sendo considerado um filtro espacial, apresenta a capacidade de focar a recepção em uma direção específica. A técnica convencional de *beamforming* é o filtro de atraso-e-soma (*delay-and-sum*) [6]. Esta técnica se baseia no conceito de interferência construtiva e destrutiva de ondas. O ângulo cujo atraso maximiza a energia na saída do filtro corresponde ao ângulo de incidência da frente de onda.

O processo de implementação do *delay-and-sum* no arranjo circular é feito de acordo com [7] e explicado com auxilio da Figura 1. Esta figura ilustra um arranjo com geometria circular, e uma frente de onda chegando de determinada direção. Utiliza-se um setor do arranjo contendo um número específico de elementos (K) para formar o feixe da direção referente à frente de onda (no caso ilustrado, utilizam-se

os elementos 1 a 5 e 28 a 32)[1]. Com esta seção escolhida, aplicam-se os atrasos nos elementos de forma a compensar os diferentes percursos da frente de onda até um elemento de referência.

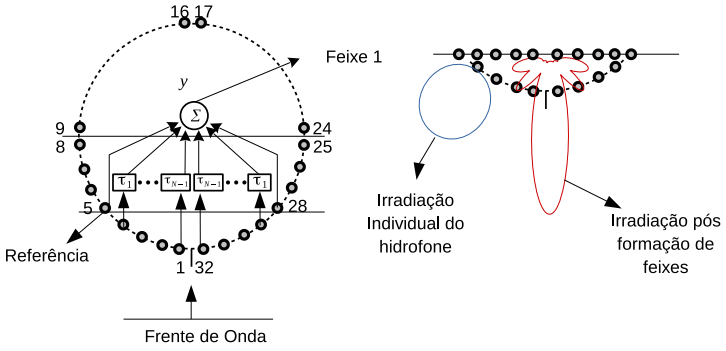

Figura 1: Diagrama de implementação do *beamforming* para o arranjo circular.

O procedimento de atrasar τ_n é uma forma de "sincronizar"os sinais, para fazer com que o arranjo circular seja considerado um arranjo em linha desigualmente espaçado. Após aplicados os respectivos atrasos, somam-se os sinais deste setor, o que resulta em um feixe referente à direção *broadside*, perpendicular ao arranjo em linha equivalente.

Na Figura 1, ressalta-se a representação da irradiação ao utilizar um sinal individual de um sensor e o diagrama de irradiação após a formação de feixes, onde percebe-se que existe um ganho direcional.

Este procedimento é repetido utilizando-se elementos adjacentes, assim, obtêm-se os feixes formados. Devido à geometria circular do arranjo, os atrasos são fixos, mudando-se apenas os elementos a serem utilizados na projeção.

3 Métodos de Programação

Este trabalho utilizará de cinco computadores distintos, com processadores e placas gráficas de diferentes modelos. Todos computadores testados possuem processadores Intel, já que as primitivas são otimizadas para estes processadores. A comparação dos tempos de execução se dará entre C/C++, IPP e CUDA, e uma análise CUDA *vs* CUDA, variando-se os blocos de paralelização também é realizada.

Como a linguagem C/C++ pode ser relacionada ao poder computacional de uma máquina, uma avaliação dos tempos normalizados pelo tempo C/C++ é feito.

3.1 Programação em C/C++

As linguagens de programação C/C++, quando utilizadas de forma otimizadas, cumprem tarefas de processamento com alto desempenho [1]. No entanto, sem o emprego de bibliotecas externas, se restringem ao uso apenas de suas expressões e operadores padrão da linguagem.

Realizando cálculos a partir da indexação de *arrays*, laços *for* e operadores aritméticos, que operam apenas sobre um elemento por vez, acabam por limitar o desempenho no processamento. A capacidade de otimização do compilador também é limitada a acelerar a indexação e o acesso dos elementos das matrizes de entrada e saída, o que contribui para se buscar outras formas de otimização quando lidamos com uma quantidade elevada de sinais para processar.

