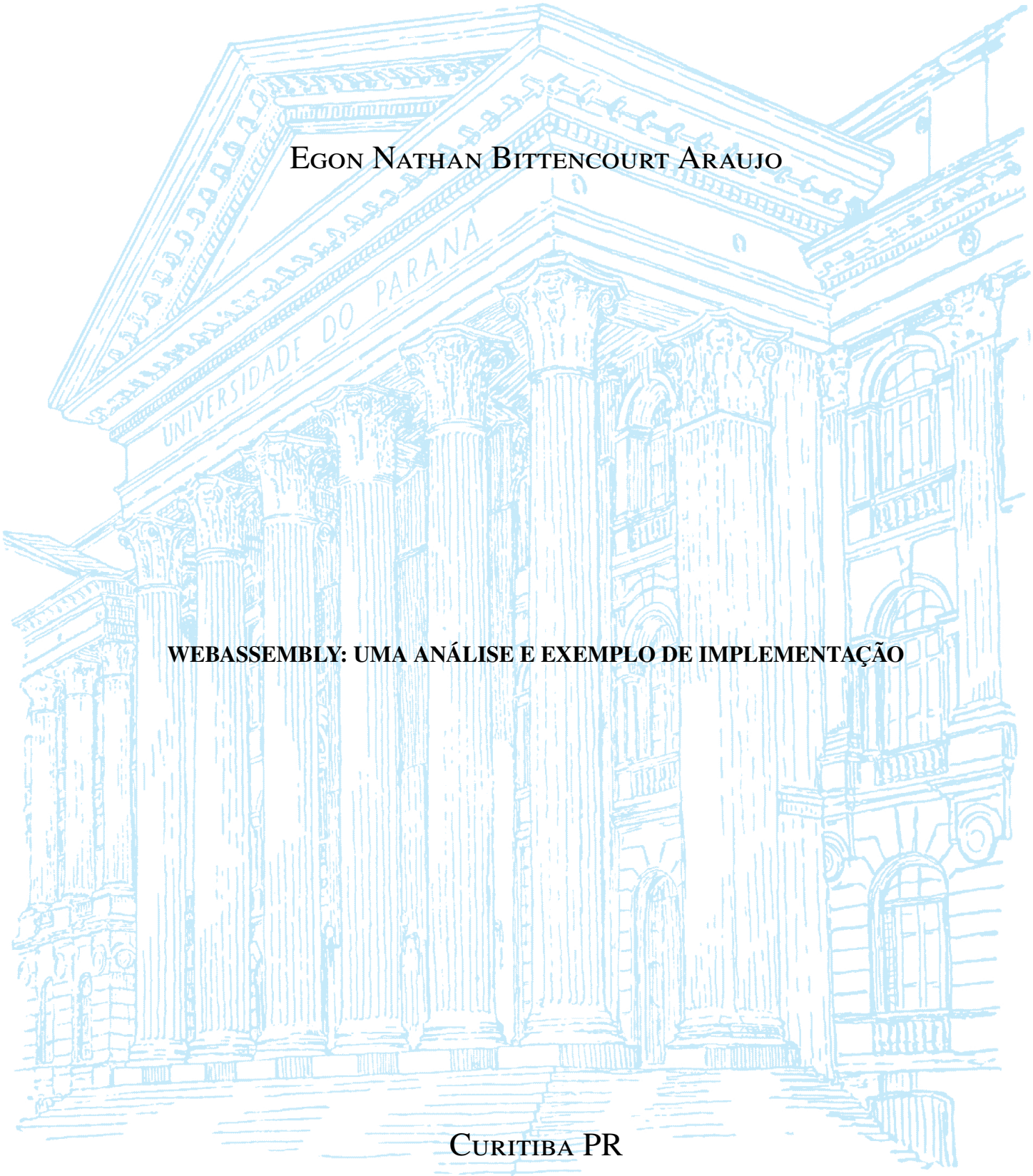
Universidade Federal do Paraná

Egon Nathan Bittencourt Araujo



WEBASSEMBLY: UMA ANÁLISE E EXEMPLO DE IMPLEMENTAÇÃO

Curitiba PR
2018

Egon Nathan Bittencourt Araujo

WEBASSEMBLY: UMA ANÁLISE E EXEMPLO DE IMPLEMENTAÇÃO

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Bruno Müller Junior.

Curitiba PR
2018

Abstract

The WebAssembly is a technology that aims to improve the performance of applications on the internet. Announced for the first time on june 17th of 2015, but only releasing a minimum viable product on march of 2017, this language tries to be faster than older solutions. With the goal of having a better performance than other solutions that run on client-side, WebAssembly features a bytecode that can be interpreted faster than its predecessors. By not being a textual language and having the ability to be generated by compilers, the generated code uses less space and can be optimized during the compilation phase. The language aims to run with native performance, that means, with the same performance as codes compiled to run directly on computers. With the goal of verifying this affirmation, this work implements an application and makes an analysis of the results obtained. A common function of image processing is used, the linear filtering, as the test application. This was compiled to the native platform, WebAssembly and asm.js. The results show that WebAssembly has a worse performance than the native application. In comparison with asm.js, WebAssembly didn't execute slower in any case, using only more memory in a test case using a specific browser. WebAssembly represents an improvement over actual solutions, but is still far from the native performance.

Keywords: WebAssembly, linear filtering, asm.js, image processing, client-side.

Resumo

O WebAssembly é uma tecnologia que visa melhorar o desempenho de programas na internet. Sendo anunciado pela primeira vez dia 17 de junho de 2015, mas apenas divulgando um produto viável mínimo em março de 2017, essa linguagem busca ser mais rápida que as soluções anteriores. Com o objetivo de possuir um melhor desempenho que outras soluções que atuam do lado do cliente, o WebAssembly utiliza um bytecode e pode ser interpretado de forma mais rápida que seus antecessores. Por não ser uma linguagem textual e poder ser gerada por compiladores, o código gerado consome menos espaço e pode ser otimizado em tempo de compilação. A linguagem tem o intuito de rodar com desempenho nativo, ou seja, com o mesmo desempenho de códigos compilados para rodarem diretamente em computadores. Com o objetivo de verificar essa afirmação, este trabalho implementa uma aplicação e realiza a análise dos resultados. Uma função comum de processamento de imagens, a filtragem linear, é utilizada como aplicação para os testes. Esta foi compilada para a plataforma nativa, WebAssembly e asm.js. Os resultados mostram que o WebAssembly apresenta um desempenho pior que a aplicação nativa. Em comparação ao asm.js, o WebAssembly não executou mais lento em nenhum caso, gastando mais memória apenas em um caso de teste utilizando um navegador específico. O WebAssembly representa uma melhora sobre as soluções atuais, mas ainda está longe do desempenho nativo.

Palavras-chave: WebAssembly, filtragem linear, asm.js, processamento de imagens, lado do cliente.

Lista de Figuras

Lista de Tabelas

Lista de acrônimos

FAQ	Questões frequentemente respondidas (do inglês *Frequently Answered Questions*)
JS	JavaScript
VM	Máquina Virtual (do inglês *Virtual Machine*)
WASM	WebAssembly
PGM	Mapa de Cinza Portátil (do inglês *Portable Gray Map*)

Sumário

1 Introdução

A internet é algo quase indispensável para a sociedade atual. Em vários lugares e situações é possível encontrar alguém que a esteja utilizando. Somente no período de 2005 a 2013, ocorreu um aumento de 264% no número de usuários que utilizam a internet ao redor do mundo, de 15.8% para 37.9% da população mundial [9]. Em 2018, esse número já subiu para 55.1%[1], demonstrando o crescimento contínuo dessa tecnologia.

Um modelo bastante usado e que forma a base de grande parte de uso da internet é o chamado 'Modelo Cliente-Servidor'. Esse modelo consiste em dividir a aplicação em duas partes, chamadas Servidores e Clientes. O lado do Servidor trabalha com processamento, geralmente pesado ou em grande quantidade. O lado do Cliente envia requisições para os Servidores, e processam o resultado dessas [15]. As tecnologias no lado do Cliente ficaram continuamente melhores, tanto em termos de velocidade de execução quanto tamanho da versão final do produto. Uma das ferramentas mais recentes é o WebAssembly, uma tecnologia anunciada em 2015 [20].

Com o objetivo de trazer aplicações escritas pensando na execução nativa, diversas soluções surgiram para compilar estas aplicações para JavaScript. Como JS não foi pensado com esse contexto, acessos direto na memória e ao sistema de arquivos eram inviáveis, além de certas otimizações. Com o objetivo de remover estes problemas, o *asm.js*[2] foi criado. Essa linguagem consiste de um subconjunto de funções do JS e ainda apresentava problemas como a modificação da memória diretamente e a representação textual. O WebAssembly surge como uma opção à essa tecnologia, com uma representação binária que permite um menor tamanho de arquivo e uma interpretação mais rápida.

Neste trabalho será apresentado um contexto da criação da linguagem WebAssembly, os problemas que ela propõe resolver e as vantagens de se usar ela ao invés das tecnologias empregadas atualmente. Para isso, foi desenvolvido um software para comparação entre *asm.js*, WebAssembly e *softwares* rodando diretamente no computador, chamados de nativos. Uma função comum em *softwares* de edição de imagens foi utilizada para as análises, permitindo a análise de desempenho e a viabilidade de se escrever *softwares* desta área para a internet.

Para atingir esse objetivo, esse texto foi dividido em 6 Capítulos. No Capítulo 2 são apresentados os conceitos necessários para o entendimento do trabalho e uma breve explicação da história e motivação da criação do WebAssembly. No Capítulo 3 ocorre um levantamento dos trabalhos relacionados e uma análise dos editores de imagem na internet atualmente. No Capítulo 4 é explicado como os testes foram realizados e quais os objetivos desse trabalho. No Capítulo 5 são detalhados o algoritmo utilizado para os testes e como os valores foram obtidos. No Capítulo 6 é apresentado uma análise dos resultados obtidos e a comparação entre eles. No Capítulo 7 ocorre a conclusão deste trabalho, resumindo os resultados e abordando como futuros trabalhos podem se beneficiar deste.

[1] `https://www.internetworldstats.com/stats.htm`
[2] `http://asmjs.org/`

2 Fundamentação teórica

Neste capítulo serão apresentados o conceito do Modelo Cliente-Servidor, a história da criação do WebAssembly, o que são formatos de imagem e o conceito de processamento de imagens. O conhecimento desses temas são necessários para a compreensão do texto.

Na Seção 2.1, será explicado como funciona e o que é o Modelo Cliente-Servidor. Na Seção 2.2 são explorados os conceitos de linguagens interpretadas e linguagens compiladas, apresentando a diferença entre elas. Na Seção 2.3 a linguagem WebAssembly, o compilador encontrado e uma breve história do motivo da criação da linguagem são detalhados. Na Seção 2.4 o conceito de arquivos em um sistema operacional é explicado e o formato das imagens utilizadas neste trabalho é detalhado. Na Seção 2.5 são documentados as operações utilizadas para o trabalho: as operações morfológicas e a filtragem linear, apresentando também uma breve explicação da área de processamento de imagens.

2.1 Modelo Cliente-Servidor

Como explicado brevemente no Capítulo 1, o Modelo Cliente-Servidor forma a base de grande parte do uso da rede. Nesse Modelo, a aplicação é dividida em duas partes, os Servidores e os Clientes. A Figura 2.1 mostra como esse Modelo é idealizado, onde diversos Clientes acessam os Servidores por meio da internet.

Os Servidores, detalhados na Subseção 2.1.1, em geral possuem alto poder computacional, e são comumente utilizados para armazenar uma grande quantidade de dados ou realizar um processamento demorado. Os Clientes, detalhados na Subseção 2.1.2, são os computadores dos usuários que acessam o serviço, em geral com menos poder computacional, e tem como função o processamento dos dados recebidos do Servidor e a sua visualização [15].

2.1.1 Servidor

Os Servidores são um conjunto de um ou mais computadores, que normalmente possuem um alto poder computacional. Essas máquinas receberão requisições dos Clientes, que devem ser processadas e respondidas. Essas respostas podem ser um conjunto de dados ou o resultado de algum processamento [15]. Em geral, os Servidores precisam estar sempre disponíveis para serem requisitados pelos Clientes.

