Uma avaliação empírica sobre o uso de WebAssembly para descarga de funções de realidade aumentada

Matheus Pires[1], Gabriel Nery Espindola[1], Karlla Chaves Rodrigues[1], Kleber Vieira Cardoso[1], Fabio Luciano Verdi[2], Bruno Silvestre[1], Sand Luz Correa[1]

[1]Universidade Federal de Goiás (UFG), Goiânia, GO – Brasil

[2]Universidade Federal de São Carlos (UFSCar), Sorocaba, SP – Brasil

{matheuspires23, gabriel.nery, karllachaves}@discente.ufg.br,
{sandluz, kleber, brunoos}@ufg.br,
verdi@ufscar.br

Abstract. *The growing demand for augmented reality applications has culminated in the development of high-performance algorithms, but with high computational demands. Therefore, there is an urgent need to bring this type of application to popular environments, especially browsers and mobile devices. This article evaluates two versions of an AR application, the first runs the algorithms in the browser, the second allows to offload them to an edge server. Both solutions are based on WebAssembly, which guarantees compatibility between environments. Results show an optimistic perspective for using WebAssembly as a offloading mechanism. However, there are still limitations to be overcome.*

Resumo. *A crescente demanda por aplicações de realidade aumentada, culminou no desenvolvimento de algoritmos de alto desempenho, porém com alta demanda computacional. Dessa forma, é urgente a necessidade de se levar esse tipo de aplicação para os ambientes populares, especialmente os navegadores e dispositivos móveis. Este artigo avalia duas versões de uma aplicação de AR, a primeira executa os algoritmos dentro do navegador, já segunda faz a descarga deles para um servidor de borda. Ambas as soluções se baseiam em WebAssembly, o qual garante a compatibilidade entre os ambientes. Resultados mostram uma perspectiva otimista para o uso do WebAssembly como mecanismo de offloading. Contudo, ainda existem limitações a serem superadas.*

1. Introdução

Nos últimos anos, avanços em tecnologias de Realidade Estendida (*Extended Reality* - XR), incluindo Realidade Aumentada (*Augmented Reality* - AR) e Realidade Virtual (*Virtual Reality* - VR), bem como em redes de comunicação de baixa latência, como as redes 5G, permitiram a proposição de novos casos de uso onde o mundo real é combinado com conteúdo virtual para aprimorar a experiência do usuário móvel. Esses novos casos de uso têm o potencial de promover novas formas de interação humana, aprendizagem e entretenimento [Cao et al. 2023, Gapeyenko et al. 2023, Alriksson et al. 2021]. Na navegação, por exemplo, dispositivos como óculos de AR (*Head-Mounted Displays - HMDs*) e *smartphones* podem realizar o reconhecimento de objetos no ambiente para localizar o usuário e sobrepor instruções de navegação.

Um dos princípios da AR é que objetos virtuais, geralmente representados em 3D, são sobrepostos ao ambiente físico para agirem em sincronia com o mundo real, proporcionando uma percepção de profundidade adequada ao usuário [Bekele et al. 2018]. Para atingir esse objetivo, HMDs e *smartphones* devem analisar com precisão o ambiente físico em tempo real. Essa tarefa, no entanto, requer algoritmos de visão computacional que consomem muita energia e que geralmente não são adequados para serem executados em hardware com restrição de recursos. Outro desafio que impede as aplicações AR de atingirem todo o seu potencial em dispositivos móveis é a renderização de objetos virtuais 3D, pois essa tarefa deve ser realizada com um atraso inferior a 15 ms [Morín et al. 2022].

De fato, a capacidade limitada de processamento e armazenamento, o tempo de vida de bateria e a dissipação de calor nos HMDs e *smartphones*, bem como a necessidade de requisitos estritos de latência, motivam o *offloading* (descarregamento) de algumas funções de AR (computacionalmente intensivas) do dispositivo para a borda da rede, expandindo a funcionalidade da aplicação para um ambiente computacional localizado mais próximo do usuário [Siriwardhana et al. 2021]. Do ponto de vista da aplicação AR, essa expansão permite uma melhor qualidade de experiência, uma vez que gráficos aprimorados, taxas de quadros mais altas e tempos de processamento menores são possibilitados pelo uso de uma infraestrutura de computação mais poderosa que a disponível no dispositivo do usuário.

Por outro lado, o surgimento da linguagem WebAssembly [Haas et al. 2017] possibilitou a execução de código dentro de navegadores Web, porém com desempenho próximo ao do código nativo, permitindo executar algoritmos que demandam maior desempenho dentro do navegador. Neste contexto, WebAssembly se torna uma alternativa promissora para aplicações AR que executam em dispositivos móveis. Outra vantagem do WebAssembly é que, por ser uma especificação de um formato binário (e sua semântica formal), ele pode ser executado em ambientes fora do navegador, através de *runtimes* independentes ou interpretadores, tornando-se também uma solução atrativa para implementação de códigos que executam em servidores [Gadepalli et al. 2019]. A possibilidade de usar a mesma tecnologia no cliente e no servidor, aliado à sua portabilidade entre navegadores e arquiteturas de hardware, tornam o WebAssembly uma ferramenta promissora para superar os problemas de heterogeneidade ao fazer o *offloading* de tarefas para a borda da rede [Hoque and Harras 2022]. No entanto, compilar um programa sofisticado em WebAssembly não é uma tarefa trivial, uma vez que eles podem utilizar serviços do sistema operacional que ainda não são providos pelo WebAssembly, como *threads* e *sockets* [Jangda et al. 2019].