3.2 Programação Utilizando Primitivas da Intel

A biblioteca *Intel Integrated Performance Primitives* (Intel IPP) é desenvolvida e disponibilizada pela Intel com funções básicas para emprego em processamento de dados, imagens e sinais. Essa biblioteca utiliza conjuntos de instruções específicos das *CPUs (Central Processing Unit)* Intel resultando em um desempenho maior do que o alcançado pelo compilador quando utilizado o conjunto de instruções x86.

Incluindo instruções *SIMD (Single Instruction, Multiple Data)*, aplica operações, de forma simultânea, em vários elementos de um conjunto, por exemplo um vetor [8]. A instrução,

```
IppStatus ippsAdd_32f(const Ipp32f* pSrc1, const Ipp32f* pSrc2, Ipp32f* pDst, int len);
```

é um exemplo que realiza a soma de *len* elementos dos vetores apontados por *pSrc1* e *pSrc2* e escreve o resultado em *pDst* [9].

[1]Para este arco dizemos que o feixe é centrado aos canais 1 e 32, assumindo a simetria par.

Pela otimização e simplicidade no uso de suas funções de processamento, as primitivas da Intel são utilizadas em diversas aplicações que exigem uma melhor *performance* de processamento [2] [3].

3.3 Programação Paralela Utilizando CUDA

CUDA (*Compute Unified Device Architecture*) é uma plataforma de computação paralela e utilizada da *API (Application Programming Interface)* criada pela NVidia que fornece uma camada de software que permite executar códigos na *GPU (Graphics Processing Unit)*. As *GPU*, tipicamente possuem centenas a milhares de núcleos de processamento, enquanto as CPU's possuem tipicamente menos de uma dezena.

Atualmente, a NVidia disponibiliza linhas de placas gráficas com objetivos distintos. Elas são compostas pela: GeForce, que é considerada uma linha básica, a linha *Quadro*, especializada em processamento gráfico, e por último a linha *Tesla*, otimizada para processamento [10].

As funções executadas em CUDA são chamadas de *kernels*. Cada 8 núcleos de CUDA, ou *cuda cores*, são agrupados em uma *SM (Stream Multiprocessor)*. O *SM* cria, gerencia, escalona e executa os *kernels* em grupos de 32 *threads*, chamadas de *warps*. Cada *kernel* possui um tamanho de bloco, *cuda block*, de até três dimensões, cujos limites são de até 1024 para x e y e 512 para z. o produto $x * y * z$ é também limitado em 1024 [11].

Apesar de sua implementação ser considerada mais complexa, comparada a IPP e C/C++, sua execução basicamente consiste em: Alocação de memoria na placa de video, cópia dos dados da memória da CPU para a GPU, execução do *kernel* utilizando o padrão de setas <<<>>>>, sincronização das *threads* e por fim, realiza a cópia para a CPU.

4 Método de Comparação entre Implementações

O algoritmo de *beamforming*, implementado em C/C++, IPP e CUDA foi testado em cinco computadores, dentre *Desktops* e *Laptops*, e um computador possuindo duas Placa Gráficas da linha Tesla. As configurações dos computadores são apresentadas nas tabelas 1 e 2.

Os parâmetros de números de canais e amostras, que são utilizados ao formar os feixes, foram variados de forma a avaliar os melhores resultados. Desta forma, os testes alternavam os blocos de dados com 16, 32, 64 e 128 canais e 128, 256, 512, 1024, 2048, 4096, 8192 e 16384 amostras, medindo o tempo de execução.

Para os testes usando CUDA, foram utilizados *cuda blocks* bidimensionais cujo tamanho foi variado em 11 passos de potências de dois mantendo o produto $x * y = 1024$, conforme descrito na Equação 1. A configuração de *cuda blocks* que resultou no menor tempo de processamento foi utilizada nas análises subsequentes.

$$cuda\ blocks(x, y) = (2^k, 2^{10-k}), k \in \mathbb{N}, 0 \leq k \leq 10 \tag{1}$$

Cada configuração foi executada 100 vezes com objetivo de enriquecer a estatística das análises, Além disso, os testes foram realizados sem que outros processos estivessem em execução, de modo a não sofrer interferência dos mesmos.