2.1.2 Cliente

Os Clientes são um conjunto de um ou mais computadores, que geralmente possuem um menor poder computacional. Os Clientes mandam requisições aos Servidores sempre que precisam de alguma informação que eles possuem, como uma página *web*. Outra tarefa dos computadores Clientes é o processamento e a visualização da resposta obtida [15]. Hoje em dia

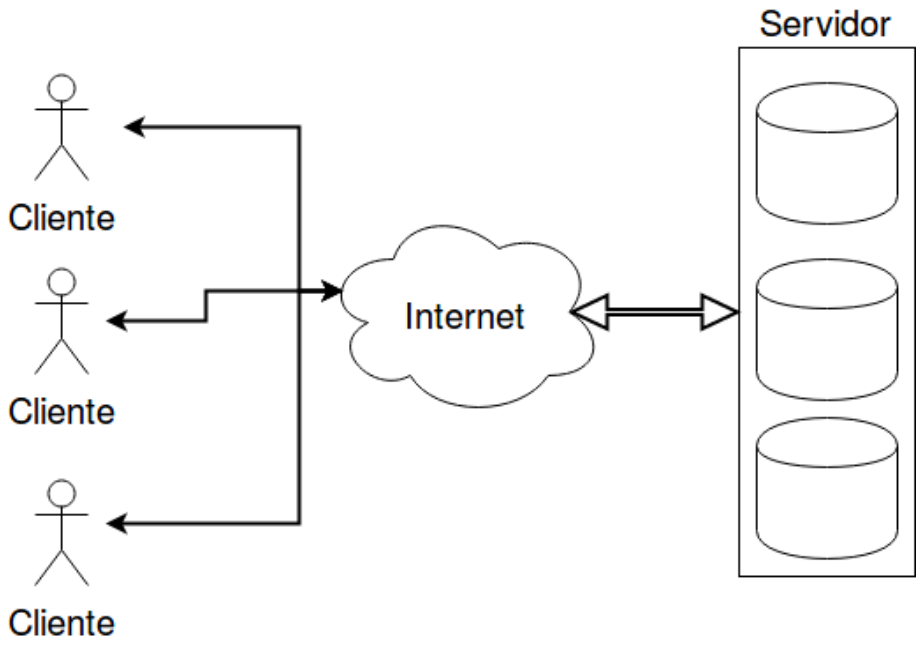


Figura 2.1: Modelo Cliente-Servidor

temos, como exemplo, jogos que são transmitidos pelos Servidores, porém são os Clientes que controlam o processamento e a visualização destes.

2.2 Linguagens interpretadas e compiladas

Os navegadores podem ser considerados os Clientes do Modelo acima. A maioria dos navegadores atuais suporta apenas uma linguagem de processamento, que é interpretada[1], o *JavaScript*(JS)[2].

As linguagens interpretadas são executadas da forma como foram escritas. Essas linguagens precisam de um interpretador para serem executadas. O interpretador converte para um formato executável, executa e otimiza o código ao mesmo tempo. Os erros de sintaxe são detectados durante a conversão para o formato executável.

Uma outra categoria das linguagens são as chamadas compiladas. As linguagens compiladas precisam passar por um compilador antes de serem executadas. Esse compilador pode otimizar o código para a execução e o converte para um formato executável na plataforma alvo. A detecção de erros de sintaxe no programa ocorre durante a compilação [7].

Os interpretadores geralmente convertem uma linha por vez, e necessitam converter novamente a cada instante que o programa alcança aquela linha[3]. Essa limitação causa um *overhead* grande no programa pois cada linha é convertida e, caso precise ser executada novamente, precisa ser convertida novamente. Linguagens compiladas não necessitam dessa re-conversão e podem ser diretamente executadas. Outra limitação das linguagens interpretadas é a execução do código como foi escrito, enquanto as linguagens compiladas geram um código otimizado para a execução.

[1] https://eager.io/blog/a-brief-history-of-weird-scripting-languages/
[2] https://www.javascript.com/
[3] https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zappldev/zappldev_85.htm

A Figura 2.2 mostra qual processamento está ocorrendo em dado tempo de uma aplicação. A proporção dos elementos são meramente ilustrativas pois esta e a relação entre cada área varia entre interpretadores e ambientes de execução. Podemos ver que a linguagem compilada e interpretada acabam levando um mesmo tempo de carregamento na memória. Após o carregamento, ocorre a execução do código compilado sem interrupções, exceto no caso de o código gerar uma interrupção como acesso ao disco ou entrada de teclado. No código interpretado, temos uma divisão deste tempo nas seções de conversão para código executável, execução e otimização do código. Essas etapas adicionais causam as linguagens interpretadas a executarem geralmente com um desempenho mais baixo que as compiladas.

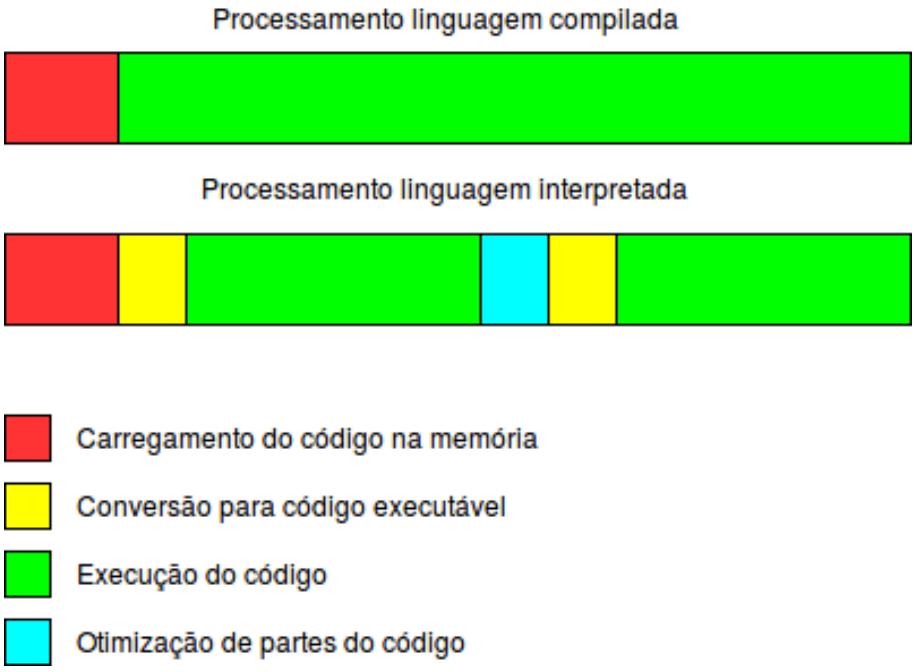


Figura 2.2: Figura com processamento de linguagens compiladas e interpretadas

2.3 Web Assembly

Se utilizando do Modelo Cliente-Servidor, algumas aplicações, antes somente disponíveis quando instaladas no computador, puderam ser disponibilizadas via internet. Programas como editores de áudio e vídeo, visualizadores 3D, e jogos são alguns desses exemplos. Estes podem ser executados no lado do Servidor como, por exemplo, enviando uma imagem para o Servidor e apenas controlando quais alterações são feitas sobre ela. Mas também podem ser executados no lado do Cliente, com o Servidor transmitindo o código completo para o Cliente processar. O *JavaScript* é a única linguagem de processamento que vem por padrão na maioria dos navegadores atuais[1]. Criada em 1995 [10], existem alguns novos problemas que não foram pensados em sua criação, além de novas funcionalidades terem que respeitar a definição inicial da linguagem.

Uma necessidade que surgiu com esse avanço foi a passagem de aplicações nativas para a internet. No intuito de trazer esses programas para a internet, muitas soluções criaram compiladores de linguagens com memória insegura para JavaScript. Na definição da linguagem, o JS limita o acesso a memória do sistema para evitar falhas de segurança, o que se mostra uma limitação na compilação de linguagens que acessam e alteram a memória diretamente. Códigos

em C/C++, por exemplo, fazem uso de ponteiro para a modificação da memória e não necessitam de um coletor de lixo, além de serem naturalmente rápidos [6].

Com o objetivo de tirar esta limitação, engenheiros das quatro maiores empresas que desenvolvem navegadores *web* se juntaram para criar um *bytecode* de baixo nível, chamado de WebAssembly [6]. Esta tecnologia busca uma representação compacta, validação e compilação eficientes, e uma execução segura com *overhead* mínimo ou inexistente. O JavaScript falha nessas questões pois limita o acesso a memória, impedindo certas operações, e possui uma representação textual que ocupa mais espaço que um binário e leva mais tempo para ser interpretado. A alteração nos navegadores seria mínima, dado que já existe um interpretador de JavaScript por padrão, e as empresas poderiam se utilizar deste para criar o interpretador de WebASsembly.

Devido ao WebAssembly ser uma abstração do *hardware*, esta oferece uma opção que seja independente de linguagem, *hardware* ou plataforma utilizados, permitindo a utilização fora do contexto da Internet. A definição permite que seja construído um interpretador de WebAssembly diretamente para o sistema nativo, retirando a necessidade de um navegador. Outra analogia seria a execução de arquivos WebAssembly diretamente nos navegadores, mesmo sem utilizar a internet.

Na Subseção 2.3.1 um compilador para WebAssembly é descrito. Na Subseção 2.3.2 são analisadas as linguagens criadas anteriormente para serem executadas nos navegadores e a razão de criação do WASM.

2.3.1 Compilador

Em busca de ferramentas que permitiam o uso de WebAssembly, foi realizada uma pesquisa de compiladores de C/C++ para esta linguagem. Essa busca, realizada em junho de 2018, revelou apenas o Emscripten[4]. Ele utiliza a tecnologia LLVM, que se classifica como uma coleção modular e reutilizável de compiladores e conjuntos de *toolchains*[5].

As bibliotecas LLVM contém um otimizador independente de código fonte e plataforma em conjunto com uma geração de código para diferentes CPUs. Essas bibliotecas são construídas sobre o conceito de uma linguagem intermediária de representação chamada de LLVM IR[6].

Com o resultado gerado dessas operações, outro compilador chamado de *binaryen* compila o código gerado em LLVM IR para WebAssembly. Ele também gera uma pagina em JavaScript e a integração entre o código compilado e a página gerada[7].

Na Figura 2.3, adaptada do site oficial do Emscripten[8], está demonstrado como o compilador funciona. Os arquivos fonte, no caso .c e .cpp, são dados de entrada para o compilador. O compilador gera um código LLVM intermediário, e executa as otimizações necessárias. Também é possível passar configurações via argumentos ou via arquivo. O resultado do processo podem ser arquivos HTML, WebAssembly, asm.js ou JavaScript.

É importante ressaltar que a entrada é um programa escrito em C ou C++, e a saída são programas com funcionalidades equivalentes. Como exemplo, considere um programa ‘mmc.c’, escrito na linguagem C, que computa o mínimo múltiplo comum entre os números. O compilador vai gerar um código intermediário em LLVM que é equivalente ao código do arquivo ‘mmc.c’. Esse código intermediário é compilado para ‘mmc.asm.js’, ‘mmc.wasm’ ou ‘mmc.js’ de acordo com as especificações dadas ao compilador. Os arquivos gerados realizam a mesma

[4] http://kripken.github.io/emscripten-site/index.html
[5] http://kripken.github.io/emscripten-site/docs/introducing_emscripten
[6] https://llvm.org/
[7] https://kripken.github.io/emscripten-site/docs/compiling/WebAssembly.html
[8] http://kripken.github.io/emscripten-site/_images/EmscriptenToolchain.png

funcionalidade do arquivo ‘mmc.c’. O Emscripten ainda pode gerar uma página HTML e JS que facilita a interação com os programas gerados, oferecendo uma interface pronta para ser utilizada.