Neste contexto, apresentamos neste trabalho uma avaliação empírica sobre o uso de WebAssembly para o *offloading* de funções AR. Mais especificamente, avaliamos o desempenho de uma aplicação Web implementada em dois modelos de *offloading*. O primeiro, denominado *low-offload*, executa toda a aplicação dentro do navegador, do lado do cliente. O segundo modelo, denominado *mid-offload*, executa as funções de AR que demandam mais poder de processamento, codificadas em uma biblioteca de *Simultaneous Localization and Mapping* (SLAM) [Reitmayr et al. 2010], no lado do servidor. Resultados mostram que a utilização do WebAssembly permitiu que a biblioteca de SLAM executasse com bom desempenho no navegador. Além disso, foi possível utilizar a mesma biblioteca no servidor, facilitando assim o desenvolvimento de ambos os modelos de *of-*

floading. No entanto, em nossos experimentos, o modelo *low-offload* apresentou melhor desempenho (em termos de tempo total de processamento e taxa de quadros) que o modelo *mid-offload*, indicando alguma limitação no ambiente de execução do WebAssembly, impactando nas funcionalidades complexas que lidam com o sistema operacional. Por outro lado, o *offloading* de tarefas do cliente para o servidor, permitiu economia de energia no primeiro. Esses resultados demonstram uma perspectiva otimista para o uso do *WebAssembly* como mecanismo de *offloading*, porém ainda existem limitações a serem superadas pela nova tecnologia.

O restante deste trabalho está organizado conforme descrito a seguir. Na Seção 2 apresentamos os fundamentos relacionados com o trabalho. Na Seção 3 discutimos os trabalhos relacionados. Na Seção 4 detalhamos o ambiente de experimentação e a metodologia utilizada. Os resultados são discutidos na Seção 5. Finalmente, na Seção 6 apresentamos as considerações finais e direções para trabalhos futuros.

2. Fundamentos

Nesta seção, primeiramente discutimos alguns fundamentos sobre *offloading* de aplicações AR. Em seguida, introduzimos alguns conceitos básicos relacionados com WebAssembly.

2.1. *Offloading* em Aplicações AR

O princípio básico da AR é que os objetos virtuais são sobrepostos a ambientes físicos. Esses objetos são geralmente renderizados pelos dispositivos do usuário (e.g., HMDs e *smartphones*) e podem incluir modelos 3D, vídeos 2D e texto 2D [Cao et al. 2023]. A sobreposição dos objetos no mundo real pode ser alcançado de diferentes maneiras. Um dos métodos mais utilizados consiste em incorporar os objetos como se fizessem parte do ambiente, realizando um mapeamento do ambiente em torno do usuário através de um algoritmo de SLAM.

Para atingir esse objetivo, tipicamente, uma aplicação AR envolve o conjunto de funcionalidades ilustradas no lado esquerdo da Figura 1. A função mais básica consiste na captura de imagens do ambiente em torno do usuário, bem como dados de sensores do dispositivo, como câmeras RGBs, microfones, GPS, IMU. Essas imagens e dados capturados são então usados pelo algoritmo de SLAM, o qual realiza três funções: extração de características, localização do dispositivo e geração de nuvem de pontos. A primeira função identifica pontos de interesse nas imagens, como esquinas, bordas ou outros elementos distintivos. Usando as características extraídas, a segunda função estima a posição do dispositivo no ambiente para, em seguida, gerar uma nuvem de pontos. Esta, por sua vez, representa os pontos tridimensionais do ambiente com base nas características extraídas, sendo usada para criar e atualizar o mapa do ambiente. De fato, as funções do SLAM atualmente são umas das principais funcionalidades em AR, uma vez que, para inserir um conteúdo virtual em um ambiente real, é necessário criar um mapa espacial do ambiente, localizar-se dentro deste mapa e decidir onde os novos conteúdos serão colocados. Uma vez que o SLAM estima a localização do dispositivo, objetos virtuais são sobrepostos na imagem da câmera ou renderizados diretamente nas visualizações do mundo físico. Áudio e som também podem ser codificados no vídeo para posterior transmissão. Essas funcionalidades compõem o serviço de processamento de vídeo e transporte.

O SLAM é um algoritmo computacionalmente intensivo, especialmente para dispositivos móveis com restrição de processamento, memória e bateria [Toczé et al. 2019].

Com o surgimento da comunicação de baixa latência na tecnologia 5G e pós-5G, torna-se possível transferir partes do processamento e das funcionalidade descritas acima para um servidor de borda e melhorar a experiência do usuário. No trabalho [Alriksson et al. 2021], são propostos três modelos diferentes com esse propósito, como ilustrado no lado direito da Figura 1. O primeiro modelo, denominado *low-offload*, executa todas ou praticamente todas as funções de AR no dispositivo do usuário. No segundo modelo, denominado *mid-offload*, as funções de extração de características, localização, geração de nuvem de pontos e detecção e rastreamento de objetos são executadas no servidor de borda, enquanto o dispositivo executa somente a captura de dados dos sensores e a renderização de imagens. Finalmente, no modelo *high-offload*, todas as funções são executadas no servidor de borda, exceto a captura de dados dos sensores, a qual é realizada pelo dispositivo. Uma vez que as arquiteturas *mid-offload* e *high-offload* movem funcionalidades computacionalmente intensivas do dispositivo para a borda da rede, elas abrem oportunidade para a economia de energia no dispositivo.