Tabela 1: Computadores utilizados para execução dos testes.

Computador	C1 (Laptop)	C2 (Desktop)	C3 (Desktop)	C4 (Laptop)
Processador	i7-7700HQ	i7-8700K	i7-4790	i7-5500U
Núcleos	2.80 GHz x 4	3.70 GHz x 6	3.60 GHz x 4	2.40 GHz x2
Lançamento	Q1 2017	Q4 2017	Q2 2014	Q1 2015
Placa Gráfica Geforce	GTX 1050 Ti	GTX 1070 Ti	GTX 750	830M
Núcleos CUDA	768	1920	512	256
Memória (MB)	4096	8192	2048	2048

Como citado, os testes também foram realizados em um servidor com duas placas gráficas dedicadas para processamento de dados utilizando a linha Nvidia Tesla. Nesta máquina não foi possível realizar os testes com a biblioteca IPP.

Tabela 2: Computador com NVidia linha Tesla, utilizado para execução dos testes.

Computador	C5a	C5b
Processador	i7-4790K	
Núcleos	4.00 GHz x 4	
Lançamento	Q2 2014	
Placa Gráfica Tesla	K40	C2075
Núcleos CUDA	2880	448
Memória (GB)	12	6

5 Resultados

Os resultados apresentados nesta seção baseiam-se na análise do tempo de processamento entre as implementações nos diferentes computadores, ao variar os parâmetros utilizados pela placa gráfica ou da característica do arranjo/*beamforming*.

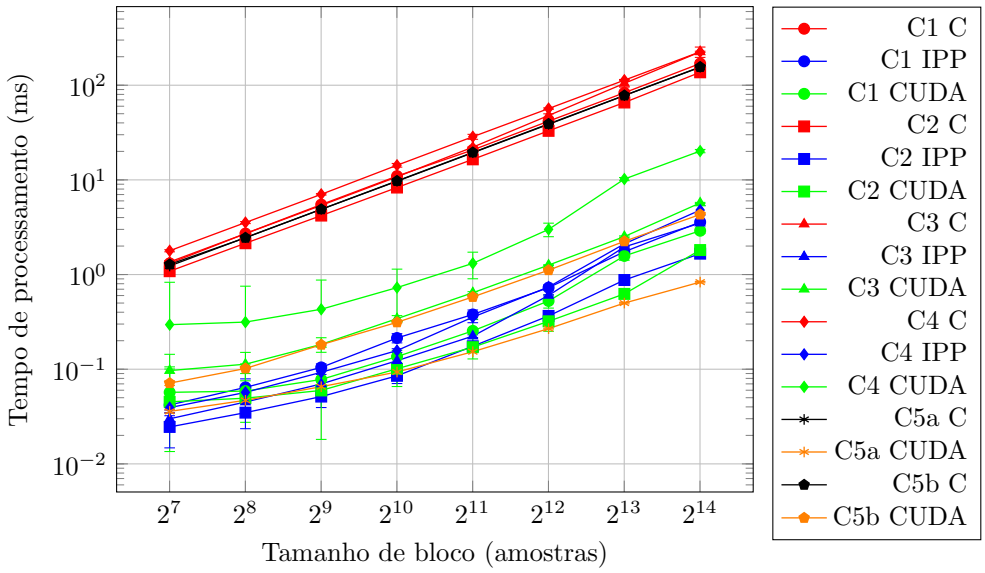

Figura 2: Tempo de Processamento (ms) para 32 Canais

A figura 2 apresenta o tempo de processamento para cada um dos computadores utilizando a configuração de um arranjo de 32 canais. O número de amostras por bloco foi incrementado de 128 a 16384 amostras. Observa se que o tempo de processamento em C é aproximadamente linear com o aumento do número de amostras. Tanto IPP quanto CUDA apresentam um desempenho de cerca de 10 a 100 vezes maior que C.