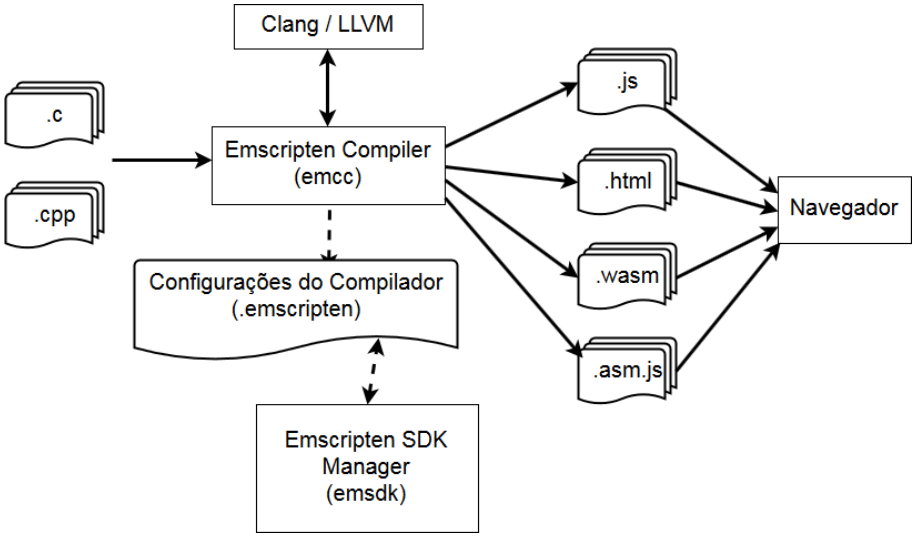


Figura 2.3: Fluxograma exemplificando o compilador Emscripten

2.3.2 Objetivo de criação

Devido ao JavaScript(JS) ser uma linguagem interpretada, esta possui problemas como a execução de código não otimizado [14]. Para resolver esse e outros problemas, o asm.js[9] foi criado. O asm.js consiste de um subconjunto restrito do JavaScript, que pode ser interpretado de maneira eficiente pelo navegador [8]. A ideia do asm.js foi criar um conjunto de funções e operações que conseguem rodar normalmente em uma Máquina Virtual (do inglês *Virtual Machine*, VM) do JS[10]. A principal diferença surge para os navegadores que se utilizam desse subconjunto limitado de operações para inferir informações sobre o código, tornando a execução mais rápida. Isso permitiu que aplicações feitas em asm.js fossem compiladas e otimizadas antes de sua execução.

Uma VM que consegue identificar e interpretar esse subconjunto o executa de maneira eficiente. Isso ocorre pois asm.js tem uma abstração próxima às VMs, pois possui suporte a carregamento e armazenamento de variáveis em uma pilha, operações em inteiro e ponto flutuante, definições de função de primeira ordem e ponteiro de funções. Essa característica permite que linguagens com memória insegura, como C[11] ou C++[12], sejam compiladas para a internet [8].

Esse subconjunto do JavaScript ainda possui pontos que poderiam ser melhorados, pois está limitado pela linguagem. O WebAssembly, utilizando o formato binário, pode ser interpretado mais rápido que o JS (em alguns experimentos, até vinte vezes mais rápido)[13]. Comparando a execução das linguagens nos celulares atuais, quantidades grandes de código asm.js demoram de 20 a 40 segundos somente para serem convertidas para um formato executável. Um formato binário permite uma análise rápida e o uso de técnicas que permitem a execução do código recebido antes de se ter acesso ao conteúdo completo. Este novo formato permitiria um

[9] http://asmjs.org
[10] http://asmjs.org/faq.html
[11] https://www.geeksforgeeks.org/c-language-set-1-introduction/
[12] https://www.programiz.com/cpp-programming
[13] https://github.com/WebAssembly/design/blob/master/FAQ.md

desempenho próximo do nativo, além de espaço para a inclusão de novas funcionalidades de maneira facilitada.

O WebAssembly tem como objetivo ser uma linguagem segura, rápida, portátil e de baixo nível na internet. As antigas tentativas de preencher todos estes quesitos, desde ActiveX[14], passando por Native Client[15], até a atual asm.jsm, não apresentaram toda a base que uma linguagem de baixo nível deveria ter: uma semântica segura, rápida e portátil e uma representação segura e eficiente [6].

Essas características tornaram possíveis algumas aplicações antes inviáveis ou limitadas. Exemplos destas incluem jogos, editores de imagem ou vídeo, criptografia, aplicações com músicas (utilizando *streaming* e *caching*), aplicações ponto a ponto, entre outras[16]. Buscando verificar tais afirmações feitas no anúncio da linguagem, este trabalho implementa um editor de imagem. Esta aplicação permite a análise do desempenho nas áreas de operações com matrizes e leitura e escrita de arquivos para a memória [11].

2.4 Formatos de Arquivos

A aplicação a ser utilizada é um editor de imagens. Para processarmos qualquer tipo de imagem, é preciso definir qual seria o formato utilizado por ela. Um formato de arquivo inclui qualquer forma de codificar dados para criar um arquivo digital, e são necessários pois um computador armazena e processa somente dados em binário [16]. Esses dados podem representar diversas informações, e possuir um formato específico permite ao sistema operacional representar ele de forma correta.

Como este trabalho propõem um editor de imagens, foram analisados formatos de arquivos utilizados para representar imagens. Apesar de os formatos mais comuns serem JPG/JPEG, GIF, PNG, e SVG [17], este trabalho utiliza o formato PGM desenvolvido pela biblioteca Netpbm, que se classifica como "um pacote de programas gráficos e uma biblioteca de programação". Visando reduzir o número de bibliotecas externas necessárias, esse formato permite a criação de uma função própria de leitura do arquivo visando a redução de código e otimização [12]. Este formato foi escolhido pela simplicidade e facilidade de manipulação e a não ocorrência de casos específicos como controle de transparência.

2.4.1 PGM

O formato PGM de "Portable Gray Map"(Mapa de Cinza Portátil) foi desenvolvido para ser extremamente fácil de aprender e escrever programas que o utilizem. Esse formato representa uma imagem em escalas de cinza de forma direta, onde cada *byte* do arquivo representa um *pixel* [13].

O formato consiste em um cabeçalho composto por: 1) um "**Número Mágico**" que identifica o formato, no caso do PGM esse número são os caracteres "P5"; 2) a **Largura** da imagem, formatada como números decimais em ASCII; 3) a **Altura** da imagem, também formatada como números decimais em ASCII; 4) o **Valor Máximo** que cada *pixel* pode conter, entre 0 e 65536. Cada um dos elementos deve ser separado por um ou mais **Espaços em Branco** (definidos como espaços, 'TAB's, ou quebras de linha).

Logo após o cabeçalho e exatamente um **Espaço em Branco** começam os dados da imagem. Estes são **Altura**×**Largura** *bytes* em sequência. Cada elemento representa um valor

[14] `http://msdn2.microsoft.com/en-us/library/aa751972(VS.85).aspx`
[15] `https://developer.chrome.com/native-client/overview`
[16] `https://webassembly.org/docs/use-cases/`

indicando a escala de cinza do *pixel*. Cada valor deve ser representado com um *byte*, se o **Valor Máximo** for menor que 256, e deve ser representado por dois *bytes* caso o **Valor Máximo** for maior ou igual a 256.

O exemplo na Figura 2.4 apresenta uma imagem 3x3 e seu arquivo PGM. Os valores dos *pixels* na imagem são mostrados sobre cada *pixel*. Os valores indicados abaixo da terceira linha no arquivo PGM, que possui o **Valor Máximo**, são as representações em hexadecimal de cada *byte*. Na representação textual foram adicionados quebras de linha e espaços em branco para facilidade de leitura, porém logo depois do **Valor Máximo** e um espaço em branco, tais caracteres seriam lidos como *bytes* e representariam um *pixel*.

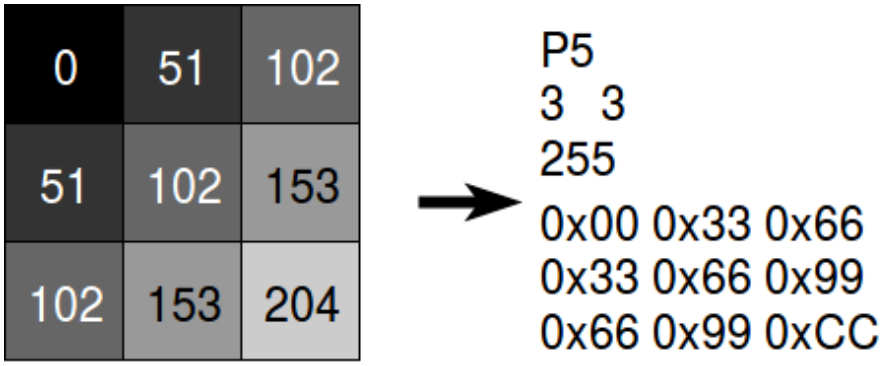


Figura 2.4: Exemplo de uma imagem PGM

2.5 Processamento de Imagens

A sequência de linhas e colunas contendo as informações da imagem também pode ser vista como uma função $f(x, y)$ em cima de um plano. Onde (x, y) são coordenadas espaciais, e o valor de f para uma coordenada é a intensidade do nível de cinza naquele ponto. Quando todos os x, y e os valores de f são finitos e discretos, chamamos de uma imagem digital. O processamento de imagens é a área que processa essas imagens digitais por meio de computadores [3]

Duas técnicas para o processamento de imagens utilizados na implementação serão apresentadas. A primeira, descrita na Subseção 2.5.1, é chamada de operações morfológicas, com origem na matemática. A segunda, descrita na Subseção 2.5.2, é chamada de filtragem linear, e serve como base para diversas aplicações na área de processamento de imagens.

2.5.1 Operações morfológicas

As primeiras operações com imagens implementadas neste trabalho vieram da área da matemática chamada de operações morfológicas. A morfologia apresenta uma maneira poderosa e unificada de resolver problemas de processamento de imagens pois utiliza a linguagem da teoria de conjuntos. Um modo de ver uma imagem preto e branco, utilizando o conceito acima, é como um conjunto de coordenadas no espaço Z^2, em que $a \in A | a = (x, y)$, com cada elemento a representando um *pixel*. Este *pixel* pode ser interpretado como preto ou branco, de acordo com a convenção utilizada. Uma imagem cinza pode ser definida no espaço Z^3, sendo os dois primeiros elementos as coordenadas do *pixel*, e o terceiro elemento o valor referente à escala de cinza do *pixel* [4]. A erosão e a dilatação são operações matemáticas chamadas de morfológicas, e são definidas para os conjuntos definidos acima.

As duas operações morfológicas se utilizam do deslocamento de imagens. O deslocamento consiste na movimentação de uma imagem, tal que todos os pontos da imagem sejam deslocados de acordo com a posição indicada. Considere uma imagem $I \in Z^2 | I = \{(0,0),(0,1)\}$, e um deslocamento $d \in Z^2 | d = (2,2)$. A imagem resultante I' do deslocamento de I por d, é composta pelos pontos $I' = \{(0+2,0+2),(0+2,1+2)\} = \{(2,2),(2,3)\}$. A notação $I + d$ será utilizada para indicar o deslocamento de I por d.

A erosão consiste em, fazendo uso de uma linguagem informal, reduzir ou contrair a área de interesse. Essa operação é definida por duas imagens preto-e-branco, $A, B \in Z^2$. A imagem resultante R se dá pelo conjunto de pontos $D \in Z^2$, tal que o conjunto B, deslocado por $d \in D$, está contido em A [4]. A imagem B geralmente é menor que A, e os deslocamentos podem ser vistos como um estêncil sobre a imagem. Caso a interseção de $B + d$ com A inclua todos os elementos de $B + d$, o elemento d aparece na imagem resultante R. O resultado efetivo desta operação é a redução das áreas de interesse. Todas as formas em A menores que B desaparecem da imagem resultante. Todas as formas em A maiores que B são reduzidas substancialmente. Com uma imagem B de tamanho maior ou igual que A, nenhum elemento apareceria na imagem resultante.

A Figura 2.5 apresenta um exemplo de erosão, com as áreas de interesse sendo as partes escuras da imagem. Esta apresenta as duas imagens A e B, utilizadas como entrada e a imagem R obtida como saída. Na imagem resultante R a área de interesse da imagem A é dividida em duas partes, e o detalhe no canto superior direito desaparece.