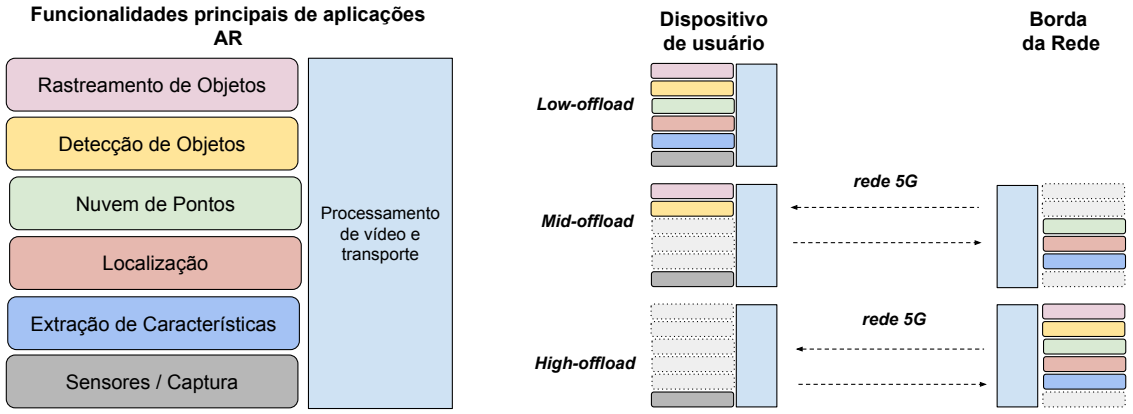
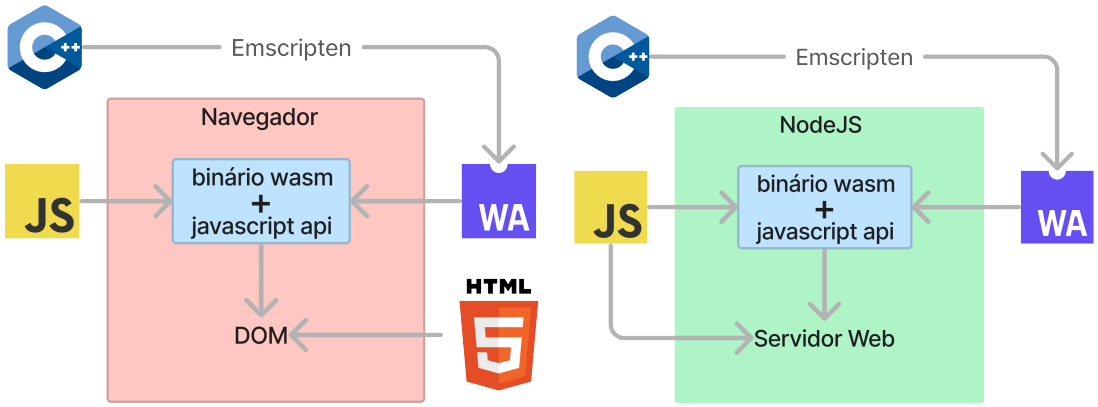


Figura 1. Opções para descarregamento de funcionalidade da realidade estendida. Adaptado de [Alriksson et al. 2021].

2.2. Fundamentos sobre WebAssembly

A utilização da Web tem passado por transformações significativas desde sua criação. Inicialmente, era predominantemente utilizada para a visualização estática de dados, oferecendo pouca interatividade. Com sua evolução, surge a demanda por uma interação mais intensa e colaborativa, levando ao desenvolvimento de linguagens como o JavaScript, que possibilita implementações mais dinâmicas no lado do cliente. No entanto, JavaScript é notoriamente difícil de compilar de forma eficiente [Selakovic and Pradel 2016]. Como consequência, aplicações escritas ou compiladas em JavaScript normalmente apresentam um desempenho muito inferior que seus equivalentes nativos. Para resolver esse problema, grandes empresas de tecnologia, como Google, Microsoft e Apple, se juntaram para desenvolver a linguagem WebAssembly.

WebAssembly, frequentemente referenciado como *Wasm*, é uma especificação de um formato binário (e sua semântica formal) que pode ser executada de várias maneiras, incluindo navegadores, *runtimes* independentes e interpretadores [Hoque and Harras 2022]. Em todos os métodos, o desenvolvedor implementa usando linguagens de programação de alto nível (e.g., C, C++, Go e Rust) e o código-fonte resultante é compilado em WebAssembly, gerando um arquivo binário (.wasm) que pode ser distribuído para ser executado



(a) Uso do Wasm no navegador.

(b) Uso do Wasm fora do navegador.

Figura 2. Exemplos de uso do WebAssembly.

em um dos métodos acima [Yan et al. 2021]. Além disso, soluções não baseadas em navegador (i.e., *runtimes* e interpretadores) suportam uma extensão chamada *WebAssembly System Interface* (WASI), a qual define uma interface padrão para o WebAssembly executar chamadas de sistema em diferentes plataformas, permitindo executáveis independentes. O WebAssembly também oferece forte isolamento e pouco consumo de memória, além de desempenho próximo ao desempenho do código nativo [Rossberg 2024].

De fato, a portabilidade concedida através desta independência de arquitetura e plataforma é uma das principais vantagens do WebAssembly. Isso significa que o código escrito para WebAssembly pode ser executado tanto no dispositivo do usuário, através de um navegador moderno, quanto num servidor de borda, sem dependência de um sistema operacional específico ou arquitetura de hardware subjacente. Esta característica é particularmente benéfica para a realização de *offloading* porque quando processos são descarregados de um dispositivo de usuário para servidores de borda, existe o obstáculo gerado pela diferença entre os tipo de arquiteturas e plataformas destes equipamentos. Na Figura 2, exemplificamos o poder do Wasm no contexto do *offloading* de aplicações Web. A Figura 2a mostra o uso de um código fonte C++ compilado para Wasm, de modo que esse só pode ser executado no navegador por uma API javasript, a qual se comunica com o binário WebAssembly. A Figura 2b ilustra como a mesma aplicação C++ poderia ser facilmente levada para um servidor de borda, executando dentro de um ambiente NodeJS, por meio da mesma API javascript.