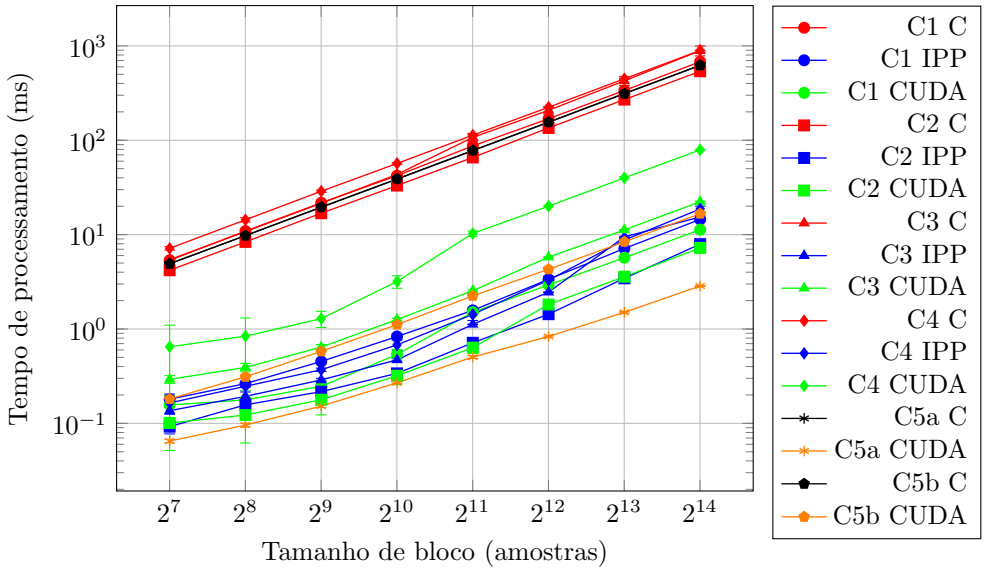

Figura 3: Tempo de Processamento (ms) para 128 Canais

A figura 3 apresenta o resultado ao aumentar o número de canais para 128. Nesta configuração, com 4 vezes mais dados, nota-se o aumento do tempo em todos os resultados.

Vale ressaltar nas figuras 2 e 3 o resultado do computador *C4* ao implementar CUDA. Este computador possui uma placa de video considerada obsoleta e o resultado da implementação utilizando IPP é cerca de 10 vezes melhor neste computador.

Por outro lado *C5a* utilizando CUDA obteve o melhor desempenho todas as condições para 128 e para mais de 2048 amostras e 32 canais.

