Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Otimização de alocação de memória em arquiteturas de processamento em *stream*

Rafael Douglas Sousa Barreto dos Santos

Recife

Janeiro de 2015

Rafael Douglas Sousa Barreto dos Santos

Modelos de programação linear inteira para a alocação de memória em arquiteturas de processamento em *stream*

Monografia apresentada como exigência para obtenção do grau de Bacharel em Sistemas de Informação da Universidade Federal Rural de Pernambuco.

Orientador: Victor Wanderley de Costa Medeiros
Co-Orientador: Glauco Estácio Gonçalves

Recife
Janeiro de 2015

À Deus, Meus familiares e a todos os meus amigos da minha graduação.

AGRADECIMENTOS

Primeiramente agradeço à Deus, Por ser a minha fortaleza e que sempre se fez presente na minha vida. Aos meus familiares e meus amigos que me apoiaram em todos os momentos da minha graduação. Ao meu orientador Victor Medeiros e ao meu co-Orientador Glauco Gonçalves que me guiaram e auxiliaram nos mais diversos caminhos para conseguir produzir este trabalho de conclusão de curso.

Como é feliz o homem que acha a sabedoria, o homem que obtém entendimento.
Pois melhor é o lucro que ela dá, do que o lucro da prata, e a sua renda do que o ouro.

Provérbios 3:13

RESUMO

A alocação adequada de *buffers* em memórias que pertencem a arquiteturas de processamento em *stream* é um fator de extrema importância para se obter alto desempenho. Para se obter um melhor acesso aos dados que estão nos *buffers*, os quais serão alocados nas memórias, é necessário que ocorra uma distribuição balanceada dos *buffers* de modo que se utilize quase toda banda disponível, evitando ao máximo os desperdícios dos bancos de memórias utilizados. O desenvolvimento deste trabalho está focado na criação de modelos de programação linear inteira e a partir da minha modelagem do problema, solucioná-los por meio de algoritmos de otimização numérica implementados em um *solver*. Através destes modelos é possível encontrar a melhor alocação para os conjuntos de *buffers* e memórias, quando esta alocação é possível. O conjunto de dados de teste foi baseado em um problema real de processamento em *stream* sobre plataformas FPGA que apresenta alto custo computacional e acesso massivo aos dados.

Palavras-chave: processamento em *stream*, alocação de memória, computação de alto desempenho, programação linear inteira, *solvers*.

ABSTRACT

The proper allocation of memory buffers in stream processing architectures is a very important factor for obtaining high performance. To gain better access to data in buffers, which will be allocated in memory, it is necessary a balanced distribution of buffers that enables the use of almost all-available bandwidth, avoiding storage waste on memory banks. The development of this work is focused on creating integer linear programming models and solve them using numerical optimization algorithms implemented in a solver. Through this model it is possible to find the best allocation for sets of buffers and memories, when allocation is possible. The test dataset was based on a real problem in stream processing on FPGA platforms that has a high computational cost and massive data access.

Keywords: stream processing, memory allocation, high performance computing, linear programming, solvers.

SUMÁRIO

1 INTRODUÇÃO 9
1.1 Cenário 9
1.2 Descrevendo o problema 11
1.3 Objetivo 12
2 FUNDAMENTAÇÃO TEÓRICA 14
2.1 Elementos Envolvidos 14
2.1.1 Stream Processing (processamento em stream) 14
2.1.2 Buffers 14
2.1.3 Bancos de memória 14
2.1.4 Largura de banda ou taxa de transferência 15
2.2 *Solvers* 15
2.2.1 CPLEX (IBM ILOG CPLEX Optimization Studio) 15
3 SOLUÇÃO 17
3.1 Modelo I 17
3.1.1 Parâmetros 17
3.1.2 Variáveis 17
3.1.3 Restrições 18
3.1.4 Função objetivo 19
3.1.5 Modelagem Cplex (JAVA) 19
3.2 Modelo II 19
3.2.1 Parâmetros 19
3.2.2 Variáveis 20
3.2.3 Restrições 20
3.2.4 Função objetivo 21
3.2.5 Modelagem Cplex (Java) 21
4 Testes 21

4.1 Gerador de dados 21
4.2 Descrições dos testes 22
5 RESULTADOS 23
5.1 Resultados Obtidos 23
6 CONCLUSÃO 30
6.1 Dificuldades encontradas 30
6.2 Lições aprendidas 31
6.3 Trabalhos Futuros 31
REFERÊNCIAS 32
APÊNDICE A – Código-fonte do Modelo I 33
APÊNDICE B – Código-fonte do Modelo II 34
APÊNDICE C – Código-fonte do Gerador de Entradas 35

1 INTRODUÇÃO

1.1 Cenário

O processamento em *stream* é um modelo de processamento que possibilita explorar limitadas formas de processamento paralelo. Esta técnica é bastante semelhante ao processamento SIMD (*single instruction multiple data* - instrução única sobre múltiplos dados), no entanto se diferencia devido aos processamentos em SIMD utilizarem instruções simples para operar em vetores de dados, enquanto os processamentos em *stream* utilizam *kernels de processamento (núcleos de processamento)* para operar em *streams.*

Um *stream de entrada* é uma matriz de elementos de dados que podem ser operados em paralelo. *Streams* de entrada são alimentados em um fluxo chamado de processamento em *stream*, onde em algum momento essas entradas serão operadas por coleções de instruções *kernels* aplicadas pelo processador de *stream*. Um *kernel* é uma sequência de instruções que se destina a ser aplicada a cada elemento de um *stream*. Assim, uma função de *kernel* age como um pequeno laço que se repete uma vez para cada elemento do *stream* de entrada.

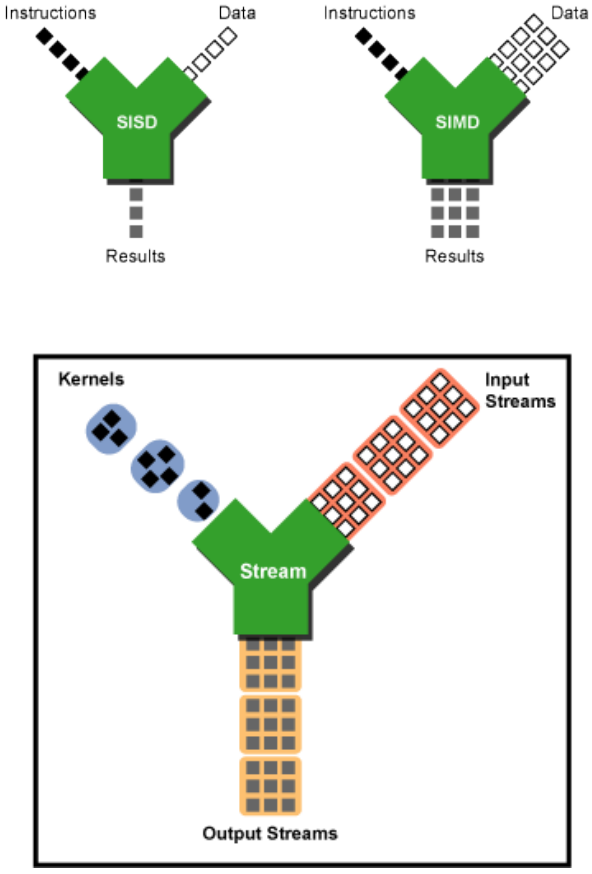

Figura 1 – Comparação entre *SISD*, *SIMD* e *Stream Processing*

Fonte: (STOKES, 2006)

Na Figura 1, está simbolizado de forma abstraída como os dados são tratados em um SISD, SIMD e processamento em *stream*. Nota-se a partir da análise da figura 1 que diferentemente do SISD (*single instruction single data* – instrução única sobre dado único) e SIMD o processamento em *stream* executa diversas operações em cima dos conjuntos de dados (STOKES, 2006).

Observando a figura 1 percebe-se que as etapas que ocorrem no processamento em *stream são de fácil entendimento*, pois nota-se que os *kernels* fluem para o processamento juntamente com o *stream dos dados de entrada* e em seguida após serem executadas as operações sobre os dados, geram-se *streams* de saída.

Uma diferença importante a ser notada entre processamento em stream e SISD ou o processamento SIMD é que os dados gerados são armazenados em registros (o contador de programa e registradores de dados), enquanto *kernels* e *streams* são ambos armazenados no *cache*. Assim o processamento em *stream* faz uso da localidade de referência, agrupando explicitamente os códigos e dados relacionados em conjunto para facilitar a busca na cache.

Para que aplicações desta natureza possam extrair o máximo de desempenho do *hardware* disponível na plataforma utilizada, e para que se evite o desperdício de recursos, técnicas de otimização da alocação destes recursos devem ser exploradas. Este trabalho tem como objetivo principal a otimização da alocação dos recursos de memória que fornecem dados para uma aplicação de processamento em *stream*.

Na literatura encontramos alguns trabalhos que também abordam este tópico de diferentes formas. O trabalho de (BALASA; CATTHOOR; MAN, 1994) mostra que o problema em questão já é abordado há algum tempo. Este trabalho propõe uma técnica de alocação de memória baseada em análise de fluxo de dados, empregando a área de silício utilizada como um estimador do custo. Esta técnica de alocação produz uma ou várias arquiteturas de memória de acordo com as características pré-definidas e que atendam as demandas de frequência de funcionamento nas operações de leitura e escrita.

Outros trabalhos como (GAL; CASSEAU; HUET, 2008) abordam o gerenciamento do acesso a memória em aplicações de processamento de dados digitais multimídia. Através da detecção de padrões de acesso às memórias, uma ferramenta é capaz de gerar interfaces de memória inteligentes capazes de se adequar de forma otimizada ao padrão de acesso identificado. Como consequência do uso desta ferramenta o autor afirma que foi possível reduzir consumo de energia e latência ao acessar os dados na memória.

O trabalho de (KULKARNI; ARUNACHALAM, 2011) se concentra na definição de um

escalonador dinâmico de memória para aplicações multimídia. Inicialmente o artigo faz uma revisão sobre os escalonadores de memórias já apresentados na literatura. Em seguida, ele apresenta uma solução de escalonador de memória para o padrão de vídeo H.264. Os resultados mostraram uma redução no tempo de execução quando comparados com as soluções anteriores.