3. Trabalhos Relacionados

Muitos trabalhos foram propostos na literatura tendo como foco o *offloading* de tarefas computacionalmente intensivas do dispositivo móvel para servidores localizados na nuvem ou na borda da rede. Muitos desses trabalhos exploram as facilidades da API Web Workers[1] em executar código JavaScript em segundo plano, sem bloquear a *thread* principal, para fazer a descarga de Web Workers para servidores remotos. Por exemplo, em [Jeong et al. 2019], os autores introduzem um mecanismo de *offloading* que permite que um Web Worker em execução migre de um dispositivo móvel para um servidor, usando o

[1] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API

módulo de subprocesso do Node.js para executar Web Workers no servidor. Mais tarde, em [Jeong et al. 2020], os mesmos autores estendem o trabalho anterior propondo uma abordagem de *offloading* dinâmico. Avançando o estado da arte no uso de Web Workers para o *offloading* de tarefas, em [Liu and You 2022], os autores implementam um arcabouço onde cada Web Worker que migra para o servidor remoto é executado em um ambiente JavaScript isolado, usando o motor V8 da linguagem. Diferentemente dos dois trabalhos anteriores, a abordagem proposta em [Liu and You 2022] é mais segura e não expõe o servidor remoto a vulnerabilidades. Uma desvantagem, no entanto, é que o desempenho da linguagem JavaScript é consideravelmente menor que o desempenho de código nativo.

O uso de WebAssembly como uma ferramenta de *offloading* de tarefas para a borda da rede foi investigado em [Nurul-Hoque and Harras 2021], onde os autores apresentam o Nomad, um ambiente que emprega o WebAssembly como a principal ferramenta para a execução de *offloading*. No Nomad, os autores incorporam uma funcionalidade de migração diretamente em um interpretador WebAssembly, possibilitando o descarregamento de tarefas em tempo real entre máquinas com arquitetura de hardware distintas e sistemas operacionais diversos. Uma limitação do trabalho, no entanto, é que, para testar o Nomad, os autores utilizam apenas um conjunto de *benchmarks* de CPU. Esses *benchmarks*, por sua vez, não possuem a complexidade de aplicações avançadas, como, por exemplo, aplicações de realidade estendida.

Alguns trabalhos começaram a investigar o uso de WebAssembly para computação de borda, focando especialmente no contexto de *serveless computing* [Gadepalli et al. 2019] [Wen and Weber 2020] [Hall and Ramachandran 2019]. Contudo, tal paradigma não é apropriado para o *offloading* de tarefas que mantém estados, como é o caso de um algoritmo de SLAM. A Tabela 1 resume as principais características dos trabalhos centrados no *offloading* de tarefas para a borda da rede. Como mostrado na tabela, ainda há uma lacuna na investigação do uso de WebAssembly para o *offloading* de tarefas computacionalmente intensivas, complexas e que mantém estado. Este trabalho visa preencher essa lacuna.

Trabalho	**Linguagem**	**Aplicação Alvo**
[Jeong et al. 2019]	JavaScript	*benchmark*
[Jeong et al. 2020]	JavaScript	Visão Computacional
[Liu and You 2022]	JavaScript	*benchmark*
[Nurul-Hoque and Harras 2021]	WebAssembly	*benchmark*

Tabela 1. Resumo de trabalhos sobre *offloading* para a borda da rede.

4. Estudo Experimental

Nesta seção, inicialmente apresentamos como foram implementados os dois modelos de *offloading* investigados neste trabalho. Em seguida, descrevemos a *testbed* utilizada nos experimentos, bem como os parâmetros de teste avaliados.

4.1. Modelos de *Offloading* Considerados

Como mencionado na Seção 2, as funções de extração de características, localização do dispositivo, geração de nuvem de pontos e rastreamento de objetos em uma aplicação

AR são normalmente providas por um algoritmo de SLAM. Portanto, para conduzirmos uma investigação experimental do uso de WebAssembly como ferramenta de *offloading* para aplicações AR, utilizamos uma biblioteca de terceiros, denominada AlvaAR, disponível em [Ross 2023]. Esta biblioteca implementa uma versão modificada de dois algoritmos de SLAM muito populares, OV^{2}SLAM [Ferrera et al. 2021] e ORB-SLAM2 [Mur-Artal et al. 2015], os quais foram modificados pelos autores para serem executados no navegador usando WebAssembly.

Em nosso estudo experimental, essa biblioteca é usada em uma aplicação que implementa o modelo *low-offload*, sendo, portanto, completamente executada no navegador. Utilizamos esta biblioteca também para implementar uma aplicação cliente-servidor que implementa o modelo *mid-offload*, ou seja, onde o algoritmo de SLAM executa em um servidor de borda, implementado em Node.js.

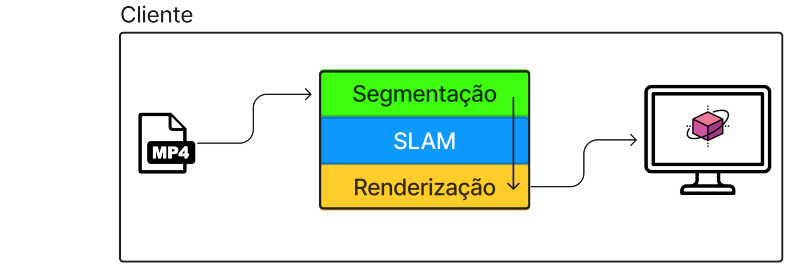
Como mencionado também na Seção 2, normalmente, numa aplicação AR, o usuário fornece acesso à câmera do seu dispositivo móvel e a aplicação processa em tempo real os quadros fornecidos pela câmera. Contudo, para fins de experimentação, nas duas implementações (*low-offload* e *mid-offload*) desenvolvidas neste trabalho, o cliente utiliza um vídeo pré-gravado no formato MP4. Essa decisão é importante porque, ao utilizarmos o mesmo vídeo, evitamos que tenhamos mais uma variável no experimento, podendo, por exemplo, influenciar no desempenho do SLAM. No restante deste trabalho, utilizamos o termo aplicação *low-offload* para nos referirmos à implementação que executa completamente no navegador e, o termo aplicação *mid-offload* para a implementação cliente-servidor.