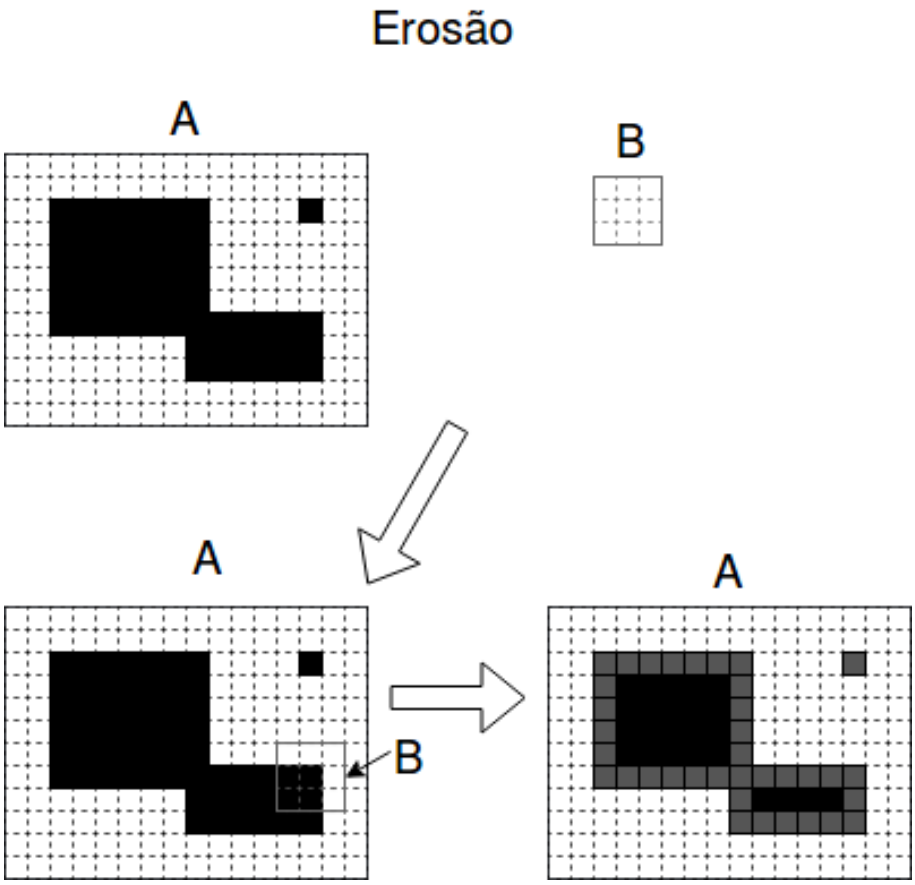


Figura 2.5: Exemplo de erosão

A dilatação pode ser vista como a operação dual da erosão. Fazendo uso da linguagem informal, a área de interesse é expandida ou dilatada. A operação é definida por duas imagens

preto-e-branco $A, B \in Z^2$. A imagem resultante R se dá pelo conjunto de pontos $D \in Z^2$, tal que a reflexão do conjunto B, deslocado por $d \in D$, apresenta pelo menos um elemento em comum com A [4]. Assim como a erosão, a imagem B é menor que A, e a mesma analogia com a movimentação do estêncil pode ser utilizada. Dado que a interseção entre A e $B + d$ apresente pelo menos um elemento em comum, o elemento d aparece na imagem resultante R. O resultado obtido desta operação é o aumento das áreas de interesse, onde um elemento isolado de A teria o mesmo tamanho de B na imagem resultante. As outras formas seriam aumentadas substancialmente.

A Figura 2.6 apresenta um exemplo de dilatação, com as áreas de interesse sendo as partes escuras da imagem. Esta apresenta as imagens de entrada A e B e a imagem resultante R. Na imagem R é possível observar que a área de interesse aparece com uma área maior. O detalhe no canto superior direito agora tem uma área maior e está mais destacado.

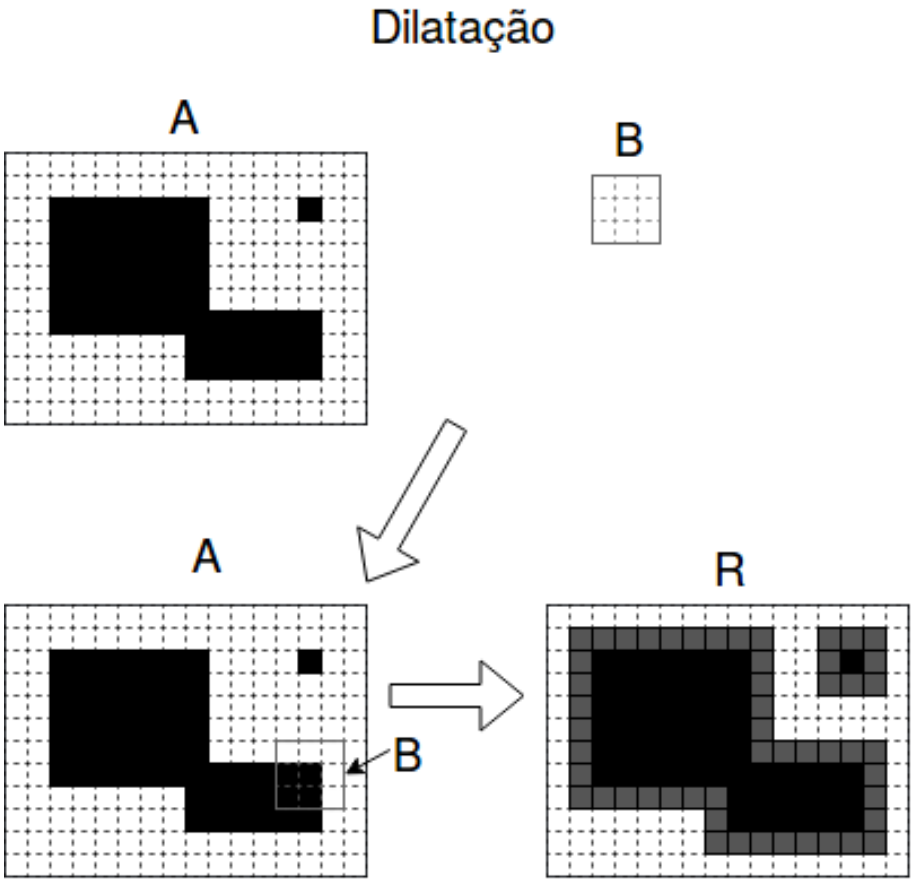


Figura 2.6: Exemplo de dilatação

Para a utilização no trabalho, estas operações foram adaptadas para o espaço Z^3. Para isso, o elemento d recebe, além da coordenada, o valor dos elementos da interseção entre A e B. Este valor é o *(a)* mais alto, no caso da dilatação ou *(b)* mais baixo, no caso da erosão, dentre os vizinhos cobertos por $B + d$. Essa alteração permite abandonar a necessidade da interseção incluir *(a)* todos os elementos, no caso da erosão ou *(b)* apenas um deles, no caso da dilatação.

2.5.2 Filtragem linear

Outra operação comumente utilizada na área de processamento de imagens é a filtragem linear [5]. Esta opera nos chamados vizinhos do *pixel* desejado e na correspondência destes com

uma matriz com tamanho igual ao da vizinhança. Os vizinhos do *pixel* consistem nos elementos que se localizam próximos àquele sendo analisado. Esta matriz tem, como terminologia predominante, os nomes de filtro, máscara, ou *kernel*, e seus elementos são chamados de componentes [5]. Para este trabalho, o termo filtro será utilizado.

O processamento acontece movendo o filtro ponto-a-ponto sobre a imagem. Para cada ponto (x, y) a **resposta** é calculada seguindo uma relação pré definida. Essa relação é dada pela soma dos produtos dos componentes pelos *pixels* correspondentes à área coberta pelo filtro. Considere um filtro de tamanho 3×3, com $k(i, j)$ o elemento (i, j) do filtro e $f(x, y)$ o valor do *pixel* (x, y) da imagem. Sendo $(0, 0)$ o canto superior esquerdo, $(0, i)$ a i-ésima coluna da primeira linha e $(i, 0)$ a i-ésima linha da primeira coluna. O valor do *pixel* (x, y) da resposta $(R(x, y))$ é dado por:

$$
\begin{aligned}
S &= k(0,0) * f(x-1, y-1) \quad &+k(0,1) * f(x-1, y) \quad &+k(0,2) * f(x-1, y+1) \\
S' &= k(1,0) * f(x, y-1) \quad &+k(1,1) * f(x, y) \quad &+k(1,2) * f(x, y+1) \\
S'' &= k(2,0) * f(x+1, y-1) \quad &+k(2,1) * f(x+1, y) \quad &+k(2,2) * f(x+1, y+1)
\end{aligned}
$$

$$R(x, y) = \lfloor S + S' + S'' \rfloor$$

Para um filtro de tamanho $m \times n$, assumimos que $m = 2a + 1$ e $n = 2b + 1$ com $a, b \in \mathbb{Z}^*$. Isso indica que os filtros possuem índices ímpares, com o menor filtro sendo $3x3$ [5] .

Um exemplo desta relação pode ser vista na Figura 2.7. A Figura apresenta um filtro F, com o divisor separado, ou seja, os elementos são $\frac{4}{16}$, $\frac{2}{16}$ e $\frac{0}{16}$. Isso foi feito para manter a soma do filtro igual a 1, de forma que o valor resultante não torna o *pixel* mais claro nem mais escuro que sua vizinhança. Esta apresenta também a imagem I e os valores de uma região desta imagem. A operação está exemplificada para aquela região, apresentando como o filtro é utilizado para o cálculo. Após o cálculo, o resultado é escrito na imagem resultante R, na posição (x, y) igual à do *pixel* central utilizado para o cálculo.

É possível ver esta operação também como um estêncil sendo deslocado sobre a imagem. Uma limitação desse formato é a operação nas bordas. As alternativas mais comuns são *(a)* ignorar as bordas em que o filtro não se encaixaria perfeitamente, retornando uma imagem menor; ou *(b)* criar uma moldura adicional de forma que todos os *pixels* da imagem sejam analisados, mantendo o tamanho da imagem; ou *(c)* realiza a operação com apenas uma parte do filtro nas bordas, ignorando os componentes que ficam para fora da imagem. A opção *(c)* foi escolhida, com a variação de que o peso dos componentes que fiquem para fora da imagem seja redistribuído entre os que foram utilizados. Esta variação busca reduzir o erro nas bordas.

Esta operação tem diversas aplicações na área de processamento de imagens, como detecção de limites, ofuscamento, redução de ruídos e aguçamento. Este é um processo computacionalmente custoso pois executa $m \times n$ multiplicações para cada *pixel*. Isso resulta num total de $m \times n \times w \times h$ multiplicações sendo w a largura e h a altura da imagem. Por esta razão a filtragem linear foi escolhida para a análise de desempenho.

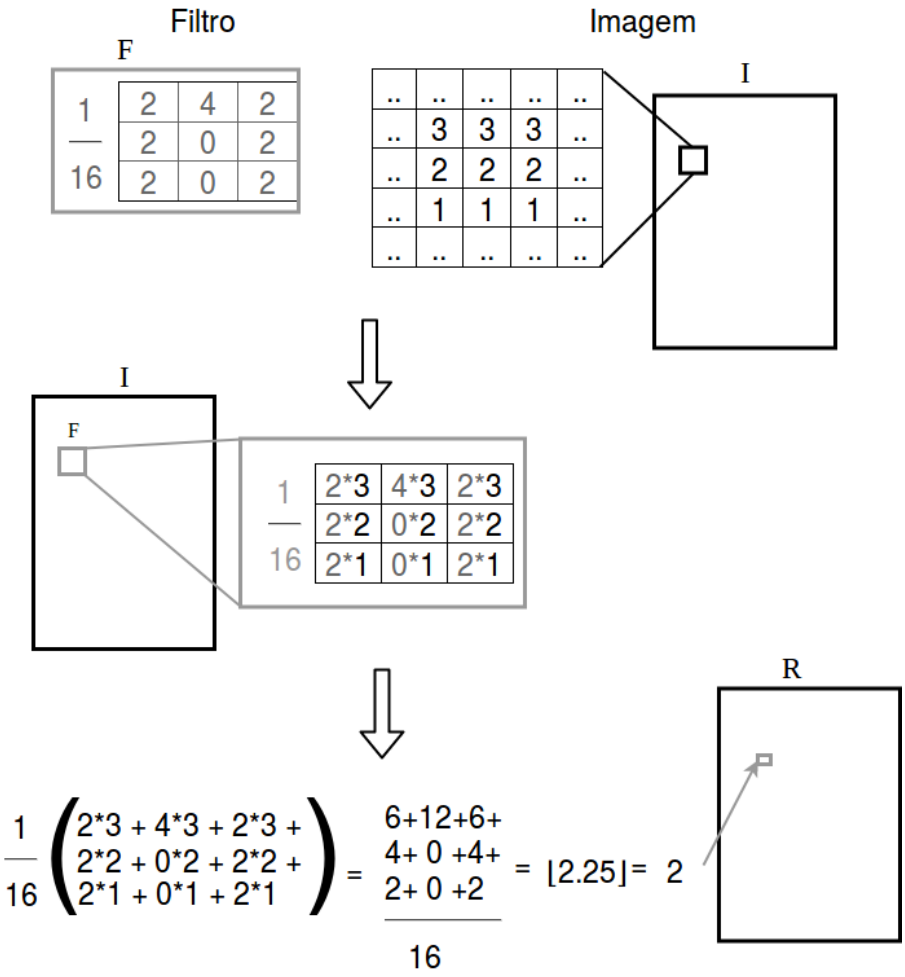


Figura 2.7: Exemplo de filtragem linear

3 Revisão bibliográfica

Há um bom número de publicações que procuram avaliar o desempenho de WASM. Essas publicações geralmente comparam o WASM com o Javascript ou asm.js e serão apresentados e analisados na Seção 3.1.