O trabalho apresentado em (WANG *et al.*, 2012) explora a alocação de *buffers*, que são uma abstração de uma região de memória com algumas restrições específicas, em memórias em processos de software embarcado. Os autores propõem um esquema de alocação de memória que reduz em média 26% do consumo de memória quando comparado com outras técnicas apresentadas. No melhor caso a redução chega a 57%.

Em (MEFTALI *et al.*, 2001) é apresentado um fluxo de projeto automático para geração de uma arquitetura de memória compartilhada para uma aplicação específica. A solução busca uma arquitetura ótima minimizando o custo global de acesso a memória compartilhada. O problema é tratado através da criação de um modelo de programação linear inteira. O funcionamento da técnica é demonstrado através de um exemplo de roteamento de pacotes.

Para que se obtenha alto desempenho em arquiteturas de processamento em *stream* é essencial uma ótima alocação dos recursos de memória. Uma má alocação destes recursos implica na subutilização dos recursos de processamento. Portanto, este trabalho está focado na otimização da alocação de memória e não no tratamento de aspectos relacionados ao processamento em *stream* propriamente dito.

1.2 Descrevendo o problema

A necessidade de procurar uma forma de otimizar a alocação de memória em arquiteturas de processamento em *stream (Figura 1),* surgiu com o estudo de uma aplicação real baseada em FPGAs que processa uma carga massiva de dados e demanda alta capacidade computacional. Esta aplicação utiliza *buffers* para armazenar as informações que serão processadas e em seguida armazenadas novamente em outros *buffers*.

A aplicação que deu origem a este estudo, atualmente executa grandes processamentos de dados e isto pode demorar bastante, pois além da quantidade enorme de dados que será processada, o desenvolvedor da aplicação pode não distribuir os dados de forma balanceada em todas as memórias e que evite o desperdício de algumas memórias. Logo, constatou-se a necessidade de criação de uma aplicação intermediária que identificaria qual a melhor alocação para um determinado conjunto de dados, que estariam agrupados em *buffers* e que seriam acessados na maior taxa de frequência possível.

Com intuito de produzir informações de como construir arquiteturas ideais para projetistas de hardware que busquem utilizar o melhor desempenho dos seus recursos

físicos (bancos de memórias) e que possivelmente evitem o desperdício destes nas suas aplicações, este trabalho se dividiu em dois problemas a serem solucionados:

O primeiro problema é referente ao melhoramento do acesso aos dados que foram alocados em memória, com intuito de encontrar uma solução para este primeiro problema, buscou-se construir um modelo que encontre a frequência máxima para executar as operações de leitura e escrita sobre um conjunto de memórias fixo que consegue alocar uma quantidade de *buffers* de dados predeterminados. Com esta informação em mãos, os projetistas de *hardware* saberiam qual a taxa de frequência máxima que aqueles *buffers* poderiam ser acessados e poderiam verificar se a arquitetura que possivelmente seria utilizada é viável para seus fins ou se seria necessário a criação de uma arquitetura com novos recursos físicos e com outras especificações.

O segundo problema no qual este trabalho propõe uma solução, é referente aos desperdícios de recursos (bancos de memórias) que estão sendo utilizados na arquitetura para a alocação dos buffers. Como proposta de solução, criamos um modelo referente ao segundo problema com o intuito de que este modelo fornecesse a capacidade mínima de bancos de memórias que fosse possível para a alocação de todos os buffers após fixar uma frequência máxima, no qual esta frequência foi obtida do modelo referente ao primeiro problema e que sofreu um pequeno relaxamento visando uma redução significativa de recursos sem grandes percas de desempenho. Percebe-se que os resultados gerados por esse modelo podem acarretar numa economia financeira de recursos (bancos de memórias) ou na diminuição do desperdício de recursos, pois, com essas informações em mãos, o arquiteto da arquitetura de hardware saberá quais bancos de memórias possuem capacidade de serem utilizados para outras funções e não só o processamento de dados em si.

1.3 Objetivo

Este trabalho tem como objetivo principal modelar o problema de alocação de *buffers* em bancos de memória em arquiteturas de processamento em *stream* como um problema de programação linear inteira e solucioná-lo por meio de algoritmos de otimização numérica implementados em um *solver*.

Durante o desenvolvimento foram empregados conceitos e técnicas que abrangem as áreas de modelagem computacional, arquitetura de computadores e computação de alto desempenho.

De forma específica, pretende-se:

- Criar um primeiro modelo que resolva o problema de alocação de *buffers* em memória com intuito de encontrar o período mínimo que é equivalente a frequência máxima de acesso aos dados baseado que foi possível uma

alocação ótima;

- Criar um segundo modelo que consiga baseado no relaxamento do período encontrado pelo primeiro modelo, possa encontrar a quantidade mínima de memórias utilizadas para a alocação dos *buffers*;
- Explorar diferentes modelos matemáticos para o problema de alocação de *buffers* em memórias;
- Converter os modelos matemáticos em código para serem aplicados no *solver* Cplex;
- Compreender o problema de alocação de *buffers* em memórias no contexto de arquiteturas de processamento em *stream* em plataformas FPGA;
- Implementar um gerador automático de entradas que serão utilizados na avaliação dos modelos matemáticos;
- Avaliar a viabilidade da solução do problema através de um *solver*;
- Analisar os dados gerados pelo *solver* na solução do conjunto de entradas de teste.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Elementos Envolvidos

Este tópico descreve de forma conceitual a técnica de processamento utilizada e os elementos envolvidos neste tipo de técnica.

2.1.1 Stream Processing (processamento em stream)

É um paradigma de programação de computadores, relacionado com SIMD (*Single Instruction*, *Multiple Data* - instrução única em múltiplos dados), que se baseia no processamento paralelo de grandes fluxos de dados, conseguindo ter um aproveitamento potencial dos dados que estão em constante movimento. O grande benefício está na simplificação do software e do hardware com a estratégia de utilizar uma série de operações (funções do *kernel*) aplicada a cada elemento no fluxo, ou seja, um streaming uniforme. . (“IBM Systems - Intelligent Threads - United States”, 2014)

O processamento em *stream* consegue extrair informações de dados que estão em constante movimento sem a necessidade de interromper este fluxo para que seja possível fazer esta análise. Esta característica é fundamental em sistemas onde os dados de entrada estão sempre mudando. (“IBM Systems - Intelligent Threads - United States”, 2014).

2.1.2 Buffers

Buffer pode ser definido como uma região de memória temporária utilizada para escrita e leitura de dados, geralmente são alocados em memórias RAM de alta velocidade onde o acesso aos dados é mais rápido do que se fosse armazenado em disco. Normalmente são utilizados quando existe uma diferença entre a taxa em que os dados são recebidos e a taxa em que eles podem ser processados, ou no caso em que essas taxas são variáveis (JESIEL SIQUEIRA MACHADO, 2007).

2.1.3 Bancos de memória

São unidades lógicas de armazenamento que possuem *hardware*s dependentes. Em um computador, os bancos de memórias podem ser determinados pelo controlador de acesso às memórias, juntamente com a organização física dos *slots* de *hardware* de memória. Em memórias SDRAM (*synchronous dynamic random-access memory*), um banco é composto por várias linhas e colunas de unidades de armazenamento e é normalmente espalhado por vários chips. Em uma única operação de leitura ou escrita, apenas um banco é acessado, por isso o número de bits em uma coluna ou uma linha, por banco e por chip, equivale à largura do barramento de memória em bits (um canal). O tamanho de um banco é ainda determinado pelo número de bits em uma coluna ou uma linha, por chip, multiplicado pelo número de chips num banco.

2.1.4 Largura de banda ou taxa de transferência

A definição de largura de banda refere-se à taxa de transferência máxima de um canal de comunicações. No caso das memórias, a quantidade de Bytes que podem ser transferidos de ou para a memória em cada segundo, sendo MegaBytes por segundo (MB/s) ou GigaBytes por segundo (GB/s), unidades usualmente utilizadas (“Tudo o Que Você Precisa Saber Sobre as Arquiteturas de Memória de Dois, Três e Quatro Canais - Clube do Hardware”, [s.d.]).

2.1.5 Portas

Existe um módulo *Multiport* na aplicação que deu origem a este estudo, que exerce o trabalho de uma camada de abstração entre as memórias RAM que estão ligadas ao FPGA e o núcleo de processamento, este módulo provê uma interface simples e eficiente para a leitura e escrita de dados. Através deste módulo a aplicação tem acesso a múltiplas portas virtuais que podem ter diferentes tamanhos de palavras. (MEDEIROS, V. W. C.).

Na aplicação tratada neste trabalho, cada *buffer* utiliza uma ou duas portas caso ele seja de apenas leitura, apenas escrita ou ambos. Cada banco físico de memória apresenta uma restrição com relação a quantidade máxima de portas que podem ser utilizadas.

2.2 *Solvers*

Solver é um termo genérico usado para indicar um pedaço de um *software*, possivelmente na forma de um programa padrão de computador ou como uma biblioteca, que busca resolver problemas matemáticos, baseado em descrever os problemas de uma forma genérica e encontrar uma possível solução. Um sinônimo de *solver*, em inglês, é “*optimizer*” (otimizador), pois na maioria dos casos o *solver* é utilizado para otimizar a alocação de vários recursos.

Em nosso problema o *solver* foi utilizado para encontrar o menor período possível, que corresponde a maior taxa de frequência de acesso aos dados, dado um arranjo de memórias e *buffers* com suas devidas restrições. E em seguida foi feita outra otimização, buscando obter uma quantidade mínima de memórias para alocação ótima dos buffers, baseada na fixação do período encontrado na primeira otimização e aplicando um pequeno relaxamento do seu valor.

2.2.1 CPLEX (IBM ILOG CPLEX Optimization Studio)

É um *solver* que possui um pacote de *software* de otimização abrangente para desenvolver aplicativos de suporte à decisão analítica, com base em otimização. O CPLEX resolve problemas de programação inteira, programação linear de larga escala, programação quadrática e problemas com restrições quadráticas convexas (“IBM ILOG CPLEX Optimization Studio”, 2015).