Como ilustrado na Figura 3a, a aplicação *low-offload* foi dividida, primordialmente, em 3 fases principais:

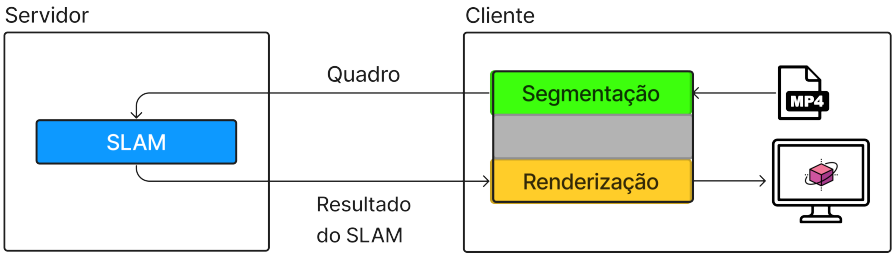
1. **Segmentação**: Fase em que um quadro é extraído de um video por meio de APIs de manipulação de imagem do navegador, via objeto *canvas*;
2. **SLAM**: Fase em que o quadro passa por um algoritmo que mapeia o ambiente e realiza a nuvem de pontos, com o uso da biblioteca AlvaAR;
3. **Renderização**: Fase em que é feita a renderização de um objeto virtual no quadro, por meio da biblioteca Tree.js.

Na aplicação *mid-offload*, ilustrada na Figura 3b, as fases de segmentação e renderização são realizadas no cliente, enquanto a fase de SLAM é executada no servidor. O cliente envia, via socket, os quadros extraídos do vídeo para o servidor. Por sua vez, o servidor (implementado em Node.js) processa cada quadro de vídeo recebido usando o algoritmo de SLAM implementado pela biblioteca AlvaAR, retornando para o cliente os dados da nuvem de pontos assim como as posições dos objetos virtuais. O cliente, então, realiza a renderização da cena usando o Tree.js. A comunicação entre o cliente e o servidor segue o modelo requisição-resposta, ou seja, o cliente fica bloqueado após enviar o quadro para o servidor.

Tanto na aplicação *low-offload* quanto na *mid-offload*, o resultado final exibido pelo cliente é uma página Web apresentando dois vídeos. O primeiro é uma superposição do vídeo original com objetos virtuais representando: (i) a nuvem de pontos calculada pelo algoritmo de SLAM e; (ii) cubos coloridos que foram adicionados sobre o vídeo original com o intuito de enriquecer a cena original. O segundo vídeo é um mapa 3D dos



(a) Aplicação *low-offload*.



(b) Aplicação *mid-offload*.

Figura 3. Modelos de *offloading* implementados no estudo experimental.

objetos virtuais adicionados na cena juntamente com um objeto que representa a câmera do usuário e seu campo de visão. Esse mapa é iterativo e é possível clicar e arrastar para visualizar outras partes do mapa.

4.2. *Testbed* e Parâmetros de Testes Utilizados

Para realizar os experimentos, definimos uma *testbed* com dois computadores, cada um com 8GB de memória RAM e CPU i7-3770 3.9GHz. Em um dos computadores, denominado cliente, instalamos o Ubuntu 18.04, o Google Chrome na versão 119, o *Node.js* na versão 17.9.1, a aplicação *low-offload* e o cliente da aplicação *mid-offload*. Na outra máquina, denominada servidora, instalamos o Ubuntu 22.04, o *Node.js* na versão 21.0.0 e o servidor da aplicação *mid-offload*. Ambas as máquinas foram conectadas diretamente, a fim de obter um ambiente de experimentação controlado. O vídeo usado para os testes tem duração de 60 segundos, uma resolução de 480x848 e é servido estaticamente pelo Express [2] na máquina cliente.

Para os testes com a aplicação *low-offload* (Figura 3a), os experimentos começam com o cliente abrindo automaticamente o Google Chrome com o uso da ferramenta Puppeteer[3]. Durante a execução da aplicação, coletamos os tempos necessários para extrair um quadro do vídeo (tempo de segmentação), processar o quadro do vídeo com o SLAM (tempo de SLAM) e renderizar o quadro (tempo de renderização). Coletamos também a taxa de quadros (FPS), a utilização média de CPU e o consumo médio de energia durante a execução da aplicação.

Para os testes com a aplicação *mid-offload* (Figura 3b), coletamos as mesmas métricas coletadas na aplicação *low-offload*. Por ser uma aplicação cliente-servidor, nesta versão, coletamos também o tempo gasto (apenas na rede) para enviar e receber cada qua-

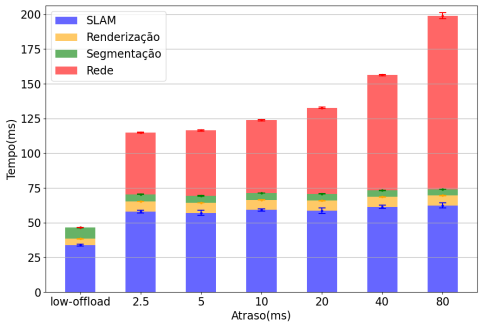
[2] `https://expressjs.com/pt-br/`
[3] `https://pptr.dev/`

dro do vídeo no cliente (tempo de rede). Adicionalmente, para os testes com a aplicação *mid-offload*, variamos alguns parâmetros da rede usando a ferramenta tc, a saber: (i) largura de banda, (ii) atraso e (iii) perda de pacotes. Esses parâmetros foram variados para avaliar como eles poderiam influenciar no *offloading* e, por consequência, na experiência do usuário. Os valores dos parâmetros utilizados nos testes estão apresentados abaixo. Destacamos que apenas um desses parâmetros foi aplicado por vez, ou seja, não foram feitos testes simulando atraso e perda de pacotes ao mesmo tempo na rede, por exemplo.