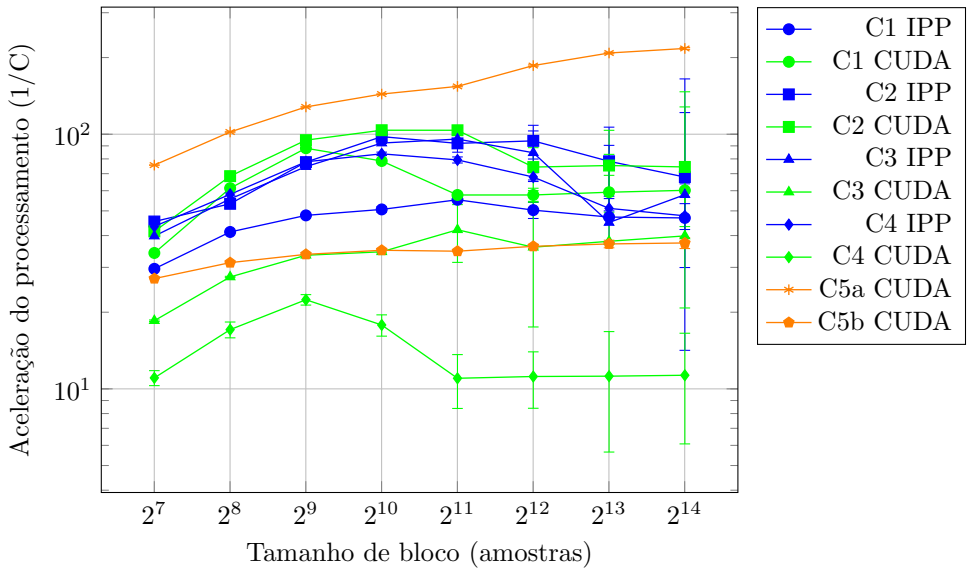

Figura 4: Aceleração do tempo de processamento utilizando 128 canais

A figura 4 apresenta o resultado dos tempos de processamento normalizados pelos tempos em C. Esta relação mostra a aceleração efetiva causada apenas pela implementação utilizando IPP e CUDA, supondo que o poder de processamento de cada computador pode ser relacionado ao C.

Pode-se observar na figura 4 que a aceleração do processamento de *C4* utilizando IPP é cerca de 5 vezes mais rápido do que ao utilizar CUDA. Esta análise, assim como apresentado na figura 3, indica efetivamente o benefício do IPP em *C4*.

Para o caso de *C5*, mostra-se o benefício da maior quantidade de *cuda cores* entre as placas gráficas. Já para o computador *C2*, pode-se inferir que, mesmo com uma placa gráfica de última geração (*GTX 1070Ti*), a implemetação da IPP com o processador de 8^a Geração traz tempos próximos ao da implementação em CUDA.

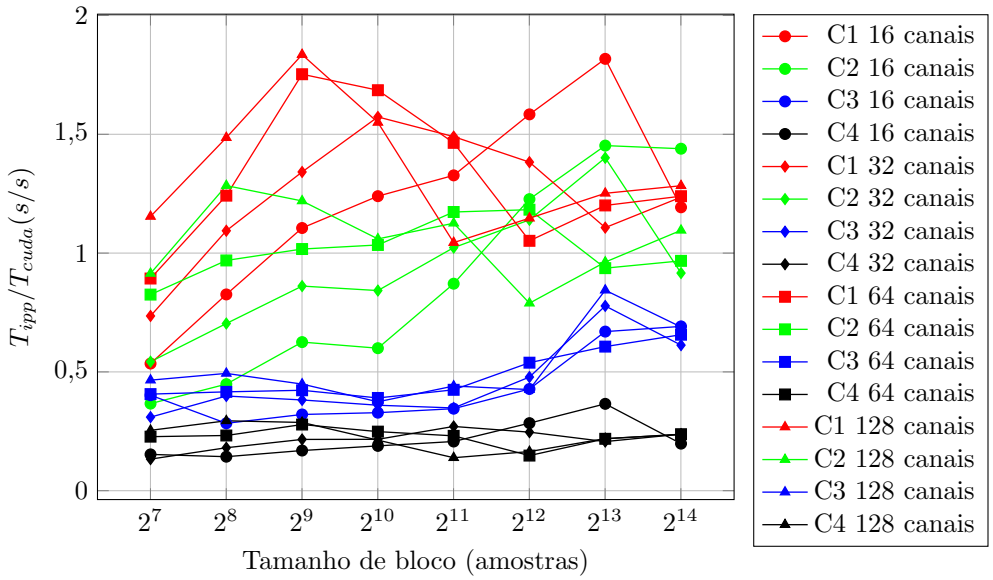

Figura 5: Razão entre tempos de processamento T_{ipp}/T_{cuda}

A figura 5 apresenta a comparação de desempenho entre CUDA e IPP para as maquinas *C1* a *C4*. O eixo y do gráfico apresenta a razão entre os tempos de processamento obtidos em IPP e CUDA. Valores maiores que 1 indicam que o desempenho de CUDA é superior ao desempenho utilizando IPP.

Observa-se que os computadores *C3* e *C4* possuem melhor desempenho ao utilizar IPP em todas as condições de teste, isso provavelmente está relacionado a quantidade de *cuda cores* das placas gráficas, o que resulta em uma menor capacidade de processamento.