O CPLEX possui uma camada de modelagem que fornece interfaces para

linguagens como C++ , C# e Java. Há uma interface para Python baseada na interface para C.

Para este trabalho de conclusão de curso, utilizou-se a interface para linguagem Java para modelar o problema, gerar as soluções e otimizações.

3 SOLUÇÃO

3.1 Modelo I

O primeiro modelo tem como objetivo encontrar a maior frequência possível de operação do processamento sobre uma alocação dos *buffers* nas memórias, com o objetivo de conseguir o maior desempenho possível da arquitetura. Ao invés de buscar a maior frequência o modelo foi construído de modo a minimizar o período de *clock*. O resultado final com esta outra abordagem é o mesmo, visto que, o valor do período T é inversamente proporcional ao valor da frequência F, logo obtendo T é possível obter F, Essa abordagem foi adotada pois quando foi tentado criar um modelo que usasse F em vez de T, o solver acabou interpretando o problema como não linear. Contudo, todas as restrições dos *buffers* e das memórias devem ser atendidas. Estas restrições e os parâmetros empregados no modelo de programação linear inteira são esclarecidos a seguir.

3.1.1 Parâmetros

m: Corresponde a quantidade de memórias;

n: Corresponde a quantidade de buffers;

$C_j \quad \forall\, j \in \{1, \ldots, m\}$

Onde C_j corresponde a capacidade da memória j;

$B_j \quad \forall\, j \in \{1, \ldots, m\}$

Onde B_j corresponde a banda da memória j;

$P_j \quad \forall\, j \in \{1, \ldots, m\}$

Onde P_j corresponde a quantidade de portas da memória j;

$s_i \quad \forall\, i \in \{1, \ldots, n\}$

Onde s_i corresponde ao tamanho do *buffer* i;

$r_i \quad \forall\, i \in \{1, \ldots, n\}$

Onde r_i corresponde a taxa de acesso do *buffer* i;

$q_i \quad \forall\, i \in \{1, \ldots, n\}$

Onde q_i corresponde a quantidade de portas do *buffer* i;

3.1.2 Variáveis

f: Frequência de operação de processamento;

T: Corresponde ao período de clock do processador e é o mesmo que 1/f;

$X_{ij} \quad \in \{0,1\}, \qquad \forall i \in \{1, \ldots, n\};\ \forall j \in \{1, \ldots, m\}$

Variável booleana X_{ij}, onde 1 significa que o buffer i está alocado na memória j, e o 0 para dizer que não foi alocado na solução.

3.1.3 Restrições

1)

$$\sum_{i}^{n} s_i \, \mathrm{X_{ij}} \leq \mathrm{C_j} \; \forall \, \mathrm{j} \in \{1, \ldots, \mathrm{m}\}$$

A primeira restrição afirma que o somatório dos tamanhos dos *buffers* alocados na memória j na solução ótima não poderá ser maior que a capacidade da memória j. Esta restrição foi criada para que não fosse possível alocar buffers quando a capacidade da memória j for atingida.

2)

$$\sum_{i}^{n} r_i \, \mathrm{X_{ij}} \leq \mathrm{B_j}.\mathrm{T} \quad \forall \, \mathrm{j} \in \{1, \ldots, \mathrm{m}\}$$

A segunda restrição afirma que o somatório das taxas de acesso dos *buffers* alocados em uma memória j, não pode ser maior que a largura de banda da memória j vezes um valor de período fixo.

Esta restrição foi criada para que quando a soma das taxas de acesso aos *buffers* não ultrapassassem a largura de banda da memória vezes o período mínimo de *clock*, pois caso essa restrição fosse descumprida, ocorreria um funilamento do acesso aos buffers baseado que o acesso ficaria restringido a largura de banda da memória vezes o período de *clock*.

3)

$$\sum_{i}^{n} q_i \, \mathrm{X_{ij}} \leq \mathrm{P_j} \; \forall \, \mathrm{j} \in \{1, \ldots, \mathrm{m}\}$$

A terceira restrição afirma que o somatório das quantidades de portas dos *buffers* alocados em uma memória j, não pode ser maior que a quantidade de portas disponíveis na memória j. Esta outra restrição busca restringir o uso máximo de portas que os *buffers* poderão utilizar baseado na soma das quantidades de portas que as memórias possuem.

4)

$$\sum_{i}^{n} \mathrm{X_{ij}} = 1 \; \forall \; \mathrm{j} \; \in \{1, \ldots, \mathrm{m}\}$$

A quarta restrição afirma que cada *buffer* alocado em uma memória j, só pode estar em uma única memória. Esta restrição impede que os buffers sejam alocados em mais de uma memória, oque causaria ambiguidade de dados.

3.1.4 Função objetivo

Minimizar T, ou seja encontrar o valor máximo para F pois como T é inversamente proporcional a F, o mínimo de T é o máximo de F, considerando todas as restrições apresentadas anteriormente.

3.1.5 Modelagem Cplex (JAVA)

O ILOG CPLEX 7.5 Java fornece uma API para aplicativos Java que utilizam CPLEX. Isso permite que o aplicativo Java consiga chamar o CPLEX diretamente, por meio da interface nativa java. A interface Java é construída em cima de Tecnologia Concert ILOG para Java e fornece uma rica funcionalidade que lhe permite usar objetos Java para construir seu modelo de otimização.

O código fonte do modelo I pode ser encontrado no Apêndice A, nele são modelados os parâmetros, as variáveis de decisão, as restrições e a função objetivo que retorna o período mínimo, no qual é o equivalente a frequência máxima de acesso aos dados para alocação dos *buffers*.

3.2 Modelo II

O segundo modelo tem como objetivo minimizar a quantidade de memórias baseado num valor fixo do período T. Esta característica permite uma redução do desperdício de memória e uma melhor utilização do *hardware*. Aplicam-se a este modelo as restrições de memórias e *buffers* apresentadas em seguida.

3.2.1 Parâmetros

m: Corresponde a quantidade de memórias;

n: Corresponde a quantidade de buffers;

$C_j \quad \forall \, j \; \in \{ 1, \quad \ldots \quad , m \}$

Onde C_j corresponde a capacidade da memória j;

$B_j \quad \forall \, j \; \in \{ 1, \quad \ldots \quad , m \}$

Onde B_j corresponde a banda da memória j;

$P_j \quad \forall \, j \; \in \{ 1, \quad \ldots \quad , m \}$

Onde P_j corresponde a quantidade de portas da memória j;

$s_i \quad \forall\, i \in \{1, \ \dots \ , n\}$

Onde s_i corresponde ao tamanho do *buffer* i;

$r_i \quad \forall\, i \in \{1, \ \dots \ , n\}$

Onde r_i corresponde a taxa de acesso do *buffer* i;

$q_i \quad \forall\, i \in \{1, \ \dots \ , n\}$

Onde q_i corresponde a quantidade de portas do *buffer* i;

3.2.2 Variáveis

f: Frequência de operação de processamento;

T: Corresponde ao período de clock do processador e é o mesmo que 1/f;

$X_{ij} \quad \in \{0,1\}, \qquad \forall i \in \{1, \dots , n\};\ \forall j \in \{1, \dots , m\}$

Variável booleana X_{ij}, onde 1 significa que o buffer i está alocado na memória j, e o 0 para dizer que não foi alocado na solução.

$Y_j \quad \in \{0,1\}, \qquad \forall j \in \{1, \dots , m\}$

Variável booleana Y_j, onde 1 significa que a memória j foi escolhida para alocação dos *buffers* e o 0 para informar que a memória j não foi utilizada na alocação dos *buffers*.

3.2.3 Restrições

1)

$$\sum_{i}^{n} s_i \, \mathrm{X_{ij}} \ \leq \mathrm{C_j} \ \ \forall\, \mathrm{j} \in \{1, \dots , \mathrm{m}\}$$

A primeira restrição afirma que o somatório dos tamanhos dos *buffers* alocados nas memórias escolhidas na solução ótima não poderá ser maior que a capacidade das memórias escolhidas.

2)

$$\sum_{i}^{n} r_i \, \mathrm{X_{ij}} \ \leq \mathrm{B_j}.\mathrm{Y_j}.\mathrm{T} \ \ \forall\, \mathrm{j} \in \{1, \dots , \mathrm{m}\}$$

A segunda restrição afirma que o somatório das taxas de acesso dos *buffers* alocados em uma memória escolhida, não pode ser maior que a largura de banda da memória escolhida vezes um valor de período fixo.

3)

$$\sum_{i}^{n} q_i \, \mathrm{X_{ij}} \leq \mathrm{P_j Y_j} \quad \forall \, \mathrm{j} \in \{1, \ldots, \mathrm{m}\}$$

A terceira restrição afirma que o somatório das quantidades de portas dos *buffers* alocados em uma memória escolhida, não pode ser maior que quantidade de portas disponíveis na memória escolhida.

4)

$$\sum_{i}^{n} \mathrm{X_{ij}} = 1 \; \forall \, \mathrm{j} \in \{1, \ldots, \mathrm{m}\}$$

A quarta restrição afirma que cada *buffer* alocado em uma memória j, só pode estar em uma memória.

3.2.4 Função objetivo

$$minimizar \sum_{j}^{m} \mathrm{Y_j}$$

Minimizar o somatório da quantidade de bancos de memórias, baseado em que todas as restrições foram atingidas, com intuito de obter uma redução da quantidade de recursos físicos de memória utilizados para alocar os *buffers* de dados na aplicação.

3.2.5 Modelagem Cplex (Java)

O código fonte do modelo II pode ser encontrado no apêndice **B**, nele são modelados os parâmetros, as variáveis de decisão, as restrições e a função objetivo que retorna a quantidade mínima de memórias para que seja possível a alocação de buffers em memórias foi utilizada uma mínima redução do período encontrado pelo modelo I, correspondente a 10% do seu valor total.