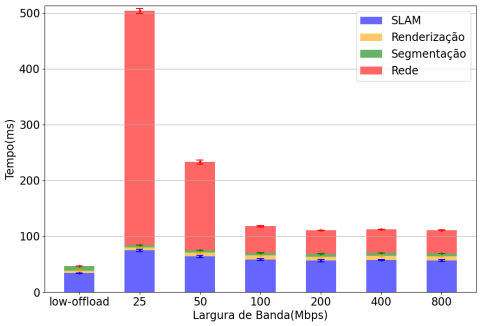
- **Largura de banda (Mbps):** 25, 50, 100, 200, 400, 800
- **Atraso (ms):** 2.5, 5, 10, 20, 40, 80
- **Perda de Pacotes (%):** 1, 2, 3, 4, 5, 6

5. Resultados

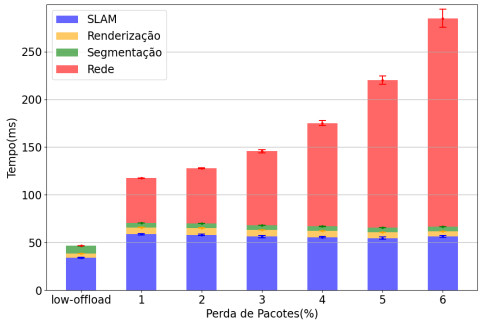
Os resultados apresentados na atual seção são a média de 30 experimentos, com um intervalo de confiança de 95%. A Figura 4 apresenta a comparação dos tempos de segmentação, SLAM, renderização e rede nas aplicações *low-offload* e *mid-offload*. As Figuras 4a, 4b e 4c apresentam os resultados obtidos variando o atraso, a largura de banda e a perda de pacotes, respectivamente. A primeira barra de cada gráfico é o tempo consumido pela aplicação *low-offload*, que não é afetado por nenhum parâmetro de rede, pois é uma execução local no cliente. Essa barra foi inserida para auxiliar na comparação com os tempos da aplicação *mid-offload* (demais barras).



(a) Variação do atraso de rede.



(b) Variação da largura de banda.



(c) Variação da perda de pacotes.

Figura 4. Tempos de *low* e *mid-offload*.

Nos gráficos da Figura 4, tanto o tempo de segmentação quanto o de renderização da aplicação *mid-offload* não foram afetados de forma significativa pelas variações de

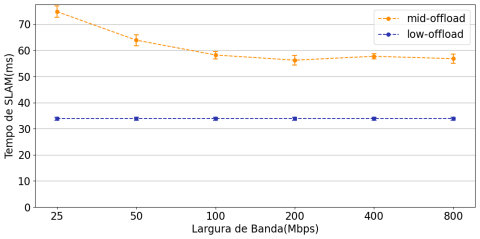


Figura 5. Tempo de SLAM.

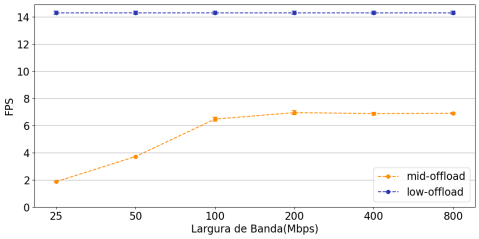


Figura 6. Quadros por segundo (FPS).

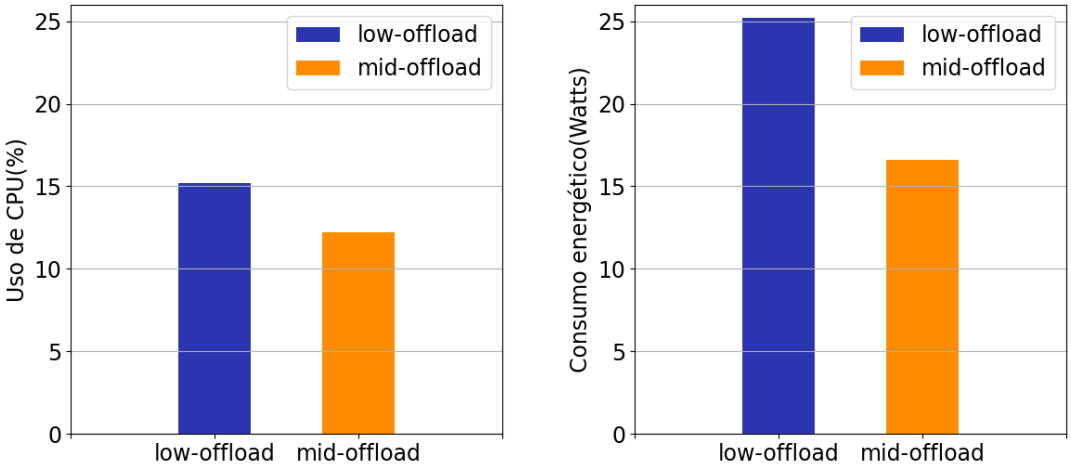
rede. Além disso, eles tiveram um desempenho semelhante ao da aplicação *low-offload*. Esses tempos também interferem pouco no desempenho total da aplicação, que é dominado pelo tempo de SLAM e rede. Podemos perceber que o tempo de rede segue um comportamento esperado, ou seja, o tempo de rede diminui conforme a qualidade da rede aumenta. Porém, na Figura 4b ocorre um comportamento diferente: no intervalo entre 200Mbps e 800Mbps, o tempo de rede fica constante, em torno de 50ms. Isso é um indício de que o programa enfrentou outro gargalo diferente da largura de banda. Possivelmente, nessas configurações, a limitação de CPU (processando o algoritmo de SLAM) se torna o novo limitante para o consumo de mais quadros de vídeo e com isso a rede fica ociosa.

Outro destaque é o tempo de execução do algoritmo de SLAM na aplicação *mid-offload*, que apresentou desempenho inferior quando comparado com a aplicação *low-offload*. Esse resultado fica mais evidente na Figura 5, onde ilustramos apenas o de tempo de SLAM nas duas aplicações à medida que variamos a largura de banda da rede nos experimentos com a aplicação *mid-offload*. Como o SLAM é executado no servidor na aplicação *mid-offload*, houve a suspeita de que a mudança do ambiente (do navegador Chrome para o Node.js) pudesse ter afetado o processamento do WebAssembly (biblioteca AlvaAR). Realizamos testes no Node.js semelhante ao do navegador, ou seja, sem transmissão de rede, apenas processando localmente o vídeo. Nesta situação, os tempos de SLAM das duas aplicações ficaram similares. Dessa forma, a suspeita é que, ao migrarmos o SLAM para um serviço Web executando Socket.io, introduzimos alguma concorrência de entrada e saída em conjunto com a tarefa de SLAM (que é CPU intensiva). A suspeita sobre concorrência com a rede fica maior ao observar que quando a largura de banda é menor (25Mbs e 50Mbs), o tempo de SLAM é mais impactado. A própria documentação do Node.js recomenda o uso de *Workers* (threads) no caso de tarefas que consomem muita CPU, mas devido ao tempo, optamos por deixar essa nova implementação da aplicação *mid-offload* usando *Workers* como trabalhos futuros.

A Figura 6 mostra a taxa de quadros (FPS) obtida nas duas aplicações quando variamos a largura de banda da rede. Como mencionado anteriormente, a aplicação *low-offload* não é afetada por nenhum parâmetro de rede, pois é uma execução local no cliente. No entanto, ela foi inserida para auxiliar na comparação com os tempos da aplicação *mid-offload*. A métrica de FPS é importante para as aplicações AR para garantir a fidelidade do vídeo apresentado com o que se espera da realidade. Ela também é uma medida que representa o desempenho geral da aplicação. O resultado encontrado indica o alto impacto dos tempos de rede e de SLAM no resultado final da aplicação. Isso é esperado, uma vez

que eles representam cerca de 90% do tempo total para processamento de um quadro do vídeo. Ademais, fica claro pela figura que, após a marca de 100Mbps, o desempenho da aplicação fica estagnado, consequência direta do gargalo de processamento discutido anteriormente.

Na Figura 7, apresentamos uma comparação da utilização de CPU e consumo de energia da aplicação *low-offload* e do cliente da aplicação *mid-offload*. Para o cálculo do consumo de CPU nós realizamos leituras a cada 1 segundo do arquivo */proc/stat* e calculamos a média ao final do experimento. A redução no consumo geral de CPU apresentada no cliente da aplicação de *mid-offload* já era esperada, visto que a funcionalidade de SLAM foi movida do cliente para o servidor. Podemos notar que essa redução do uso de CPU impactou positivamente no consumo de energia (Figura 7b), o qual também foi reduzido. Para a coleta do consumo de energia nós empregamos a ferramenta Powerstat[4].



(a) Consumo de CPU. **(b) Consumo de energia.**

Figura 7. Experimento de CPU e energia na aplicação *low-offload* e *mid-offload*.

6. Conclusões e Trabalhos Futuros

Neste trabalho, apresentamos um estudo experimental quantitativo do desempenho de aplicações AR usando diferentes abordagens de *offloading* e também diferentes configurações na rede entre o cliente e o servidor. A utilização do WebAssembly permitiu a execução do algoritmos de SLAM diretamente no navegador, além de permitir que a mesma biblioteca de SLAM fosse utilizada no servidor, facilitando assim o desenvolvimento e permitindo que o algoritmo fosse avaliado em diferentes ambientes.

Os experimentos com a aplicação *low-offload* mostraram que a abordagem de executar todo o processamento no navegador do cliente pode ser eficaz em termos de desempenho, proporcionando uma experiência de AR com uma maior taxa de quadros por segundo. Enquanto isso, os experimentos com a aplicação *mid-offload* destacaram a influência da qualidade da rede no desempenho da aplicação AR. O tempo de transferência de dados entre o cliente e o servidor, especialmente em um ambiente de rede

[4]https://github.com/ColinIanKing/powerstat

com baixa qualidade, teve um alto impacto na taxa de quadros por segundo da aplicação. Além disso, foi possível perceber que aumentar a qualidade da conexão com o servidor não é o bastante para otimizar a aplicação. Também é fundamental que os recursos de CPU e memória sejam proporcionais à qualidade da rede, a fim de evitar outros gargalos.

Esses experimentos também trazem a conclusão de que o *offloading* é uma opção viável para reduzir o consumo de energia do cliente, permitindo, assim, a execução de aplicações que demandam grande desempenho, como AR, em dispositivos móveis durante longos períodos de tempo. Em geral, o uso do WebAssembly no contexto de uma ferramenta para o *offloading* de aplacações Web, precisa de mais investigação e atenção, ainda mais no cenário de aplicações emergentes tais como AR.

Como trabalhos futuros, pretendemos analisar o modelo *high-offload*. Neste trabalho, exploramos somente os modelos *low-offload* e *mid-offload*, mas com algumas adaptações seria possível criar uma versão com mais funcionalidades sendo desempenhadas pelo servidor. O desafio em questão seria migrar o processo de renderização para o servidor a fim de minimizar o uso de recursos do cliente. Além disso, pretendemos implementar e avaliar a aplicação *mid-offload* usando Workers para entender melhor qual interferência ocasionou no maior tempo de processamento no servidor. Para uma aplicação que usa o modelo cliente-servidor ser viável, é necessário que o servidor consiga lidar com uma alta carga de trabalho, preferencialmente sendo capaz de atender múltiplos clientes ao mesmo tempo. É um desafio fazer isso com uma aplicação AR devido ao uso intenso de recursos computacionais. Seria interessante entender melhor qual interferência ocasionou no maior tempo de processamento no servidor.

Agradecimentos

Os autores agradecem o apoio do Centro de Pesquisa em Engenharia (CPE) SMART NEtworks and ServiceS for 2030 (SMARTNESS) (FAPESP, Processo 21/00199-8).

Referências

Alriksson, F., Kang, D. H., Phillips, C., Pradas, J. L., and Zaidi, A. (2021). Xr and 5g: Extended reality at scale with time-critical communication. *Ericsson Technology Review*, 2021(8):2–13.

Bekele, M. K., Pierdicca, R., Frontoni, E., Malinverni, E. S., and Gain, J. (2018). A survey of augmented, virtual, and mixed reality for cultural heritage. *J. Comput. Cult. Herit.*, 11(2).

Cao, J., Lam, K.-Y., Lee, L.-H., Liu, X., Hui, P., and Su, X. (2023). Mobile augmented reality: User interfaces, frameworks, and intelligence. *ACM Comput. Surv.*, 55(9).

Ferrera, M., Eudes, A., Moras, J., Sanfourche, M., and Le Besnerais, G. (2021). OV^{2}SLAM : A fully online and versatile visual SLAM for real-time applications. *IEEE Robotics and Automation Letters*.

Gadepalli, P. K., Peach, G., Cherkasova, L., Aitken, R., and Parmer, G. (2019). Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615.

Gapeyenko, M., Petrov, V., Paris, S., Marcano, A., and Pedersen, K. I. (2023). Standardization of extended reality (xr) over 5g and 5g-advanced 3gpp new radio. *IEEE Network*, 37(4):22–28.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200.

Hall, A. and Ramachandran, U. (2019). An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, page 225–236, New York, NY, USA. Association for Computing Machinery.

Hoque, M. N. and Harras, K. A. (2022). Webassembly for edge computing: Potential and challenges. *IEEE Communications Standards Magazine*, 6(4):68–73.

Jangda, A., Powers, B., Berger, E. D., and Guha, A. (2019). Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA. USENIX Association.

Jeong, H.-J., Jeong, I., and Moon, S.-M. (2020). Dynamic offloading of web application execution using snapshot. *ACM Trans. Web*, 14(4).

Jeong, H.-J., Shin, C. H., Shin, K. Y., Lee, H.-J., and Moon, S.-M. (2019). Seamless offloading of web app computations from mobile device to edge clouds via html5 web worker migration. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 38–49, New York, NY, USA. Association for Computing Machinery.

Liu, A.-C. and You, Y.-P. (2022). Offworker: An offloading framework for parallel web applications. In Chbeir, R., Huang, H., Silvestri, F., Manolopoulos, Y., and Zhang, Y., editors, *Web Information Systems Engineering – WISE 2022*, pages 170–185, Cham. Springer International Publishing.

Morín, D. G., Pérez, P., and Armada, A. G. (2022). Toward the distributed implementation of immersive augmented reality architectures on 5g networks. *IEEE Communications Magazine*, 60(2):46–52.

Mur-Artal, R., Montiel, J. M. M., and Tardós, J. D. (2015). ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5):1147–1163.

Nurul-Hoque, M. and Harras, K. A. (2021). Nomad: Cross-platform computational offloading and migration in femtoclouds using webassembly. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 168–178.

Reitmayr, G., Langlotz, T., Wagner, D., Mulloni, A., Schall, G., Schmalstieg, D., and Pan, Q. (2010). Simultaneous localization and mapping for augmented reality. In *2010 International Symposium on Ubiquitous Virtual Reality*, pages 5–8.

Ross, A. (2023). Alvaar. Disponível em `https://github.com/alanross/AlvaAR`.

Rossberg, A. (2024). WebAssmbly Specification. Technical report, WebAssembly Community Group.

Selakovic, M. and Pradel, M. (2016). Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software*

Engineering, page 61–72, New York, NY, USA. Association for Computing Machinery.

Siriwardhana, Y., Porambage, P., Liyanage, M., and Ylianttila, M. (2021). A survey on mobile augmented reality with 5g mobile edge computing: Architectures, applications, and technical aspects. *IEEE Communications Surveys & Tutorials*, 23(2):1160–1192.

Toczé, K., Lindqvist, J., and Nadjm-Tehrani, S. (2019). Performance study of mixed reality for edge computing. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, page 285–294, New York, NY, USA. Association for Computing Machinery.

Wen, E. and Weber, G. (2020). Wasmachine: Bring the edge up to speed with a webassembly os. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 353–360.

Yan, Y., Tu, T., Zhao, L., Zhou, Y., and Wang, W. (2021). Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 533–549, New York, NY, USA. Association for Computing Machinery.