A figura 5 mostra que *C2* é um computador em que IPP e CUDA produzem tempos semelhantes de processamento. Este resultado é condizente ao apresentado na figura 4, mostrando que apesar da alta capacidade de processamento da placa gráfica, a implementação da IPP com aquele processador é bem otimizada.

Para o caso de *C1*, nota-se que a implementação em CUDA se beneficia da placa gráfica, o que já não ocorre com a IPP e o processador de 7^a Geração com 4 núcleos. No entanto, em algumas situações, como $\{1024, 2048\}$ amostras e $\{128, 64\}$ canais notamos que o desempenho é semelhante.

A figura 6 apresenta o resultado do tempo de processamento do CUDA em função da variação do *cuda block*. Neste teste foi utilizado um bloco de processamento de processamento de 4096 amostras e 128 canais. O eixo x do gráfico representa a dimensão x do *cuda block*.

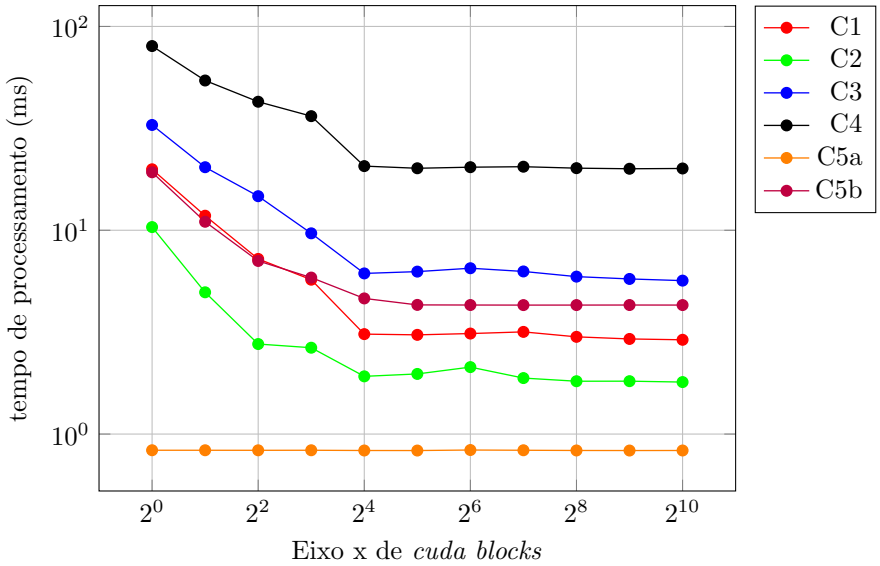

Figura 6: Tempos de processamento variando em função da variação de *cuda blocks*

Através da figura 6 observa-se que os computadores *C1* a *C4* e *C5b* apresentam um comportamento

decrescente, se estabilizando apenas a partir de (16,64). O computador *C5a* apresenta um tempo de processamento inferior e aproximadamente constante independente do tamanho de bloco, isso se deve a arquitetura otimizada e maior número de *cuda cores*.

6 Conclusão

Este estudo apresentou uma análise comparativa da implementação do algoritmo de formação de feixes utilizando linguagens de programação convencional, primitivas e programação paralela. O *beamforming*, que é um processamento espacial para arranjo de sensores, tem extensa aplicação em sistemas acústicos submarinos, como o sonar. Os parâmetros do *beamforming* foram variados de forma a analisar, caso a caso, qual implementação trouxe menor tempo de processamento. Assim, este trabalho estabeleceu um método de avaliação, que pode ser aplicado a diversos algorítimos, servindo como critério de escolha da implementação a ser feita.

Uma das análises realizadas mostrou que a utilização da linguagem C tem sempre pior desempenho, quando comparada com IPP e CUDA. Notou-se ainda que o tempo de processamento é aproximadamente linear com o tamanho de bloco em C. As implementações em IPP e CUDA apresentaram comportamentos não lineares em algumas configurações testadas.