4 Testes

4.1 Gerador de dados

O gerador de dados para os modelos I e II foi criado em Java e foi projetado para alimentar o *solver* com dados semelhantes aos usados em uma aplicação real.

Foi necessário um conhecimento técnico sobre as configurações de *hardware* que a aplicação em que esse estudo teve origem possui ou poderia possuir, para que o gerador de dados construa problemas com restrições mais próximas possíveis do mundo real e para que as simulações que serão executadas posteriormente estejam relacionadas com dados

que não sejam muito diferentes dos usados realmente na arquitetura FPGA em que a aplicação está.

O gerador de dados gera problemas com no mínimo 2 bancos de memórias e no máximo 6, onde cada banco de memória gerado possui uma capacidade entre 256MB à 8GB, uma largura de banda entre 6400MB/s (DDR2 - 800MHz) e 12800MB/s (DDR3 – 1600MHz) e 15 portas de acesso aos dados. Com relação aos *buffers* gerados, o gerador produz problemas que existem no mínimo 5 *buffers* e no máximo 50, cada um possui tamanho máximo de 1048576KB ou frações deste valor, em relação a taxa de acesso, os valores variam de no máximo 1600KB correspondente à 1MHz ou frações deste valor e em relação a quantidade de portas varia de 1 a 15 portas.

Os valores são gerados de acordo com as especificações citadas no parágrafo anterior e são armazenados em *arrays*, onde esses dados são escolhidos através da utilização da classe *Random* do Java com intuito de gerar de forma aleatória, problemas que são aplicados nos modelos criados neste trabalho.

O código fonte do gerador de dados pode ser encontrado no apêndice C, nele as restrições dos buffers e das memórias são criadas para que hajam vários problemas que são utilizados para validar os modelos I e II.

4.2 Descrições dos testes

Para que fosse mais viável obter informações dos problemas gerados pelo gerador de dados, foi necessário categorizar os problemas em 3 tipos: baixa, média e alta complexidade, nos quais foram classificados a partir de um cálculo que se baseia na captura da maior proporção dos valores dos problemas, onde essas proporções seriam, a soma dos tamanhos dos buffers de dados dividida pela soma das capacidades de memórias, a soma da quantidade de portas dos buffers dividida pela soma da quantidades de portas das memórias e por último a soma das taxas de acesso aos buffers dividida pela largura de banda das memórias. A seguir segue a estrutura explicada anteriormente:

- **Proporção da soma de valores das características entre buffers e memórias:**
  - Pc = (∑ tamanho dos buffers)/(∑ capacidade das memórias)
  - Pt = (∑ taxa de acesso dos buffers)/(∑ banda das memórias)
  - Pp = (∑ quantidade de portas dos buffers)/(∑ quantidade de portas das memórias)

Após capturar essas proporções pega-se o argumento máximo das proporções e classifica o problema com a seguinte regra: a maior proporção está entre 0 e 40% o problema é classificado como de baixa complexidade, a maior proporção está entre 40% e

80% o problema é classificado como de média complexidade, a maior proporção está acima de 80% o problema é classificado como de alta complexidade. Esta classificação foi relacionada com estes valores, pois quando os valores das restrições dos buffers se aproximam muito com as dos bancos de memória, a alocação dos buffers fica mais complexa de ser encontrada. A seguir segur a classificação detalhada:

- **Categorias**
    - Baixa complexidade: A maior proporção está entre 0 e 40%.
    - Média complexidade: A maior proporção está entre 40% e 80%.
    - Alta complexidade: A maior proporção está entre 80% e 100%.

Após a classificação dos problemas, estes são colocados para serem executados pelos *solvers* e geram um arquivo com as informações geradas pelo *CPLEX*.

Para que fosse possível simular os casos com uma maior proximidade do real, foi utilizado um valor mínimo para o período de 0,0025, oque corresponde a uma frequência máxima de 400MHZ.

Foram analisadas as frequências obtidas por cada categoria, os desperdícios de memória que cada alocação do modelo I estava gerando, quais as reduções de desperdícios que o modelo II gerava e por último quais as reduções de memórias físicas que foram possíveis. Para cada teste abordado a seguir, serão utilizados 24 problemas por cada categoria criados pelo gerador de dados.

Vale ressaltar que os modelos utilizaram o período em vez da frequência, mas para que a visualização dos resultados dos testes ficasse mais fácil para o leitor, foi utilizado a conversão do período em frequência.

5 RESULTADOS

5.1 Resultados Obtidos

O gráfico da Figura 2 apresenta a distribuição das frequências em MHZ para os problemas que foram categorizados como de baixa complexidade.

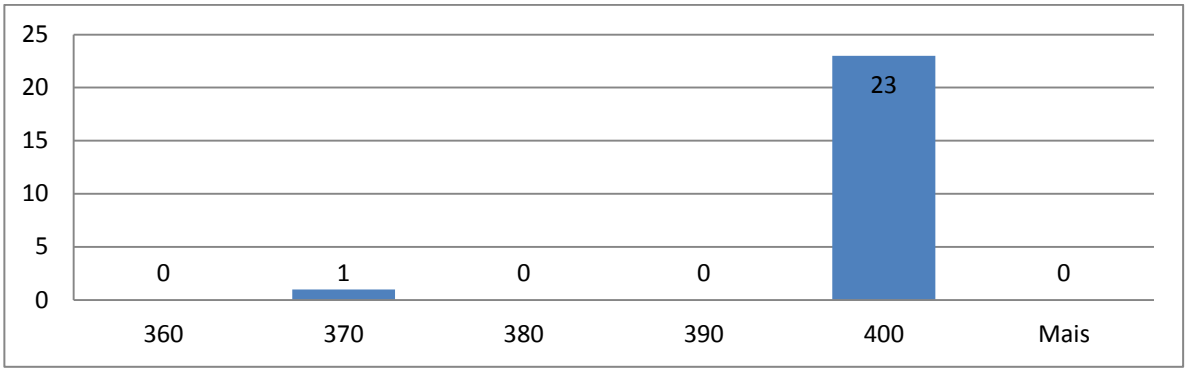

Figura 2 - Quantidade de problemas x largura de banda de memória dos problemas categorizados como de baixa complexidade

Podemos observar que as frequências (largura de banda de cada memória) dos problemas categorizados como de baixa complexidade não sofreram grandes variações, quase todos os problemas obtiveram a máxima frequência estabelecida em 400 MHz. A frequência máxima adotada está relacionada a limitação da própria plataforma FPGA. Ou seja, não faz sentido buscar-se frequências maiores do que as que são suportadas pela plataforma de hardware.

O gráfico da Figura 3 apresenta a distribuição das frequências em MHZ para os problemas que foram categorizados como de média complexidade.

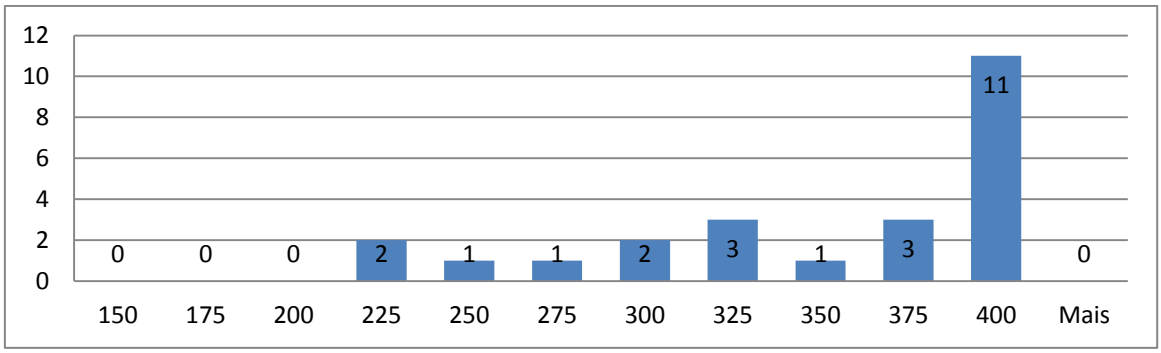

Figura 3 - Quantidade de problemas x largura de banda de memória dos problemas categorizados como de média complexidade

Já no gráfico da Figura 3 podemos observar que as frequências (largura de banda de cada memória) dos problemas categorizados como de média complexidade já tiveram uma maior variedade das frequências, pois as restrições já são de valores maiores, logo para que os *buffers* possam ser alocados nas memórias usadas é necessário que tenham uma frequência menor para atender a segunda restrição do modelo, vale ressaltar também que boa parte dos problemas ainda conseguiu ser alocado com a frequência máxima de 400 MHz.

O gráfico da Figura 4 apresenta a distribuição das frequências em MHZ para os problemas que foram categorizados como de alta complexidade.

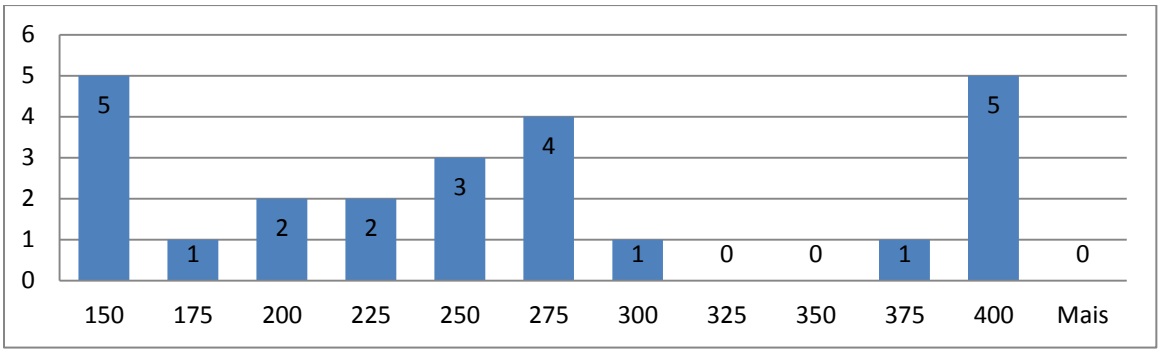

Figura 4 - Quantidade de problemas x largura de banda de memória dos problemas categorizados como de alta complexidade

Já neste último gráfico podemos observar que as frequências (largura de banda de cada memória) dos problemas categorizados como de alta complexidade já tiveram uma grande variedade, pois as restrições são de valores ainda maiores, logo para que os *buffers* possam ser alocados nas memórias usadas é necessário que tenham uma frequência menor para atender a segunda restrição do primeiro modelo, neste caso não é possível observar uma frequência que teve um número de ocorrências que se destaque.

Em seguida faremos uma análise das reduções que obtivemos com a aplicação do modelo II utilizando a frequência máxima encontrada pelo modelo I com o valor decrementado de 10% para que fosse possível obter uma redução na quantidade de memórias físicas, sem grande perdas de desempenho no acesso aos dados com essa nova frequência.

Os gráficos das Figuras 5, 6 e 7 apresentam histogramas que relacionam as reduções de memórias com a quantidade de problemas onde ocorreram as reduções nas categorias baixa, média e alta complexidade, respectivamente. Foram utilizados 24 problemas para cada categoria, totalizando 72 exemplos com 72 possíveis reduções.

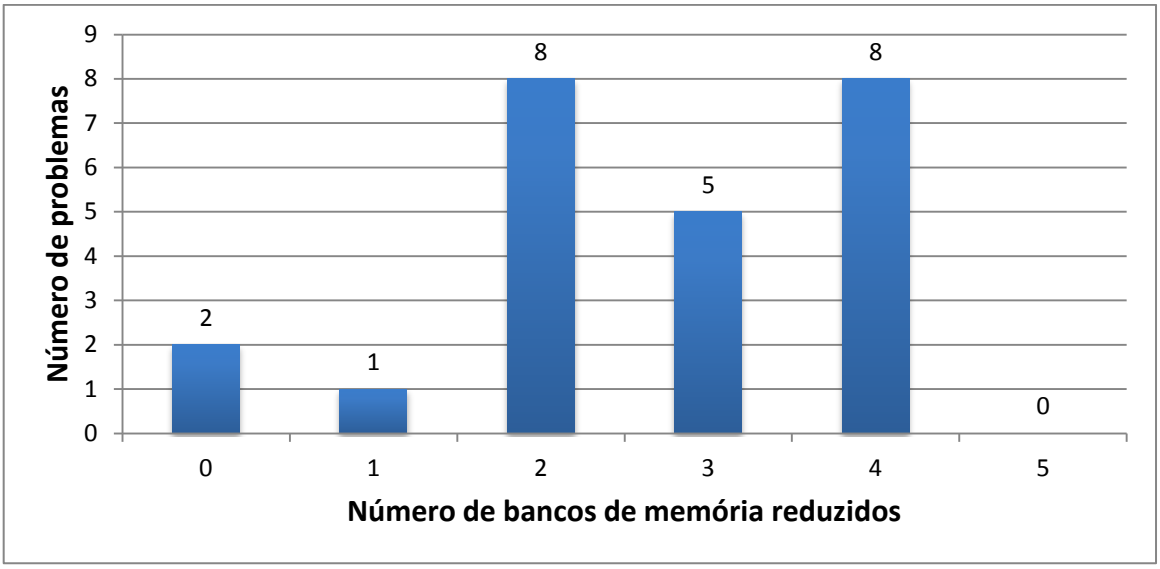

Figura 5 - Histograma do número de memórias reduzidas nos problemas categorizados como de baixa complexidade

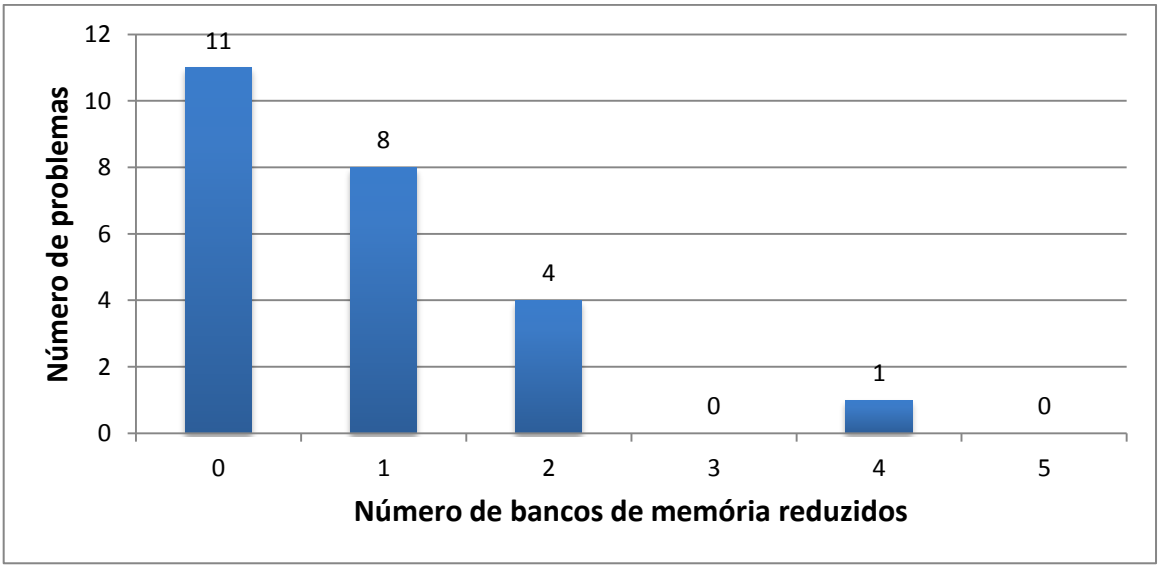

Figura 6 - Histograma do número de memórias reduzidas nos problemas categorizados como de média complexidade

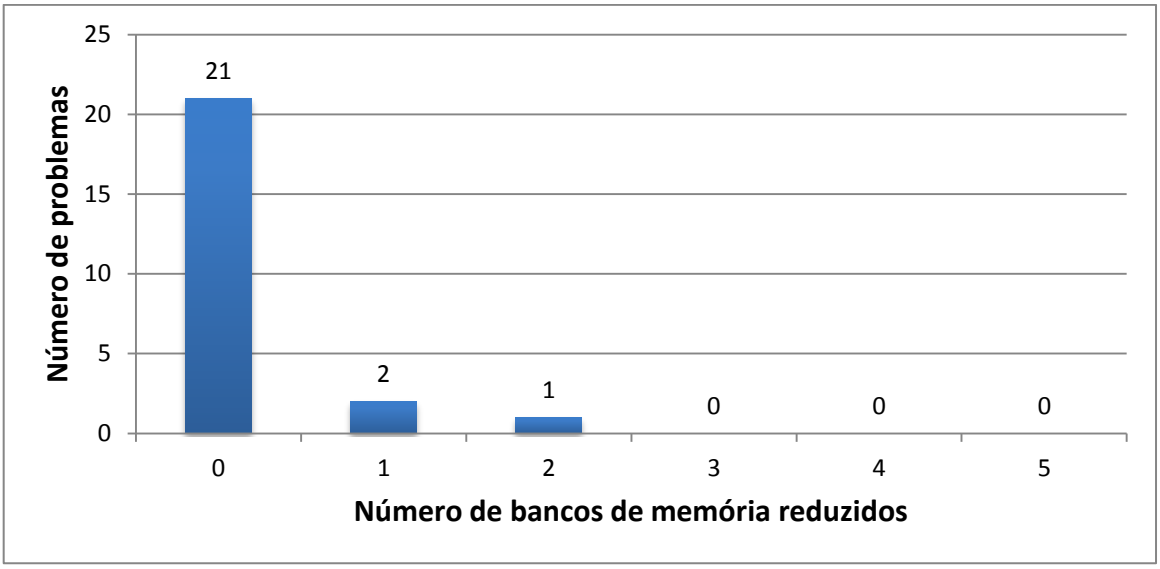

Figura 7 - Histograma do número de memórias reduzidas nos problemas categorizados como de alta complexidade

Podemos observar que as maiores reduções sempre ocorrem nos problemas categorizados como de baixa complexidade, pois estes são simples de alocar os *buffers* e geralmente uma quantia pequena de memórias é capaz de atender todas as restrições que os buffers necessitam. Nos casos de média complexidade, a maioria dos problemas também conseguiu obter uma redução das memórias físicas, já nos problemas considerados difíceis somente 3 problemas dos 24 testados conseguiram obter uma redução de memória física, isso se dá devido a alocação desses tipos de problemas ser muito custosa e requer a utilização de quase todas as memórias para que sejam atendidas todas as restrições dos *buffers* e das memórias.

O gráfico da Figura 8 representa os desperdícios de memória para 24 problemas categorizados como de baixa complexidade e que são representados em porcentagem, antes e depois da execução do modelo II. O desperdício é definido como a capacidade de armazenamento disponível de memória que não é utilizada para alocar os buffers e está disponível para uso.

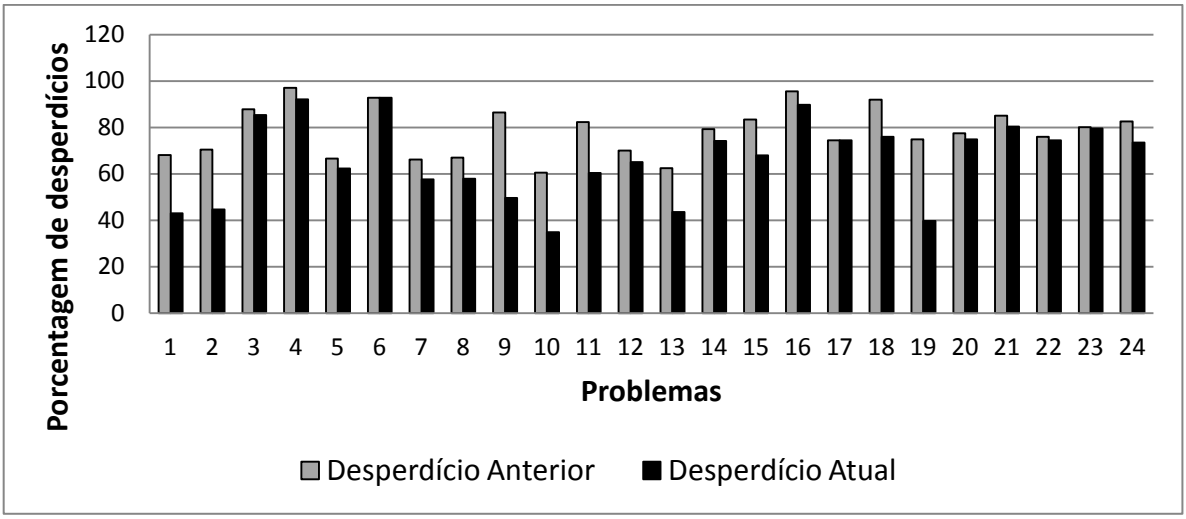

Figura 8 - Porcentagem de desperdício para os problemas categorizados como de baixa complexidade

Analisando o gráfico da Figura 8 acima, percebe-se que em 91,6% dos resultados baseados nos problemas categorizados de baixa complexidade, o *solver* encontrou uma solução que apresentou uma redução em média de 12,9% dos desperdícios nas memórias após a execução do modelo II.

O gráfico da Figura 9 representa os desperdícios de memória em 24 problemas categorizados como de média complexidade e que são representados em porcentagem, antes e depois da execução do modelo II.

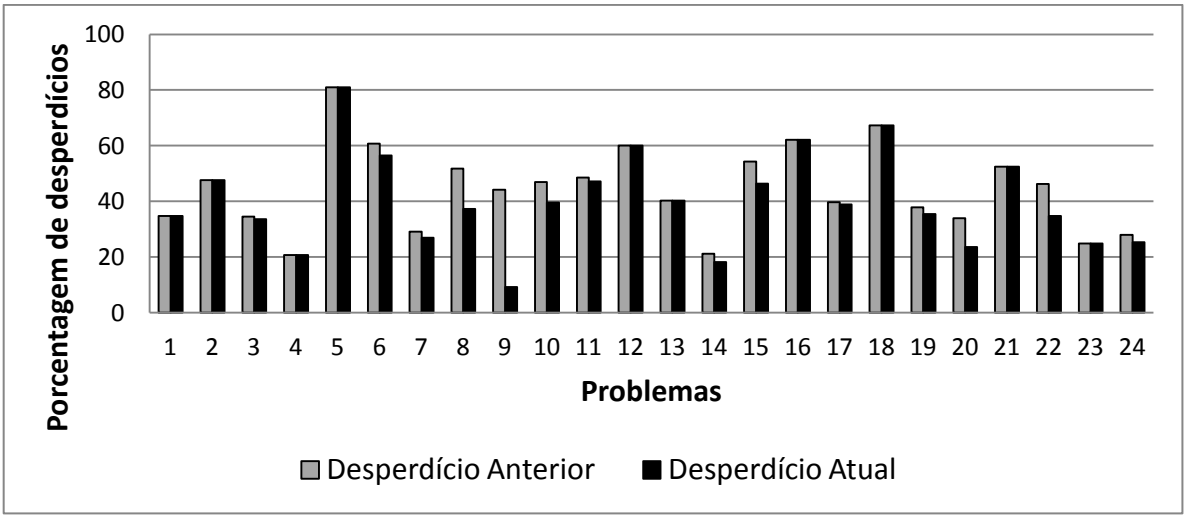

Figura 9 - Porcentagem de desperdício para os problemas categorizados como de média complexidade

Analisando o gráfico da Figura 9 percebe-se que nos resultados dos problemas categorizados de média complexidade, o *solver* encontrou uma solução em 58,3% dos

casos e conseguiu reduzir em média 7,35% dos desperdícios das memórias que ocorriam em cada caso. Nos 41,7% restante dos casos, o *solver* não encontrou uma solução que reduzisse o desperdício e a quantidade de desperdício continuou a mesma.

O gráfico da Figura 10 representa os desperdícios de memória em 24 problemas categorizados como de alta complexidade representados em porcentagem, antes e depois da execução do modelo II.

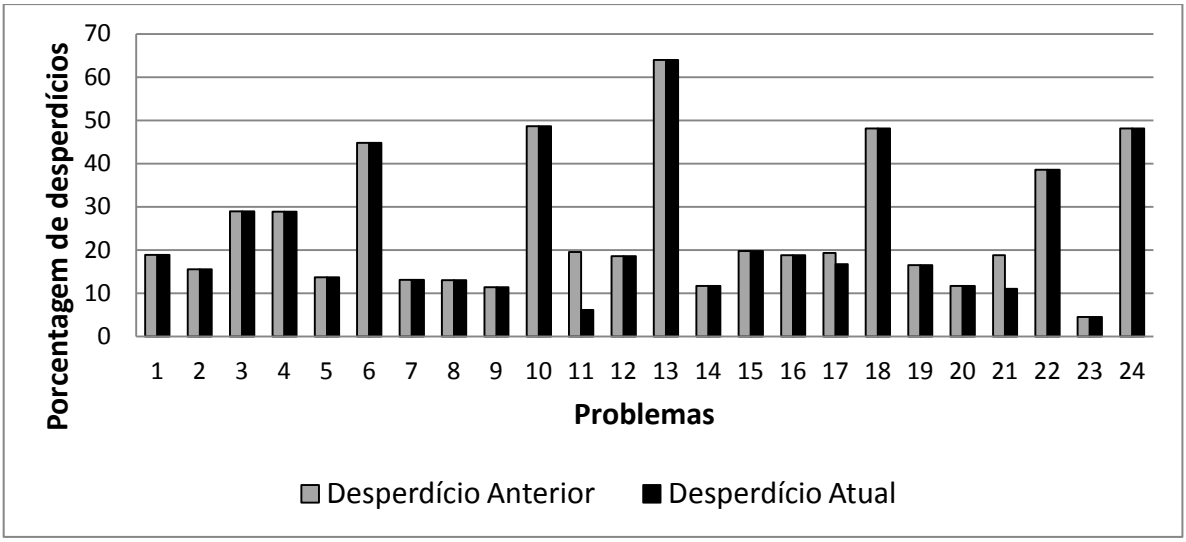

Figura 10 - Porcentagem de desperdício para os problemas categorizados como de baixa complexidade

Analisando o gráfico da Figura 10 percebe-se que apenas 12,5% dos casos o solver encontrou uma solução que reduziu em média 7,91% do desperdício de memória que ocorria em cada problema no qual o solver encontrou uma solução que era possível uma redução. Nos 87,5% dos casos restantes, o desperdício se manteve o mesmo. Como os problemas são mais difíceis de alocar com outra configuração de frequência de memória não é possível reduzir a quantidade de uso das memórias.

6 CONCLUSÃO

O presente trabalho teve como principal objetivo explorar *solvers* por meio de modelos matemáticos para conseguir capturar a maior taxa de acesso (frequência) aos dados através da alocação ótima dos *buffers* nas memórias. Outro aspecto explorado, foi a redução da quantidade de memórias que consiga alocar perfeitamente todos os buffers com o intuito de diminuir o total de desperdício da capacidade de cada memória.

Este trabalho usou restrições de um problema real, baseado numa aplicação sobre uma plataforma FPGA que executa processamento em *stream* sobre os seus dados, no qual se buscou aplicar testes sobre possíveis casos que se enquadrassem nas possíveis configurações da arquitetura.

Concluiu-se que o *Solver* é capaz de resolver estes problemas que foram classificados como problemas lineares inteiros, em tempo viável, e que gerou as frequências e a alocação ideal para os *buffers* de dados.

Este estudo também serve como base para mais adiante, implementar uma interface que seja capaz de mostrar os dados gerados pelo *solver* de uma forma intuitiva de como alocar os buffers de dados em um conjunto de memórias de forma eficiente e que busque uma alocação ideal dos dados para seu processamento, reduzindo assim o número de desperdícios da utilização da capacidade das memórias.

6.1 Dificuldades encontradas

Durante o processo de execução deste trabalho foram encontradas algumas dificuldades. A primeira delas ocorreu na modelagem dos problemas, essa modelagem foi bastante trabalhosa de ser realizada devido a necessidade de buscar conhecimento específico da aplicação real para que os modelos gerados realmente atendessem a todas as suas características.

Outra dificuldade foi criar um gerador de dados que fosse capaz de gerar dados mais próximos possíveis dos dados usados na aplicação para que os *solvers* pudessem encontrar soluções para problemas reais e usáveis.

A terceira dificuldade encontrada foi utilizar a linguagem específica do CPLEX, pois além de ter pouco material pela internet e em livros a respeito dela, a dificuldade de manipular os dados que seriam introduzidos nos modelos para se obter os testes era enorme, foi necessário encontrar uma API do CPLEX para Java com intuito de tornar mais fácil a criação dos modelos e a manipulação dos problemas gerados pelo gerador de dados.

Um quarto empecilho encontrado foi a pequena quantidade de material encontrado sobre processamento em *stream* e alocação de memórias que se assimilassem ao caso estudado neste trabalho de conclusão de curso.

6.2 Lições aprendidas

Este trabalho proporcionou inúmeros aprendizados dentre eles, vale ressaltar o aprendizado em como modelar matematicamente problemas do mundo real. Outro aprendizado está relacionado à utilização dos *solvers.* Outra lição foi o conhecimento adquirido pelo estudo de como é feita a alocação de buffers em memórias usada em aplicações que utilizam processamento em *stream.*

Por final podemos citar a vivência em um problema de computação de alto desempenho e como geralmente são abordados estes tipos de problemas com a finalidade de se obter melhoria de desempenho e gerenciamento eficiente do armazenamento de dados.

6.3 Trabalhos Futuros

Implementar um modelo híbrido que utilize o modelo I e II que visa encontrar uma otimização multi objetivo, para que não seja necessário o uso dos modelos separados e executados um após o outro, além disto favorecerá a exploração da frente de Pareto por completo e não só os seus extremos.

O segundo trabalho seria a criação de uma interface de *software* que forneça as informações geradas pelos modelos de uma forma amigável para o projetista de aplicações na plataforma FPGA.

REFERÊNCIAS

BALASA, F.; CATTHOOR, F.; MAN, H. Dataflow-driven memory allocation for multi-dimensional signal processing systems. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. p. 31–34. Disponível em: <http://dl.acm.org/citation.cfm?id=191326.191344>.

GAL, B.; CASSEAU, E.; HUET, S. Dynamic memory access management for high-performance dsp applications using high-level synthesis. **IEEE Trans. Very Large Scale Integr. Syst.**, nov. 2008. v. 16, n. 11, p. 1454–1464.

Ibm ilog cplex optimization studio. [S.l.]. Disponível em: <http://www-03.ibm.com/software/products/pt/ibmilogcpleoptistud>. Acesso em: 19 jan. 2015.

Ibm systems - intelligent threads - united states. [S.l.]. Disponível em: <http://www-03.ibm.com/systems/infrastructure/us/en/technical-breakthroughs/stream-processing.html>. Acesso em: 19 jan. 2015.

JESIEL SIQUEIRA MACHADO, S. Desenvolvimento de um sistema de captura e processamento de sinais eletroencefalograficos quantitativo não-invasivo. **Horizonte Científico**, 2007. v. 1, n. 1. Disponível em: <http://www.seer.ufu.br/index.php/horizontecientifico/article/download/3895/2899>. Acesso em: 1 fev. 2015.

KULKARNI, A. M.; ARUNACHALAM, V. Fpga implementation & comparison of current trends in memory scheduler for multimedia application. New York, NY, USA: ACM, 2011. p. 1214–1218. Disponível em: <http://doi.acm.org/10.1145/1980022.1980288>. Acesso em: 15 jan. 2015.

MEDEIROS, V.W.C. *FastRTM*: Um Ambiente Integrado para Desenvolvimento Rápido da Migração Reversa no Tempo (RTM) em Plataformas FPGA de Alto Desempenho. Fev. 2013.

MEFTALI, S. *et al.* An optimal memory allocation for application-specific multiprocessor system-on-chip. New York, NY, USA: ACM, 2001. p. 19–24. Disponível em: <http://doi.acm.org/10.1145/500001.500006>. Acesso em: 15 jan. 2015.

STOKES, J. Peakstream unveils multicore and cpu/gpu programming solution. **Ars Technica**, [S.l.], 18 set. 2006. Disponível em: <http://arstechnica.com/gadgets/2006/09/7763/>. Acesso em: 13 jan. 2015.

Tudo o que você precisa saber sobre as arquiteturas de memória de dois, três e quatro canais - clube do hardware. [S.l.], [s.d.]. Disponível em: <http://www.clubedohardware.com.br/artigos/Tudo-o-Que-Voce-Precisa-Saber-Sobre-as-Arquiteturas-de-Memoria-de-Dois-Tres-e-Quatro-Canais/673>. Acesso em: 1 fev. 2015.

WANG, A.-P. *et al.* Buffer optimization and dispatching scheme for embedded systems with behavioral transparency. **ACM Transactions on Design Automation of Electronic Systems**, 1 out. 2012. v. 17, n. 4, p. 1–26. . Acesso em: 15 jan. 2015.

APÊNDICE A – Código-fonte do Modelo I

ModeloI.java

```
1 import java.io.FileWriter;
6
7 public class ModeloI
8 {
9    public boolean solveMe(int[] tamanhoBuffer, int[] taxaDeAcessoBuffer, int[] qtdPortasBuffer,int[] capacidadeMemoria
10       , int[] larguraBandaMemoria, int[] qtdPortasMemoria,int qtdBuffers, int qtdMemoria, FileWriter arq)
11   {
12      boolean result = false;
13      try {
14         IloCplex cplex = new IloCplex(); //Definindo o modelo
15         IloNumVar T = cplex.numVar(0.01,1);
16
17         IloNumVar[][] x = new IloNumVar[qtdBuffers][];
18         for(int i = 0; i < qtdBuffers; i++){ x [ i ]  = cplex.boolVarArray( qtdMemoria ); }
19
20         for(int j = 0;j<qtdMemoria;j++)  //Primeira equacao
21         {
22            IloLinearNumExpr v = cplex.linearNumExpr();
23            for(int i = 0; i < qtdBuffers; i++){  v.addTerm( (double) tamanhoBuffer [ i ], x[ i ] [ j ] ); }
24            cplex.addLe(v, capacidadeMemoria[j]);
25         }
26         for(int j = 0;j<qtdMemoria;j++) //Segunda equacao
27         {
28            IloLinearNumExpr v = cplex.linearNumExpr();
29            for(int i = 0; i < qtdBuffers; i++) { v.addTerm( (double) taxaDeAcessoBuffer [ i ] , x[ i ] [ j ] ); }
30            cplex.addLe(v,cplex.prod(larguraBandaMemoria[j], T));
31         }
32         for(int j = 0;j<qtdMemoria;j++) //Terceira equacao
33         {
34            IloLinearNumExpr v = cplex.linearNumExpr();
35            for(int i = 0; i < qtdBuffers; i++)  { v.addTerm( (double) qtdPortasBuffer[ i ], x [ i ] [ j ] );}
36            cplex.addLe(v, qtdPortasMemoria[j]);
37         }
38         for(int i = 0;  i < qtdBuffers;  i++){ cplex.addEq( cplex.sum( x[i] ), 1); } // Quarta equacao
39
40         IloLinearNumExpr obj = cplex.linearNumExpr(); // funcao objetivo
41         obj.addTerm(1.0, T);
42
43         cplex.addMinimize(obj);
44         result = cplex.solve();
45         if(result)
46         {
47            PrintWriter gravarArq = new PrintWriter(arq);
48            gravarArq.printf(result+","+cplex.getObjValue()+","+cplex.getCplexTime()+"%n");
49         }
50         else
51         {
52            PrintWriter gravarArq = new PrintWriter(arq);
53            gravarArq.printf("false,"+"0"+","+cplex.getCplexTime()+"%n");
54         }
55      } catch (IloException e) {
56         e.printStackTrace();
57      }
58      return result;
59   }
```

APÊNDICE B – Código-fonte do Modelo II

ModeloII.java

```
import ilog.concert.*;

public class ModeloII {
   private double startTime = 0; private double endTime = 0; private double taxaDeReducao = 0.9;
   private int countMemories = 0; private int memoriesDisp;
   private double frequencia; private double frequenciaReduzida; private double periodoReduzido;
   private ArrayList<Integer> listaDeMemoriaEscolhidas = new ArrayList<Integer>();

   public boolean solveMe(int[] tamanhoBuffer, int[] taxaDeAcessoBuffer, int[] qtdPortasBuffer,
         int[] capacidadeMemoria, int[] larguraBandaMemoria, int[] qtdPortasMemoria,
         int qtdBuffers, int qtdMemoria, double periodo){
      frequenciaReduzida              = (1/periodo) * taxaDeReducao;
      periodoReduzido       = 1 / frequenciaReduzida; // Faz a redução do periodo em 10%
      boolean result = false;
      try {
         IloCplex cplex = new IloCplex();     // Definindo o modelo
         IloNumVar[][] x = new IloNumVar[qtdBuffers][]; // Variavel binaria
         for(int i = 0; i < qtdBuffers; i++){  x[ i ] = cplex.boolVarArray( qtdMemoria ); }
         IloNumVar[] y = new IloNumVar[qtdMemoria]; // Variavel binaria
            y = cplex.boolVarArray(qtdMemoria);

         for(int j = 0;j<qtdMemoria;j++){         // Primeira equação
            IloLinearNumExpr v = cplex.linearNumExpr();
            for(int i = 0; i < qtdBuffers; i++){ v.addTerm( (double)tamanhoBuffer[ i ], x[ i ] [ j ]); }
            cplex.addLe(v, cplex.prod(capacidadeMemoria[ j ], y [ j ]) );
         }
         for(int j = 0; j<qtdMemoria; j++){           // Segunda equação
            IloLinearNumExpr v = cplex.linearNumExpr();
            for(int i = 0; i < qtdBuffers; i++){  v.addTerm( (double) taxaDeAcessoBuffer[ i ], x[ i ][ j ]); }
            cplex.addLe( v,cplex.prod(larguraBandaMemoria[ j ], cplex.prod(y[ j ], periodoReduzido)));
         }
         for(int j = 0;j<qtdMemoria;j++){         // Terceira equação
            IloLinearNumExpr v = cplex.linearNumExpr();
            for(int i = 0; i < qtdBuffers; i++){  v.addTerm( (double)qtdPortasBuffer[ i ], x[ i ][ j ]); }
            cplex.addLe(v, cplex.prod(qtdPortasMemoria[ j ], y[ j ]));
         }
         for(int i = 0; i < qtdBuffers; i++){ cplex.addEq(cplex.sum(x[i]), 1); }          // Quarta equação

         IloLinearNumExpr expr = cplex.linearNumExpr(); // Função objetivo
         for (int i = 0; i < qtdMemoria; ++i) {
              expr.addTerm(y[i], 1.);
         }
         cplex.addMinimize(expr);
         startTime = cplex.getCplexTime();
         result = cplex.solve();
         endTime = cplex.getCplexTime();
         if(result){
            countMemories = (int)cplex.getObjValue();
            for(int j=0;j<qtdMemoria;j++){
               if(cplex.getValue(y[j])==1){
                  listaDeMemoriaEscolhidas.add(capacidadeMemoria[j]);
               }
            }
         }else{
            System.out.println("Infeasible");
         }
      } catch (IloException e) { e.printStackTrace(); }
      return result;
   }
```

APÊNDICE C – Código-fonte do Gerador de Entradas

GeradorEntradas.java

```
/*
 * GeradorEntradas.java
 *
 * Gera as entradas que representam os buffers que serao utilizadas pelos algoritmos de alocacao em bancos de memoria.
 * Os arquivos gerados estao organizados da seguinte forma:
 *
 * 2            // numero de problemas que virao no arquivo
 * 2:2          // numero de buffers do proximo problema e numero de memorias separados por ':'
 * 400000:2:100 // descricao dos 3 parametros do buffer separados por ':' <tamanho(KB)>:<numero_de_portas>:<taxa_de_acesso(KB/s)>
 * 250000:1:300
 * 800000:1:6500// descricao dos 3 parametros do banco de memoria separados por ':' <capacidade(KB)>:<numero_de_portas>:<largura_de_banda(KB/s)>
 * 800000:1:6500
 * 3:2
 * 3550000:2:200
 * 2345000:1:500
 * 125000:2:100
 * 800000:16:500
 * 800000:16:500
 */

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Random;

public class GeradorEntradas {

    // caracteristicas possiveis para os bancos de memoria
    private static final int MIN_NBM = 2;       // menor numero de bancos de memoria que sera considerado.
    private static final int MAX_NBM = 6;       // maior numero de bancos de memoria que sera considerado.
    private static final int MIN_CM =  8;       // menor capacidade de memoria que sera considerada em KB (256MB).
                                                // utilizar expoente da base por exemplo 0 = 2 ^ 0 = 1 KB; 4 = 2 ^ 4 = 16 KB; etc...
    private static final int MAX_CM = 23;       // maior capacidade da memoria que sera considerada em KB (8GB).
    private static final int NUM_MT = 2;        // numero de tipos de memoria que serao utilizados. eg. DDR2-800 e DDR3-1600.
    // menor largura de banda da memoria que sera considerada em KB/s (6400 MB/s - DDR2-800).
    // maior largura de banda da memoria que sera considerada em KB/s (12800 MB/s - DDR3-1600).
    private static final int[] LB_TIPO = {6553600,13107200};
    private static final int MIN_NP = 15;       // menor numero de portas que sera considerado.
    private static final int MAX_NP = 15;       // maior numero de portas que sera considerado.
    private static final int NUM_PROB_DEF = 100;

    // caracteristicas possiveis para os buffers
    private static final int MIN_NB = 5;        // menor numero de buffers que sera considerado.
    private static final int MAX_NB = 50;       // maior numero de buffers de memoria que sera considerado.

    private static final int MAX_TB = 1048576;  // maior tamanho de buffer possivel (KB). os menores serao fracao dele.
    private static final int NUM_TB = 4;        // quantidade de capacidades diferentes de buffers

    private static final int MAX_TA = 16000;    // maior taxa de acesso que o buffer podera ter a 1MHZ (KB/s). as menores taxas serao fracao desta.
    private static final int NUM_TA = 5;        // quantidade de taxas de acesso diferentes diferentes de buffers


    private static final int MIN_NPB = 1;       // menor numero de buffers que sera considerado.
    private static final int MAX_NPB = 2;       // maior numero de buffers de memoria que sera considerado.

    // vetores de possibilidades de valores para os buffers
    int ptb[] = new int[NUM_TB];            // vetor de possibilidade de valor para tamanho do buffer
    int pnpb[] = new int[MAX_NPB-MIN_NPB+1];    // vetor de possibilidade de valor para numero de portas do buffer
    int pta[] = new int[NUM_TA];                // vetor de possibilidade de valor para taxa de acesso

    // vetores de possibilidades de valores para as memorias
    int pcm[] = new int[MAX_CM-MIN_CM+1];       // vetor de possibilidade de valor para capacidade de memoria
    int pnpm[] = new int[MAX_NP-MIN_NP+1];      // vetor de possibilidade de valor para numero de portas maximo
    int plb[] = new int[NUM_MT];                // vetor de possibilidade de valor para largura de banda

    private ArrayList<String> arquivoLinhas = new ArrayList<String>();

```

GeradorEntradas.java

```
71      private Path arquivo = Paths.get("entrada_buffer.txt");
72
73
74      public static void main(String args[]) {
75          int numProblemas = 0;   // especifica o numero de problemas que serao gerados no arquivo
76          int numBuffers = 0;
77          int numMemorias = 0;
78          int totalTamanhoBuffers = 0;
79          int totalPortasBuffers = 0;
80          int totalLarguraBandaBuffers = 0;
81
82          int totalCapacidadeMemoria = 0;
83          int totalPortasMemoria = 0;
84          int totalLarguraBandaMemorias = 0;
85
86          String tempValues = "";
87
88          GeradorEntradas gerador = new GeradorEntradas();
89          gerador.carregaVetoresPossibilidades();
90          // captura o numero de problemas da entrada do usuario
91          if (args.length > 0) {
92              numProblemas = Integer.parseInt(args[0]);
93          }
94          // utiliza o valor padrao para o numero de problemas que serao gerados
95          else {
96              numProblemas = NUM_PROB_DEF;
97          }
98          // escreve no buffer do arquivo o numero de problemas
99          gerador.arquivoLinhas.add(numProblemas + "");
100         // laco para geracao de todos os problemas
101         for (int i = 0; i < numProblemas; i++) {
102             // gera um numero aleatorio de numero de buffers
103             numBuffers = GeradorEntradas.randInt(MIN_NB, MAX_NB);
104             // gera um numero aleatorio de numero de memorias
105             numMemorias = GeradorEntradas.randInt(MIN_NBM, MAX_NBM);
106             // escreve no buffer do arquivo o numero de problemas
107             tempValues = numBuffers + ":" + numMemorias + '\n';
108             for (int j = 0; j < numBuffers; j++) {
109                 int tamanhoBuffer = gerador.ptb[GeradorEntradas.randInt(0, (gerador.ptb.length-1))];
110                 totalTamanhoBuffers += tamanhoBuffer;
111                 tempValues += tamanhoBuffer + ":";
112                 int portasBuffer = gerador.pnpb[GeradorEntradas.randInt(0, (gerador.pnpb.length-1))];
113                 totalPortasBuffers += portasBuffer;
114                 tempValues += portasBuffer + ":";
115                 int larguraBandaBuffer = gerador.pta[GeradorEntradas.randInt(0, (gerador.pta.length-1))];
116                 totalLarguraBandaBuffers += larguraBandaBuffer;
117                 tempValues += portasBuffer + "\n";
118             }
119             for (int j = 0; j < numMemorias; j++) {
120                 int tamanhoMemoria = gerador.pcm[GeradorEntradas.randInt(0, (gerador.pcm.length-1))];
121                 totalCapacidadeMemoria += tamanhoMemoria;
122                 tempValues += tamanhoMemoria + ":";
123                 int portasMemoria = gerador.pnpm[GeradorEntradas.randInt(0, (gerador.pnpm.length-1))];
124                 totalPortasMemoria += portasMemoria;
125                 tempValues += portasMemoria + ":";
126                 int larguraBandaMemoria = gerador.plb[GeradorEntradas.randInt(0, (gerador.plb.length-1))];
127                 totalLarguraBandaMemorias += larguraBandaMemoria;
128                 // caso seja a ultima memoria nao coloca o '\n' para nao termos uma linha em branco entre
    problemas
129                 if (j == (numMemorias-1)) {
130                     tempValues += larguraBandaMemoria;
131                 }
132                 else {
133                     tempValues += larguraBandaMemoria + "\n";
134                 }
135             }
136             // verifica se a soma do tamnho dos buffers, da quantidade de portas e da largura de banda e maior
137             // que o disponivel em todos os buffers
138             if(totalTamanhoBuffers > totalCapacidadeMemoria || totalPortasBuffers > totalPortasMemoria ||
139                totalLarguraBandaBuffers > totalLarguraBandaMemorias) {
140                 // repete o laco
141                 i--;
142             }
143             else {
144                 // grava dados gerados nos arquivos
145                 gerador.arquivoLinhas.add(tempValues);
146             }
```

GeradorEntradas.java

```
147             totalTamanhoBuffers = 0;
148             totalPortasBuffers = 0;
149             totalLarguraBandaBuffers = 0;
150             totalCapacidadeMemoria = 0;
151             totalPortasMemoria = 0;
152             totalLarguraBandaMemorias = 0;
153             tempValues = "";
154         }
155         try {
156             Files.write(gerador.arquivo, gerador.arquivoLinhas, StandardCharsets.UTF_8);
157         } catch (IOException e) {
158             e.printStackTrace();
159         }
160     }
161 
162     public static int randInt(int min, int max) {
163         Random rand = new Random();
164         int randomNum = rand.nextInt((max - min) + 1) + min;
165         return randomNum;
166     }
167 
168     public void carregaVetoresPossibilidades() {
169         // carrega vetores de possibilidades de valores para os buffers
170         for (int i = 0; i < ptb.length; i++) {
171             if(i == 0) {
172                 ptb[i] = MAX_TB;            // vetor de possibilidade de valor para tamanho do buffer
173             }
174             else if(i == 1) {
175                 ptb[i] = MAX_TB/2;          // vetor de possibilidade de valor para tamanho do buffer
176             }
177             else {
178                 // vetor de possibilidade de valor para tamanho do buffer
179                 ptb[i] = (int)((double) MAX_TB/(Math.pow(2.0, Math.pow(2.0, (double) i))));
180             }
181         }
182         for (int i = 0; i < pnpb.length; i++) {
183             pnpb[i] = MIN_NPB + 1;                      // vetor de possibilidade de valor para numero de portas
    do buffer
184         }
185         for (int i = 0; i < pta.length; i++) {
186             if(i == 0) {
187                 pta[i] = MAX_TA;            // vetor de possibilidade de valor para taxa de acesso
188             }
189             else {
190                 // vetor de possibilidade de valor para taxa de acesso
191                 pta[i] = (int)((double) MAX_TA/(Math.pow(2.0, Math.pow(2.0, (double) (i-1)))));
192             }
193         }
194 
195         // vetores de possibilidades de valores para as memorias
196         for (int i = 0; i < pcm.length; i++) {
197             pcm[i] = 1 << (MIN_CM + i);                 // vetor de possibilidade de valor para capacidade de
    memoria
198         }
199         for (int i = 0; i < pnpm.length; i++) {
200             pnpm[i] = MIN_NP + i;           // vetor de possibilidade de valor para capacidade de memoria
201         }
202         for (int i = 0; i < plb.length; i++) {
203             plb[i] = LB_TIPO[i];            // vetor de possibilidade de valor para capacidade de memoria
204         }
205     }
206 }
```