A proposta deste trabalho é fazer uma análise com uma função utilizada por editores de imagem. As publicações encontradas optaram por funções diferentes da apresentada neste trabalho. Por isso, em busca das soluções utilizadas hoje em dia para esse problema, a Seção 3.2 descreve os editores de imagem encontrados na *Web*, assim como as tecnologias utilizadas para sua implementação.

3.1 Trabalhos relacionados

As linguagens utilizadas como fonte para os códigos em WebAssembly serão apresentadas para cada trabalho. A escolha da linguagem não altera os resultados obtidos, pois a análise é feita nos códigos já compilados. Um dos resultados encontrados comparou a velocidade de execução entre o WebAssembly e o Javascript, assim como o número de ciclos utilizados por interação [2]. O trabalho utilizou C++ como linguagem fonte e utilizou a multiplicação de matrizes com três dimensões como aplicação a ser analisada. Os testes foram realizados com três implementações diferentes: *(a)* percorrendo a matriz da forma mais simples, um laço para cada dimensão desta, como é comum de ser implementado; *(b)* com cada laço percorrendo uma dimensão da matriz diferente da forma em que foi salva, visando gerar acessos à memória que provavelmente não estariam na cache; e *(c)* percorrendo a matriz com blocos, visando minimizar os erros de acesso em cache.

Os resultados obtidos foram esperados: a execução do código chegava a ser dez vezes mais rápida no WASM que no JS. Uma adversidade encontrada foi que, para matrizes abaixo de 512 elementos, o JS se mostrou mais rápido que em WASM.

A razão dessa diferença pode estar relacionada ao segundo trabalho analisado [19]. Utilizando Rust como linguagem fonte, os testes são compostos de diversas operações de manipulação de vetores e cadeias de caracteres. Chegando em uma conclusão parecida com o trabalho acima, a explicação encontrada foi que a passagem de elementos entre JS e WASM é lenta [19].

Como a memória utilizada pelos módulos WASM é implementada como um *ArrayBuffer* do JS, a escrita dos vetores originados no JS demora para ser convertido. O tempo necessário para essa conversão chega a ultrapassar o tempo de processamento diretamente em JS em casos específicos. Como exemplo apresentado, a escrita de uma cadeia de caracteres para a memória do WASM consiste em três passos: *(a)* codificar a cadeia de caracteres (em UTF-8 por exemplo); *(b)* obter um espaço de memória simulada para o WASM (semelhante ao *malloc* em C/C++); e *(c)* finalmente escrever a cadeia de caracteres na memória simulada.

Apesar deste detalhe, empresas como por exemplo a Unity[1], uma aplicação que facilita a criação de novos jogos, já começaram a utilizar o WebAssembly. O engenheiro de *software* sênior da empresa apresenta uma comparação entre o asm.js e o WASM, e o tempo que demora para carregar a primeira cena do "jogo" utilizado como aplicação para análise [18]. Em geral o WASM foi mais rápido que asm.js, a única exceção sendo o navegador Safari[2], da Apple, que acabou tendo a implementação em asm.js cerca de 20% mais rápida. A maior diferença obtida foi no navegador Google Chrome[3], da Google, onde a solução em WASM chega a ser carregada na memória quase seis vezes mais rápido que as outras tecnologias.

Como todos os trabalhos descritos acima mostraram [18][19][2], o WASM já é mais rápido que as antigas soluções, o JS e o asm.js, na grande maioria dos navegadores. A comparação que este trabalho apresenta é entre o WASM, o código compilado rodando nativamente e o asm.js.

3.2 Estado da arte

Em busca de outras aplicações do WASM e tendo como aplicação deste trabalho uma função de um editor de imagem, foi realizado um levantamento de aplicações desta área para a internet. Com o objetivo de descobrir que tecnologias e ferramentas eles utilizam, foi realizada uma análise com os diversos editores de imagens encontrados.

Uma das aplicações analisadas foi o *Online Image Editor*[4] ou editor online de imagens, em tradução livre. Esse *website* permite criar animações a partir das imagens que foram enviadas para o servidor. Também permite adicionar textos, filtros como bordas arrendondadas e molduras e efeitos como glitter. Também possibilita a aplicação de animações pré selecionadas pelo *software* e a transformação da imagem em gif animado.

A adversidade encontrada com esse editor foi a tecnologia utilizada. Todas as operações com a imagem são enviadas para um servidor, que realiza os ajustes na imagem e retorna a imagem editada. Esse processo pode apresentar falhas como perda de envio e a não certeza de que sua imagem não será utilizada para outros fins.

Outra aplicação foi a Befunky[5]. Esse editor possui, além das funções básicas como corte e redimensionamento, suporte a diversas camadas, borramento da imagem e remoção de olhos vermelhos. O programa roda totalmente no lado do cliente, fazendo uso de JS e HTML5 para editar a imagem.

O principal problema encontrado nessa aplicação foi a limitação de certas funcionalidades apenas com a versão paga. Opções como suporte a imagens de maior resolução e a remoção de anúncios do *website* só é obtida com a versão *premium* do *software*.

O Pixlr[6] foi o próximo editor analisado. O programa disponibiliza para o usuário diversos filtros a serem colocados na imagem, além da criação a partir de uma imagem em branco. Possui ferramentas como pincel, lápis, selecionar partes da imagem, criação de camadas, entre outros. As imagens são editadas puramente no lado do cliente, utilizando o *Adobe Flash Player*[7] como base. Uma nova versão está em desenvolvimento, utilizando JS e HTML5, acessível somente via assinatura do serviço.

[1] https://unity3d.com/
[2] https://www.apple.com/br/safari/
[3] https://www.google.com/chrome/
[4] http://www.online-image-editor.com/
[5] https://www.befunky.com/create/
[6] https://pixlr.com/editor/
[7] https://get.adobe.com/br/flashplayer/

A maior complicação desse *software*, apesar da atual migração para JS, é o uso do *Adobe Flash Player*, uma tecnologia da *Adobe*[8], que tem uma expectativa de fim de vida para 2020 [1].

Continuando a lista de editores, o Fotor[9] foi analisado. No passado, esse editor já utilizou *Adobe Flash Player*[4], mas sua versão atual utiliza HTML5 e JS. O Fotor apresenta diferentes ferramentas de edição, criação de textos, e alteração da imagem. Oferece rotação, corte e edição de fatores como brilho e contraste e possui uma ampla biblioteca de molduras e filtros.

O único ponto ruim a ressaltar é a limitação de certas modificações somente aos usuários que assinam o serviço. A utilização do JS, para esse trabalho, também é visto como uma possibilidade de melhoria.

O último editor a ser analisado foi o piZap [10]. Permite menos modificações que as outras aplicações analisadas, mas inclui funções adicionais como um "criador de memes", diferentes desenhos para serem adicionados à imagem, e uma ferramenta que detecta uma pessoa em uma foto para uso no editor.

A falha desse editor está no uso do *Adobe Flash Player*. Como comentado acima, essa tecnologia tem um fim de vida previsto para 2020.

Não foi encontrado nenhum editor de imagens online feito em WebAssembly. Essa pesquisa serviu também como análise de mercado e viabilidade de um editor que se utilize dessa tecnologia para implementar suas funcionalidades.

[8] `https://www.adobe.com/`
[9] `https://www.fotor.com/`
[10] `https://www.pizap.com/`

4 Metodologia

Conforme relatado no Capítulo 3, existem trabalhos comparando o desempenho de WASM com JS. Não foram encontrados trabalhos que comparassem o WASM com o desempenho de uma linguagem nativa. Este trabalho realiza essa comparação e faz uma análise dos resultados. Neste Capítulo são explicados a metodologia de testes utilizada e os objetivos que este trabalho busca alcançar.

O mesmo código C foi utilizado em todos os testes, sendo compilado para o sistema nativo, WebAssembly e asm.js. Os compiladores utilizam a LLVM como linguagem intermediária, e ambos possuem a *flag* '-O3' que habilita o mesmo nível de otimizações em ambos os compiladores.

A Seção 4.1 explica como os testes foram realizados, quais métricas foram analisadas e em quais plataformas eles aconteceram. Também detalha de que forma o teste foi especificado e os dados de entrada para o programa. A Seção 4.2 traz o objetivo principal do trabalho e os tópicos analisados separadamente para alcançar este objetivo.

4.1 Proposta de implementação

Utilizando uma aplicação real, foi implementado um editor de imagens que implementa as operações morfológicas e a filtragem linear. A filtragem linear tem um processamento demorado e, portanto, útil para análise de desempenho. Também pode ser possivelmente utilizada para aplicações mais robustas, como um editor de imagens completo com diversas funcionalidades. Os detalhes de como foi feita a implementação estão no Capítulo 5, aqui é apresentado um resumo dessa implementação.

O programa pode ser dividido em três áreas mais críticas: *(a)* o carregamento da imagem no programa, representada como um vetor; *(b)* o processamento da imagem, ou seja, a aplicação da operação escolhida, que nesse caso foi o filtragem linear; e *(c)* a escrita da imagem resultante no disco. Como um padrão escolhido arbitrariamente para os testes, o filtro escolhido foi este:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{2}{18} & \frac{2}{18} & \frac{2}{18} & 0 \\ 0 & \frac{2}{18} & \frac{2}{18} & \frac{2}{18} & 0 \\ 0 & \frac{2}{18} & \frac{2}{18} & \frac{2}{18} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

O formato de imagens escolhido para esse fim específico foi o PGM, descrito no Capítulo 2. O conjunto de imagens consiste de três imagens quadradas, com um gradiente do preto para o branco, nos tamanhos $2500 \times 2500, 5000 \times 5000, 7500 \times 7500$. Esses valores se referem à quantidade de *pixels* na imagem. Essas imagens foram escolhidas pois contém todo o espectro de cor possível para uma PGM que é representada com elementos de um *byte*, entre 0 e 255. Os valores variam de forma a tornar o *overhead* de cópia de dados algo ignorável e a tendência do

código para processamentos demorados ser identificada. É possível detectar se ocorre divergência do desempenho nativo com problemas de tamanhos maiores.

Os dados recolhidos desses testes são os tempos que cada área crítica do programa obteve, o tempo de execução total do programa (com todas as três áreas mais o código que une todas elas) e o uso de memória de cada execução. Os ambientes de teste utilizados foram os navegadores "Mozilla Firefox 61.0.1"[1] e "Chromium 68.0.3440.106"[2], rodando em um sistema "Linux Mint 18.3 Sylvia"[3]. Os testes foram realizados em cinco categorias: 1) código nativo rodando no Linux Mint; 2) código em WebAssembly rodando no Firefox; 3) código em WebAssembly rodando no Chromium; 4) código em asm.js rodando no Firefox; e 5) código em asm.js rodando no Chromium.

As categorias possuem o mesmo código fonte, escrito em C, sendo que o código nativo foi gerado utilizando o "gcc (Ubuntu 5.5.0-12ubuntu1 16.04) 5.5.0 20171010"[4], e os códigos em WASM e asm.js utilizando o "emcc (Emscripten gcc/clang-like replacement) 1.38.1"[5]. Os dois compiladores utilizam a LLVM como linguagem intermediária, e foram executados com a *flag* '-O3', que habilita o mesmo nível de otimizações para ambos os compiladores.

Cada teste foi executado 500 vezes, e as medidas obtidas foram a média e o desvio padrão. Estes foram executados diversas vezes para retirar erros de medição como acesso à cache, interrupções do sistema operacional e processos com uma prioridade maior interferindo na medição. Vale ressaltar que a medição foi realizada utilizando a biblioteca 'time.h', explicada no Seção 5.3, que utiliza os ciclos do processador e sua frequência para obtenção dos valores. Os testes executaram enquanto o computador encontrava-se sem uso, na forma de testes automatizados que podem ser encontrados no Apêndice A. Os valores podem ser encontrados no Capítulo 6.

4.2 Objetivos

Este trabalho busca comparar o desempenho de um mesmo código, compilado para diferentes plataformas, e realizar uma análise dos resultados obtidos e dos problemas encontrados.

4.2.1 Tópicos analisados

Para alcançar esse objetivo, foi realizada uma junção de diversas tarefas menores:

1. Obter os tempos de execução médios de cada plataforma;
2. Fazer uma análise da relação entre os resultados obtidos;
3. Verificar a viabilidade de se fazer um editor de imagens em WebAssembly;
4. Comparar os 2 navegadores comentados acima, visando qual deles possui um melhor desempenho em asm.js e WebAssembly;
5. Apresentar um gráfico comparativo entre as diferentes imagens e plataformas;

[1] https://www.mozilla.org/pt-BR/firefox/
[2] https://www.chromium.org/Home
[3] https://linuxmint.com/
[4] https://www.gnu.org/software/gcc/
[5] http://kripken.github.io/emscripten-site/

5 Implementação

Este Capítulo descreve como foi feita a implementação do trabalho e como as medidas foram obtidas. Todo o código foi pensado para facilitar a portabilidade para a internet, evitando bibliotecas externas e arquivos grandes. O formato de imagem utilizado, PGM, permite por exemplo a escrita de uma função de leitura curta e sem casos específicos para serem analisados.

A Seção 5.1 detalha como a passagem da imagem para o programa foi feita nas diferentes plataformas. A Seção 5.2 explica como o código foi feito e os detalhes da parte mais computacionalmente custosa deste código. A Seção 5.3 demonstra como as medidas foram realizadas e explica de maneira sucinta o uso da biblioteca 'time.h' para as medições.

5.1 Passagem da imagem

A primeira dificuldade encontrada foi como seria feita a passagem da imagem para o programa em WebAssembly. Devido à imagem ser representada com *bits* que, possivelmente, podem não representar caracteres, receber a imagem pela entrada padrão do programa poderia ser problemático. Juntamente com o compilador e o código intermediário criados pelo Emscripten, ele também oferece uma API de um sistema de arquivos[1].

Essa API permite que chamadas realizadas ao sistema de arquivos via C continuem funcionando, pois ela cria um sistema de arquivos virtual no próprio Javascript. A passagem da imagem para o programa, é feita via uma cadeia de caracteres contendo o caminho da imagem no sistema de arquivos, seja ele virtualizado no navegador ou rodando nativamente. Essa cadeia de caracteres é passada na execução do código compilado, como argumento.

5.2 Processamento

A função de processamento foi escrita de forma própria, sem embasamento na literatura. Existem trabalhos onde essa função é explicada e detalhada, porém este trabalho implementa a função de forma própria e, discutido abaixo, de maneira não otimizada. A razão desta implementação surge da ideia de utilizar as bordas para o cálculo de forma a reduzir o impacto na imagem final. Ao aplicar o filtro nas bordas, o valor resultante do cálculo nos *pixels* dentro do filtro é copiado para as bordas, onde o filtro não possuía valor, para então o cálculo ser realizado.

O código consiste de um laço que percorre toda a imagem, armazenada como um vetor, verificando se o ponto analisado possui pontos próximos que são cobertos pelo filtro recebido. Cada verificação, totalizando 24 devido ao filtro utilizado, realiza uma operação de multiplicação e adição, como pode ser visto no Código 5.1.

[1] `https://kripken.github.io/emscripten-site/docs/api_reference/Filesystem-API.html`

As linhas 1 a 7 definem quais são as variáveis e quais valores essas possuem. As linhas 7 a 13 possuem o cálculo do valor de um elemento da imagem resultante, esse cálculo é repetido para cada elemento do filtro, com a diferença que a condição do **if** tem relação com a borda que o elemento do filtro pode ultrapassar. As linhas 14 em diante apresentam a finalização do cálculo, ou seja, o valor final que o *pixel* receberá. A variável **porcentagemForaFiltro** recebe a porcentagem do filtro que não possui valores pois está fora da borda da imagem. O **valorFinal** é então calculado levando em conta essa porcentagem, e sendo limitado por 0 inferiormente ou pelo **Valor Máximo** superiormente. Este valor é então escrito na imagem de retorno **ret->data**.

Ao final de cada laço, a porcentagem do filtro que não possui *pixels* correspondentes é calculada, e o valor parcial do *pixel* é somado com a porcentagem vezes esse valor parcial. Esse método visa reduzir as variações de se ignorar a borda ou utilizar uma borda adicional preta à imagem, porém possui um desempenho ruim, em comparação com essas, devido à quantidade de verificações realizadas em cada laço.

```
1  ...
2  // i         = indice no filtro
3  // p         = pixel atual
4  // ps        = pixel superior ao atual
5  // img->data = imagem de entrada
6  // ret->data = imagem de saida
7  ...
8  if( naoEhBordaSuperior){
9    valorParcial += filtro[i] * img->data[ps];
10 }else{
11   modificador += filtro[i];
12 }
13 ...
14 float porcentagemForaFiltro = modificador/somaFiltro;
15 int valorFinal = (int)(valorParcial +
16                  porcentagemForaFiltro*valorParcial);
17 if (valorFinal > 0){
18   if(valorFinal > img->valorMaximo){
19     valorFinal = img->valorMaximo;
20   }
21 }else{
22   valorFinal=0;
23 }
24
25 ret->data[p] = valorFinal;
26 ...
```

Código 5.1: Parte do código utilizado para a filtragem linear

5.3 Medidas

As medidas obtidas foram o tempo de execução de cada parte de aplicação. Os tempos foram obtidos utilizando a biblioteca *'time.h'*, mantida pela *Free Software Foundation*. Essa biblioteca permite fazer a contagem dos ciclos do processador por um período específico. A medição do tempo foi escolhida pois uma análise de *bytecode* em diferentes linguagens pode adicionar diversas complicações. Um exemplo seria a tradução de uma instrução em certa linguagem gerar mais de uma instrução em uma outra linguagem.

A medida é feita lendo o ciclo atual, antes da instrução desejada, e o ciclo atual, após a última instrução analisada. O ciclo final é subtraído do ciclo inicial e como resultado temos o

número de ciclos necessários para executar as operações. Esse valor é dividido pela frequência do processador, ou seja, a quantidade de ciclos que ele pode realizar por segundo, obtendo assim o tempo em segundos que a série de operações demorou para executar. O Código 5.2 é um exemplo obtido do site oficial da biblioteca[2]. A função **clock()** retorna em qual ciclo o processador está naquela instrução. O cálculo da linha 9 retorna o tempo em segundos que existe entre uma medida e outra do ciclo.

```
1 #include <time.h>
2
3 clock_t start, end;
4 double cpu_time_used;
5
6 start = clock();
7 /* Do the work (Realiza as operações). */
8 end = clock();
9 cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Código 5.2: Exemplo de obtenção do tempo de execução utilizando a biblioteca 'time.h'

[2] https://www.gnu.org/software/libc/manual/html_node/CPU-Time.html

6 Resultados

Neste Capítulo serão analisados os resultados obtidos em cada um dos testes realizados. Cada teste foi executado 500 vezes, os valores mostrados correspondem à média desses testes, com os gráficos apresentando também o desvio padrão.

Todos os testes foram gerados a partir de um mesmo código, escrito em C, e compilados utilizando os compiladores descritos na Seção 4.1. Os testes foram abreviados da seguinte maneira:

c Programa rodando nativamente no Linux Mint

fwasm Programa em WebAssembly rodando no Firefox

cwasm Programa em WebAssembly rodando no Chromium

fasmjs Programa em asm.js rodando no Firefox

casmjs Programa em asm.js rodando no Chromium

Os resultados foram divididos em duas categorias: tempo e uso de memória. A Seção 6.1 traz toda a análise dos tempos de execução em cada plataforma na ordem de carregamento da imagem, escrita da imagem, processamento e o programa completo. A Seção 6.2 possui uma análise da memória gasta pelas diferentes plataformas enquanto em execução.

6.1 Tempo

As medidas e dados coletados são todos em segundos, utilizando a biblioteca 'time.h' descrita no Capítulo 5. Os resultados obtidos eram esperados, que o programa rodando nativamente é mais rápido que rodando nas outras plataformas. Os testes em WebAssembly também rodaram mais rápido que os testes em asm.js.

6.1.1 Carregamento

Essa medida se refere ao tempo necessário para buscar a imagem em disco e carregá-la para a memória do programa. No caso do programa rodando nativamente, esse processo consiste em copiar a memória do disco rígido para a memória do programa. Como explicado no Capítulo 5, o Emscripten oferece uma biblioteca que simula o sistema de arquivos no navegador. Portanto no caso dos programas rodando em algum navegador, esse processo consiste na cópia da imagem do sistema de arquivos simulado do JavaScript para a memória simulada também em Javascript do programa.

Comparando os dados numéricos da Tabela 6.1, é possível ver que o **fwasm** obtém tempos apenas 0.08 segundos mais lento que o programa nativo. Entre o Firefox e o Chromium é

interessante notar que, para a imagem de tamanho 2500×2500, o Chromium foi levemente mais rápido Entretanto, para a imagem de 7500×7500 o Firefox chegou a ser 1.5 vezes mais rápido que o Chromium, independente da tecnologia.

Tabela 6.1: Dados numéricos do carregamento de imagem em cada plataforma.

	c	fwasm	cwasm	fasmjs	casmjs
2500x2500	0.131173	0.210556	0.20031	0.363486	0.395814
5000x5000	0.499267	0.588198	0.690452	0.963496	1.31854
7500x7500	1.10474	1.1812	1.51493	1.96055	2.86183

No gráfico, apresentado na Figura 6.1, a diferença de performance entre WASM e asm.js é clara, em nenhum teste o WASM se mostrou mais lento que o asm.js. Comparando os navegadores, é possivel ver que o asm.js é quase 2 vezes mais lento que o WASM em todos os casos de teste. Um detalhe interessante de ressaltar é a consistência do resultado obtido com o Chromium, em que o desvio padrão foi o menor em relação às outras tecnologias.

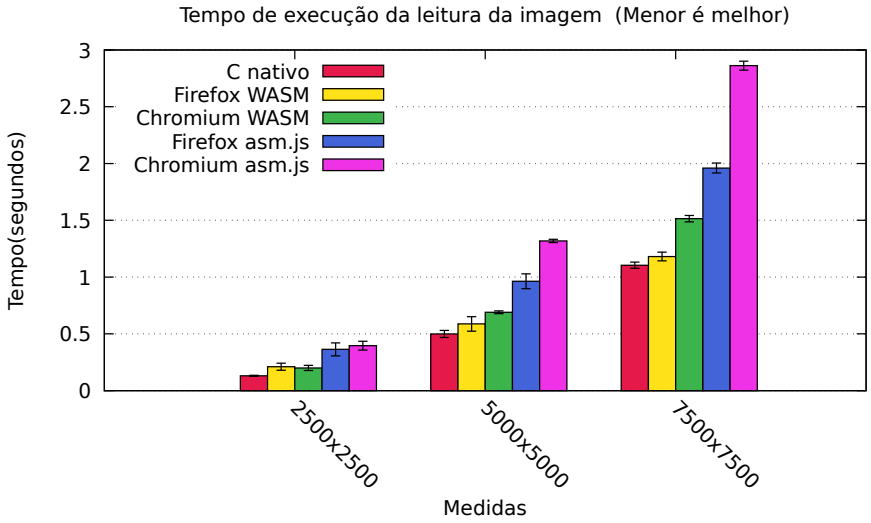


Figura 6.1: Gráfico mostrando o tempo médio de carregamento da imagem para o programa nas diferentes plataformas.

6.1.2 Escrita

Essa medida pode ser considerada a oposta da anterior, e se refere ao tempo necessário para salvar uma imagem do programa no disco. Para o programa rodando nativamente, isso significa escrever a memória armazenada para o disco. No caso dos programas rodando em algum navegador, esse tempo é o necessário para copiar a imagem da memória simulada para o sistema de arquivos virtual do JavaScript.

Em comparação com os dados obtidos anteriormente, o tempo de escrita da imagem no disco é muito pior nos navegadores. Na Tabela 6.2, os navegadores demoram mais de duas vezes o tempo que o programa nativo demora, em todas as tecnologias. A diferença aumenta com o tamanho da imagem, com a imagem de 7500×7500 os navegadores rodando com WASM chegam a ser 2.7 vezes mais lentos.

Observando o gráfico na Figura 6.2, é possível notar que o **cwasm** tende a executar mais rápido que o **fwasm**, com dados maiores. Enquanto a imagem de 2500×2500 o **cwasm** tinha

Tabela 6.2: Dados numéricos da escrita de imagem em cada plataforma.

	c	fwasm	cwasm	fasmjs	casmjs
2500x2500	0.131136	0.25532	0.327292	0.330902	0.504222
5000x5000	0.509681	1.30102	1.35341	1.54537	1.99846
7500x7500	1.12712	3.06742	3.0243	3.60301	4.43109

um tempo comparável com o **fasmjs**, na imagem 7500×7500 o **cwasm** foi mais rápido que o **fwasm**. O asm.js, nesse caso de teste, não ultrapassou os 60% mais lento em relação ao WASM, tendo um ganho considerável em relação ao teste anterior.

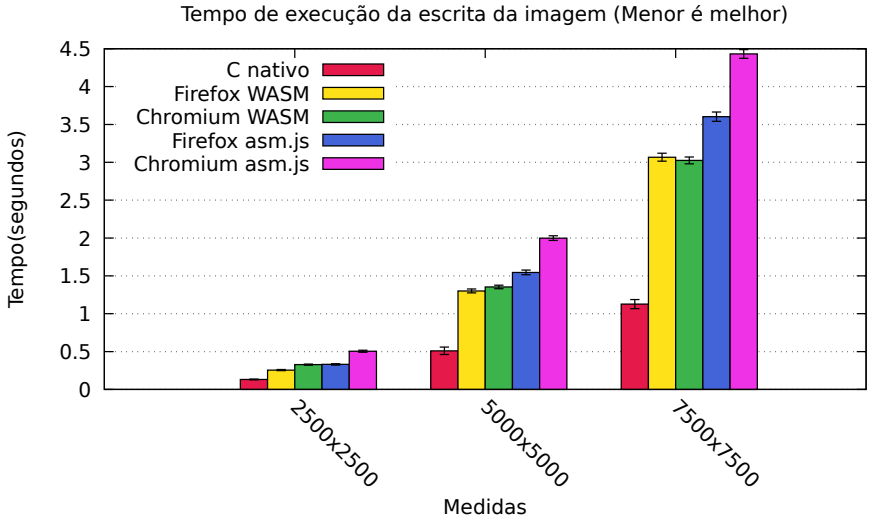


Figura 6.2: Gráfico mostrando o tempo médio de escrita da imagem para o disco nas diferentes plataformas.

6.1.3 Processamento

Essa medida se refere ao tempo que o programa demorou para processar a imagem, de acordo com o programa especificado no Capítulo 5. O mesmo código fonte em C foi utilizado para todas as plataformas.

Os dados da Tabela 6.3 mostra que o **cwasm** executou consistentemente entre 1.95 e 1.88 da velocidade do **c**, ou seja, perto de duas vezes mais lento que o processamento nativo. Apesar do tempo do **casmjs** ser, em média, 1.47 vezes mais lenta que o **fasmjs**, a diferença entre o **fasmjs** e o **c** nativo alcançou os 2.73 vezes mais lento. As discrepâncias entre plataformas estão mais acentuadas para esse caso de teste, com o **fwasm** tendo executado a 1.5 da velocidade nativa na instância mais rápida deste.

Tabela 6.3: Dados numéricos do processamento da imagem em cada plataforma.

	c	fwasm	cwasm	fasmjs	casmjs
2500x2500	0.234214	0.424932	0.456352	0.632024	0.94855
5000x5000	0.93987	1.41478	1.76686	2.35171	3.4733
7500x7500	2.10154	3.16435	3.96653	5.21675	7.57075

O gráfico representado na Figura 6.3 traz um padrão interessante. A velocidade de execução manteve um padrão com um aumento de cerca de 20% entre as tecnologias. O **c** nativo executou aproximadamente duas vezes mais lento para cada imagem teste utilizada.

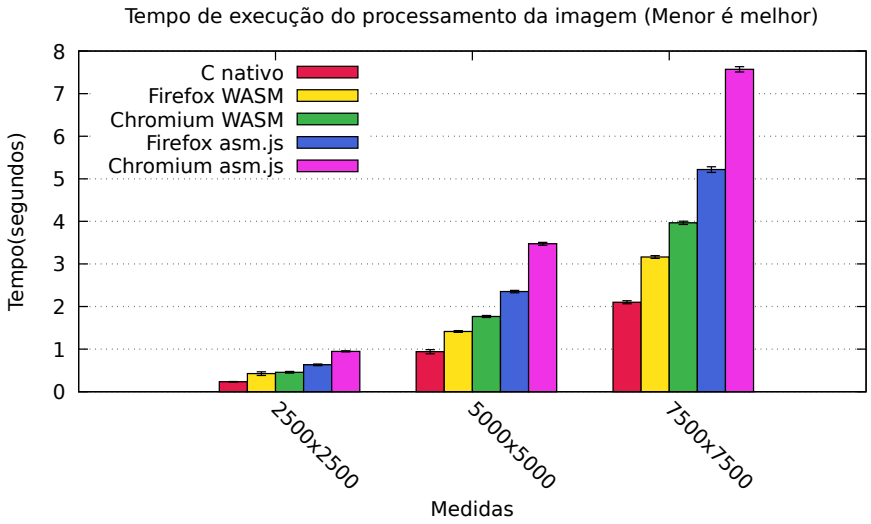


Figura 6.3: Gráfico mostrando o tempo médio de processamento da imagem nas diferentes plataformas.

6.1.4 Programa completo

Os dados analisados nesta medição podem ser considerados como a soma dos resultados obtidos anteriormente, visto que o código entre as áreas críticas do programa não deveriam ser demorados para executar.

A Tabela 6.4 mostra que o tempo de execução do programa completo no **cwasm** executou cerca de 1.15 vezes mais lento que o **fwasm**. Essa diferença entre navegadores é agravada para o caso do asm.js, com o Chromium chegando a ser 41% mais lento que o Firefox. Em relação ao **c** nativo, o WASM nunca chegou a ser 2 vezes mais lento, mas essa diferença também nunca foi menor que 1.4 vezes mais lento.

Tabela 6.4: Dados numéricos da execução do programa completo para cada plataforma.

	c	fwasm	cwasm	fasmjs	casmjs
2500x2500	0.501399	0.890938	0.983994	1.32876	1.84931
5000x5000	1.9623	3.30412	3.81077	4.86313	6.791
7500x7500	4.35956	7.4131	8.50581	10.783	14.8644

No gráfico que temos na Figura 6.4, vemos um padrão muito semelhante à Figura 6.3, com um aumento de cerca de duas vezes no tempo de execução do **c** nativo para cada imagem de entrada. A variação dos tempos ficou quase nula, com o desvio padrão de todas as tecnologias próximo de zero. A maior discrepância do gráfico é entre o **cwasm** e o **c** nativo, onde o primeiro executa mais de 3 vezes mais lento.

6.2 Uso de Memória

Outro fator a ser considerado é a quantidade de memória gasta por cada programa. Aqui necessitamos de uma análise levemente diferente, visto que o programa nativo não precisa de

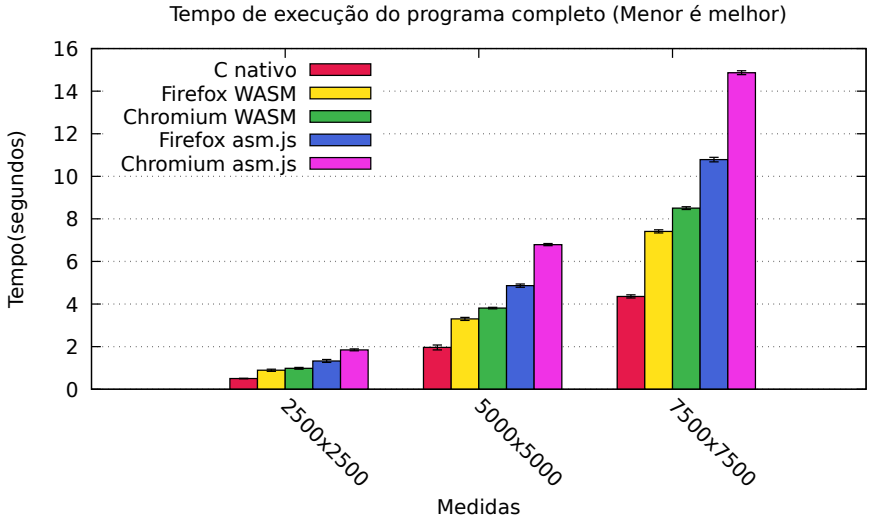


Figura 6.4: Gráfico mostrando o tempo médio da execução do programa completo nas diferentes plataformas.

nenhum suporte adicional para executar, enquanto os programas em WASM e asm.js precisam do navegador. A análise desses dados ocorrerá pela diferença de crescimento entre uma e outra tecnologia.

Na Tabela 6.5 é possível notar que o Firefox, em geral, consome menos memória que o Chromium. O programa nativo tem um aumento de 3.8 vezes entre a imagem de 2500×2500 e 5000×5000, enquanto o **fwasm** apresentou um aumento de 1.52 vezes e o **cwasm** um aumento de 2 vezes. Isso era esperado, visto que o **c** nativo é composto quase puramente da imagem, enquanto os navegadores possuem diversas ferramentas que ocupam uma memória praticamente fixa.

Tabela 6.5: Dados numéricos do uso de memória em cada plataforma.

	c	fwasm	cwasm	fasmjs	casmjs
2500x2500	51871.6	682986	754662	729118	809870
5000x5000	198487	1.03917e+06	1.54282e+06	1.15051e+06	1.62774e+06
7500x7500	442579	1.70006e+06	2.63643e+06	1.95374e+06	2.47432e+06

É interessante observar, no gráfico da Figura 6.5, que o **cwasm** acaba consumindo mais memória que o **casmjs**, mesmo sendo o mesmo navegador. O desvio padrão no **casmjs** é grande o suficiente para incluir o uso de memória médio do **cwasm**. Uma teoria que pode explicar essa diferença é que a implementação do interpretador de WebAssembly no Chromium ainda esteja no começo do desenvolvimento. De forma que a escalabilidade do interpretador não foi testada para processamentos mais demorados. O desvio padrão também aumentou no uso de memória, sendo o exemplo mais óbvio o **casmjs** para uma imagem de 7500×7500.

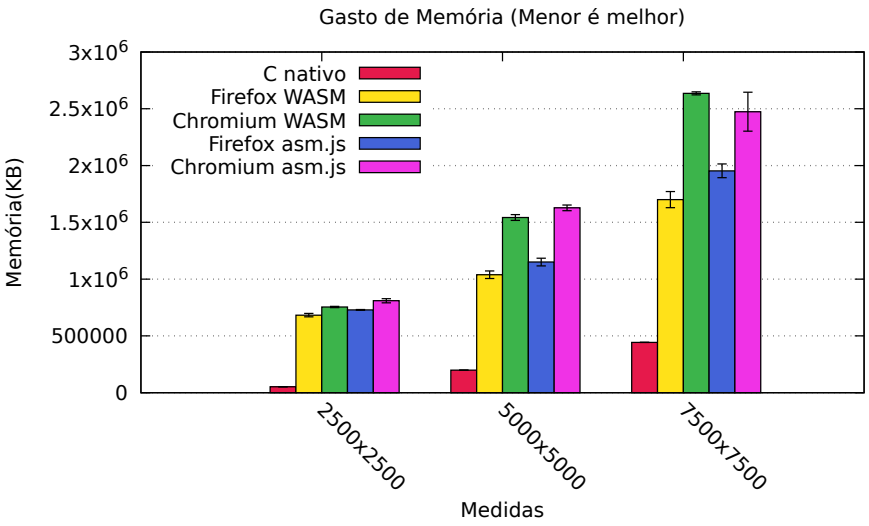


Figura 6.5: Gráfico mostrando o gasto médio de memória nas diferentes plataformas.

7 Conclusão

Este Capítulo busca trazer um resumo dos resultados obtidos e dos futuros avanços possíveis nessa área, assim como trabalhos que podem se beneficiar deste. Como visão geral, temos que o resultado obtido foi o esperado, o qual a execução nativa ainda é mais rápida que o WASM, porém o WASM apresenta um ganho alto em relação ao asm.js

A Seção 7.1 faz um resumo de todos os resultados obtidos e detalhes anormais. A Seção 7.2 apresenta algumas ideias de como melhorar ou utilizar os resultados desse trabalho.

7.1 Análise dos Resultados

Nos gráficos apresentados no Capítulo 6, conseguimos perceber um mesmo ganho de performance entre o nativo e o WASM, como vemos entre o WASM e o asm.js. Ainda existem problemas que precisam ser analisados e contornado no WebAssembly, como a passagem de elementos do JavaScript para este e a independência do WebAssembly em relação ao JS.

É possível concluir também que editores de imagem que façam uso dessa tecnologia poderão executar mais rápido e com um eventual gasto de memória reduzido.

Um detalhe interessante é que o carregamento de arquivos no programa, utilizando o sistema de arquivos virtual no navegador, consegue obter uma performance muito próxima da nativa. Isso viabiliza aplicações onde o volume de dados de entrada é grande.

Um ponto negativo dos resultados é que, caso seja necessário muita interação entre usuário e o programa, a interface do JavaScript pode ser mais lenta que o processamento dos dados. A dependência entre WASM e JavaScript ainda é grande, e existem problemas que o JavaScript resolveria de uma forma mais eficiente.

É notável que ainda exista uma necessidade de melhora do WebAssembly, mas para uma tecnologia com apenas três anos em desenvolvimento, e apenas um ano e meio desde sua divulgação, o potencial dela ainda é grande.

7.2 Trabalhos Futuros

Futuras análises de performance do WASM podem utilizar este trabalho como base para detectar uma diferença no avanço da tecnologia. Este trabalho também pode servir de base para um editor de imagens completo, rodando totalmente em WebAssembly.

Sendo um dos primeiros trabalhos que comparou o WASM com o nativo, este trabalho apresenta alguns detalhes que não foram cobertos pelos testes. Operações com números em ponto flutuante, sistema de entrada e saída de dados utilizando stdin e stdout, motor gráfico para renderização (atualmente o WebGL), entre outras características podem ser testadas. Comparações entre códigos gerados por diferentes compiladores também é uma área a ser explorada.

É possível expandir o escopo na área de execução do código, como o desenvolvimento de uma VM que execute o WASM diretamente, sem a necessidade de um navegador completo.

O WebAssembly é uma tecnologia com muito potencial, que ainda esta no começo do desenvolvimento e do uso por grandes empresas, mas já se mostra uma tecnologia quase duas vezes melhor que as anteriores. Essa possibilidade de trazer códigos compilados e otimizados para a *Web* e com um tamanho reduzido é uma mudança grande e benéfica para a área. É esperado que essa tecnologia seja amplamente utilizada no futuro, e que as otimizações que ela proporciona permitam aplicações inovadoras.

Referências

[1] Adobe Corporate Communications. Flash & the future of interactive content. https://theblog.adobe.com/adobe-flash-update, jul 2017. Acessado em 03/08/2018.

[2] Winston Chen. Performance testing web assembly vs javascript - an experiment with some surprising results. https://medium.com/samsung-internet-dev/performance-testing-web-assembly-vs-javascript-e07506fd5875, mai 2018. Acessado em 12/11/2018.

[3] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing 2nd Edition*, pages 1–3. Prentice Hall, 2002.

[4] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing 2nd Edition*, pages 519–527. Prentice Hall, 2002.

[5] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing 2nd Edition*, pages 116–123. Prentice Hall, 2002.

[6] Andreas Haas, Andreas Rossberg, Dereck L. Schuff, Ben L. Titzer, Dan Gohman, Luke Wagner, Alon Zakai, Michael Holman, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, jun 2017. Disponível em: https://github.com/WebAssembly/spec/raw/master/papers/pldi2017.pdf. Acessado em 10/02/2018.

[7] Juergen Haas. The difference between compiled and interpreted languages. https://www.lifewire.com/compiled-language-2184210, nov 2018. Acessado em 21/11/2018.

[8] D. Herman, L. Wagner, and A. Zakai. asm.js. http://asmjs.org/spec/latest/, 2014. Acessado em 12/03/2018.

[9] ITU World Telecommunication/ICT Indicators database. Key ict indicators for developed and developing countries and the world (totals and penetration rates). http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2014/ITU_Key_2005-2014_ICT_data.xls, 2014. Acessado em 21/11/2018.

[10] Netscape. Netscape and sun announce javascript, the open, cross-platform object scripting language for enterprise networks and the internet. https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html, dez 1995. Acessado em 20/06/2018.

[11] Dwayne Phillips. *Image Processing in C*. R&D Publications Inc., 1st edition, 1994.

[12] Jef Poskanzer. Netpbm. http://netpbm.sourceforge.net/doc/index.html, jan 2014. Acessado em 12/11/2018.

[13] Jef Poskanzer. Pgm format specification. http://netpbm.sourceforge.net/doc/pgm.html, out 2016. Acessado em 12/11/2018.

[14] Paul Shan. Is javascript really interpreted or compiled language? http://voidcanvas.com/is-javascript-really-interpreted-or-compiled-language/, abr 2017. Acessado em 28/11/2018.

[15] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*, pages 2–3. Prentice Hall Press, 5th edition, 2010.

[16] The Linux Information Project. File formats: A brief introduction. http://www.linfo.org/file_format.html, nov 2006. Acessado em 12/11/2018.

[17] Tsvetomira Trifonova. Most used image formats on the web. https://blog.superhosting.bg/en/image-formats-in-web.html, jul 2016. Acessado em 12/11/2018.

[18] Marco Trivellato. Webassembly load times and performance. https://blogs.unity3d.com/pt/2018/09/17/webassembly-load-times-and-performance/, set 2018. Acessado em 12/11/2018.

[19] Vladimir. Webassembly vs. the world. should you use webassembly? https://blog.sqreen.io/webassembly-performance/, ago 2018. Acessado em 12/11/2018.

[20] Luke Wagner. Webassembly. https://blog.mozilla.org/luke/2015/06/17/webassembly/, jun 2015. Acessado em 21/11/2018.

Apêndice A: Script de testes

```
1
2  #!/bin/bash
3
4  nTests=500
5  cp image2500.pgm image.pgm
6  for (( i=0 ; i<nTests ; ++i ))
7  do
8    > results/c5000_$i.txt
9    > results/fwasm5000_$i.txt
10   > results/fasmjs5000_$i.txt
11   > results/cwasm5000_$i.txt
12   > results/casmjs5000_$i.txt
13   ./memusg timeout 15 imageOperators/tgTest image.pgm out.pgm 3 0 0 0 0 0 \
14   0 2 2 2 0 0 2 2 2 0 0 2 2 2 0 0 0 0 0 0 18 &>>results/c5000_$i.txt
15   rm result.txt
16   ln -s results/fwasm5000_$i.txt result.txt
17   pkill node
18   node server.js >> result.txt &
19   ./memusg timeout 15 firefox http://www.inf.ufpr.br/enba14/TG/wasm/ \
20   2>>results/fwasm5000_$i.txt
21   rm result.txt
22   ln -s results/fasmjs5000_$i.txt result.txt
23   pkill node
24   node server.js >> result.txt &
25   ./memusg timeout 15 firefox http://www.inf.ufpr.br/enba14/TG/asm_js/ \
26   2>>results/fasmjs5000_$i.txt
27   rm result.txt
28   ln -s results/cwasm5000_$i.txt result.txt
29   pkill node
30   node server.js >> result.txt &
31   ./memusg timeout 15 chromium-browser http://www.inf.ufpr.br/enba14/TG/wasm/ \
32   2>>results/cwasm5000_$i.txt
33   rm result.txt
34   ln -s results/casmjs5000_$i.txt result.txt
35   pkill node
36   node server.js >> result.txt &
37   ./memusg timeout 15 chromium-browser http://www.inf.ufpr.br/enba14/TG/asm_js/ \
38   2>>results/casmjs5000_$i.txt
39 done
40 cp image5000.pgm image.pgm
41 for (( i=0 ; i<nTests ; ++i ))
42 do
43   > results/c5000_$i.txt
44   > results/fwasm5000_$i.txt
45   > results/fasmjs5000_$i.txt
46   > results/cwasm5000_$i.txt
47   > results/casmjs5000_$i.txt
48   ./memusg timeout 15 imageOperators/tgTest image.pgm out.pgm 3 0 0 0 0 0\
49   0 2 2 2 0 0 2 2 2 0 0 2 2 2 0 0 0 0 0 0 18 &>>results/c5000_$i.txt
50   rm result.txt
51   ln -s results/fwasm5000_$i.txt result.txt
```

```
52   pkill node
53   node server.js >> result.txt &
54   ./memusg timeout 15 firefox http://www.inf.ufpr.br/enba14/TG/wasm/ \
55   2>>results/fwasm5000_$i.txt
56   rm result.txt
57   ln -s results/fasmjs5000_$i.txt result.txt
58   pkill node
59   node server.js >> result.txt &
60   ./memusg timeout 15 firefox http://www.inf.ufpr.br/enba14/TG/asm_js/ \
61   2>>results/fasmjs5000_$i.txt
62   rm result.txt
63   ln -s results/cwasm5000_$i.txt result.txt
64   pkill node
65   node server.js >> result.txt &
66   ./memusg timeout 15 chromium-browser http://www.inf.ufpr.br/enba14/TG/wasm/ \
67   2>>results/cwasm5000_$i.txt
68   rm result.txt
69   ln -s results/casmjs5000_$i.txt result.txt
70   pkill node
71   node server.js >> result.txt &
72   ./memusg timeout 15 chromium-browser http://www.inf.ufpr.br/enba14/TG/asm_js/ \
73   2>>results/casmjs5000_$i.txt
74 done
75 cp image5000.pgm image.pgm
76 for (( i=0 ; i<nTests ; ++i ))
77 do
78   > results/c5000_$i.txt
79   > results/fwasm5000_$i.txt
80   > results/fasmjs5000_$i.txt
81   > results/cwasm5000_$i.txt
82   > results/casmjs5000_$i.txt
83   ./memusg timeout 15 imageOperators/tgTest image.pgm out.pgm 3 0 0 0 0 0 \
84   0 2 2 2 0 0 2 2 2 0 0 2 2 2 0 0 0 0 0 0 18 &>>results/c5000_$i.txt
85   rm result.txt
86   ln -s results/fwasm5000_$i.txt result.txt
87   pkill node
88   node server.js >> result.txt &
89   ./memusg timeout 15 firefox http://www.inf.ufpr.br/enba14/TG/wasm/ \
90   2>>results/fwasm5000_$i.txt
91   rm result.txt
92   ln -s results/fasmjs5000_$i.txt result.txt
93   pkill node
94   node server.js >> result.txt &
95   ./memusg timeout 15 firefox http://www.inf.ufpr.br/enba14/TG/asm_js/ \
96   2>>results/fasmjs5000_$i.txt
97   rm result.txt
98   ln -s results/cwasm5000_$i.txt result.txt
99   pkill node
100  node server.js >> result.txt &
101  ./memusg timeout 15 chromium-browser http://www.inf.ufpr.br/enba14/TG/wasm/ \
102  2>>results/cwasm5000_$i.txt
103  rm result.txt
104  ln -s results/casmjs5000_$i.txt result.txt
105  pkill node
106  node server.js >> result.txt &
107  ./memusg timeout 15 chromium-browser http://www.inf.ufpr.br/enba14/TG/asm_js/ \
108  2>>results/casmjs5000_$i.txt
109 done
```

```
cp image7500.pgm image.pgm
for (( i=0 ; i<nTests ; ++i ))
do
  > results/c7500_$i.txt
  > results/fwasm7500_$i.txt
  > results/fasmjs7500_$i.txt
  > results/cwasm7500_$i.txt
  > results/casmjs7500_$i.txt
  ./memusg timeout 25 imageOperators/tgTest image.pgm out.pgm 3 0 0 0 0 0 \
  0 2 2 2 0 0 2 2 2 0 0 2 2 2 0 0 0 0 0 0 18 &>>results/c7500_$i.txt
  rm result.txt
  ln -s results/fwasm7500_$i.txt result.txt
  pkill node
  node server.js >> result.txt &
  ./memusg timeout 25 firefox http://www.inf.ufpr.br/enba14/TG/wasm/ \
  2>>results/fwasm7500_$i.txt
  rm result.txt
  ln -s results/fasmjs7500_$i.txt result.txt
  pkill node
  node server.js >> result.txt &
  ./memusg timeout 25 firefox http://www.inf.ufpr.br/enba14/TG/asm_js/ \
  2>>results/fasmjs7500_$i.txt
  rm result.txt
  ln -s results/cwasm7500_$i.txt result.txt
  pkill node
  node server.js >> result.txt &
  ./memusg timeout 25 chromium-browser http://www.inf.ufpr.br/enba14/TG/wasm/ \
  2>>results/cwasm7500_$i.txt
  rm result.txt
  ln -s results/casmjs7500_$i.txt result.txt
  pkill node
  node server.js >> result.txt &
  ./memusg timeout 25 chromium-browser http://www.inf.ufpr.br/enba14/TG/asm_js/ \
  2>>results/casmjs7500_$i.txt
done
```

Código A.1: Script para a execução dos testes