A análise entre as implementações da IPP e CUDA mostraram que o estudo caso a caso deverá ser realizado para cada máquina e cada tamanho de bloco de processamento. Com a utilização de parâmetros comumente aplicados em arranjos sonar, pôde-se concluir que não existe uma implementação objetivamente melhor que outra, podendo variar, por exemplo, com o emprego de blocos de processamento de tamanhos diferentes.

O *beamforming*, por ser um algoritmo matricial altamente paralelizável, teria um melhor desempenho ao ser implementado em CUDA, no entanto, este estudo mostrou que as primitivas da Intel e os processadores atuais trazem uma otimização interessante. Além disso, CUDA mostrou variações de desempenho ao não otimizar condições de contorno, como por exemplo o número de blocos.

De modo geral, o CUDA atingiu os menores tempos de processamento, no entanto, o desenvolvimento em CUDA deve ser avaliado no sentido de custo, ao adquirir uma placa gráfica de alto desempenho, e no sentido portabilidade e tempo de implementação, dado que as otimizações devem ser feitas caso a caso.

Por último, pode-se avaliar o benefício da implementação utilizando o CUDA no sentido de "liberação" de espaço do processador para atender outras tarefas. Como citado neste trabalho, as avaliações foram realizadas sem outros processos ativos, de forma que a implementação utilizando IPP poderia ser degradada.

6.1 Trabalhos Futuros

Apesar do estudo feito retratar uma aplicação real, sabe-se que este trabalho cobriu apenas algumas das implementações disponíveis atualmente. A avaliação de outras bibliotecas, como *Math Library for Intel (MKL)*, *NVIDIA Performance Primitives (NPP)*, entre outras, também podem se apresentar como opções de otimização do processamento de sinais para um grande volume de dados.

Também deseja-se testar as implementações em mais computadores e implementar algoritmos que exijam mais poder de processamento, como por exemplo, algoritmos de *Fast Fourier Transformer*, *beamforming* adaptativos e outros algoritmos que envolvam inversão de matrizes.

Referências

[1] V. K. Myalapalli, J. K. Myalapalli, and P. R. Savarapu, "High performance c programming," in *2015 International Conference on Pervasive Computing (ICPC)*, Jan 2015, pp. 1–6.

[2] A. DeviRangaLakshmi, S. R. Inabithini, and P. Venkataramana, "Realization of signal processing algorithms using intel integrated performance primitives (ipp)," in *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, March 2017, pp. 1–4.

[3] K. V. S. Swaroop and K. R. Rao, "Performance analysis and comparison of jm 15.1 and intel ipp h.264 encoder and decoder," in *2010 42nd Southeastern Symposium on System Theory (SSST)*, March 2010, pp. 371–375.

[4] T. Rogala, A. Kawalec, and M. Szugajew, "Implementation of effective beamforming algorithm in cuda computing technology," in *2017 18th International Radar Symposium (IRS)*, June 2017, pp. 1–7.

[5] J. Kwon, J. H. Song, S. Bae, T. Song, and Y. Yoo, "An effective beamforming algorithm for a gpu-based ultrasound imaging system," in *2012 IEEE International Ultrasonics Symposium*, Oct 2012, pp. 619–622.

[6] H. Van Trees, *Detection, Estimation, and Modulation Theory, Optimum Array Processing*, ser. Detection, Estimation, and Modulation Theory. Wiley, 2004.

[7] ATLAS ELEKTRONIK KRUPP - TED Western Germany Press, *Atlas Sonar Equipment CSU 83-1/014 - Operating and Repair Instructions*, Março 1988.

[8] "Intel integrated performance primitives," 2018. [Online]. Available: https://software.intel.com/en-us/intel-ipp

[9] "Ipp documentation," 2018. [Online]. Available: https://software.intel.com/en-us/intel-ipp/documentation

[10] "Nvidia cuda - nvidia cuda c programming guide," 2018. [Online]. Available: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

[11] "Cuda c/c++ basics - supercomputing 2011 tutorial," 2018. [Online]. Available: https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf