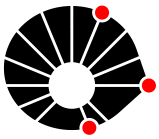
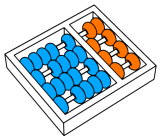




Análise de offloading em aplicações móveis

S. E. Kim *L.F. Bittencourt*

Relatório Técnico - IC-PFG-20-17
Projeto Final de Graduação
2020 - Agosto

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Análise de offloading em aplicações móveis

Seong Eun Kim*

Resumo

Este trabalho tem como objetivo realizar estudos em *offloading* em aplicações móveis. O *offloading* consiste em transferir computações intensivas para um outro processador, seja por causa de limitações do dispositivo ou para melhorar performance, consumo de bateria ou memória. Para a realização dos estudos, foi implementada uma aplicação Android que faz a computação de um filtro em uma imagem usando CPU ou GPU. Essa computação foi executada localmente e em servidores em nuvem do Google Cloud. A partir disso, pode-se realizar experimentos para analisar os casos em que seria vantajoso ou desvantajoso o *offloading* ser aplicado ao se comparar a latência, o consumo de CPU e de bateria.

1 Introdução

Nas últimas duas décadas, houve grandes avanços tecnológicos envolvendo dispositivos móveis, marcados principalmente pelo surgimento dos smartphones e de outros computadores inteligentes. Entretanto, tais dispositivos ainda são muito limitados em termos de processamento, bateria e memória. Além disso, os aplicativos neles contidos exigem cada vez mais poder computacional, principalmente com o advento da realidade aumentada e inteligência artificial. O rápido desenvolvimento dessas tecnologias tornou o desenvolvimento de software voltado a aplicações móveis e uma mudança no perfil dos usuários, os quais se tornaram mais exigentes quanto a latência e a performance de tais aplicações. [7].

A computação em nuvem provê serviços sob demanda - como servidor, armazenamento, software, *analytics* e inteligência - pela internet. Através dela, os computadores são mais independentes do limite de memória e processamento para realizar computações em curto prazo de tempo. Nesse sentido, integrando-se a computação em nuvem aos dispositivos móveis, surge o conceito de computação em nuvem móvel (*Mobile Cloud Computing*), que aumenta a capacidade dos dispositivos móveis, abrindo portas para que aplicações mais exigentes executem em dispositivos de baixo custo.

O processo de usar a computação em nuvem em dispositivos móveis requere o particionamento da aplicação, o *offloading* de parte da computação e uma distribuição das tarefas a serem executadas [3]. Deve-se levar em consideração também um ponto negativo da computação em nuvem, que é a latência. A depender da aplicação, esta pode comprometer significativamente sua performance e trazer resultados indesejados. Estudos mostram que

* Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

devido à alta latência de rede e à pequena largura de banda, nem sempre é adequado fazer o *offload* para a nuvem [4], [5], [6]. No entanto, essa decisão não é direta: ela depende de variáveis do ambiente. Assim, ela deve ser feita dinamicamente e adaptativamente para cada parte da aplicação.

Na literatura, há vários estudos que tratam as limitações dos dispositivos móveis com *offloading* computacional para a nuvem [8]. Entretanto, a nuvem não é necessariamente uma panaceia para aumentar a capacidade desses dispositivos. Pesquisas mostram que a computação do aplicativo de identificação facial Picaso, por exemplo, demora mais executando-se na nuvem do que localmente. Essa sobrecarga se deve, principalmente, pelas condições de rede (largura de banda e latência).

Para tratar essa restrição da rede, uma abordagem possível é o *fog computing*. Este consiste em distribuir a computação para bordas descentralizadas, próximo a fonte de dados. Assim, os servidores ficam mais próximos dos usuários, e a latência é significativamente reduzida.

Tendo em vista esses problemas, nesse trabalho analisamos a técnica de *offloading* em aplicações móveis em dispositivos Android. Mostramos as vantagens e desvantagens de se aplicar a técnica, baseadas em métricas-chave como consumo de bateria e latência, e como a distância usuário-servidor influencia na efetividade do *offloading*. Descrevemos os cenários de teste, os experimentos que foram realizados e como os resultados podem viabilizar o desenvolvimento de novas estratégias de *offloading* na névoa.

2 Trabalhos Relacionados

As aplicações em dispositivos móveis abrangem desde o entretenimento a uma série de serviços que solucionam problemas cotidianos ou que facilitam o dia-a-dia dos usuários. No entanto, a bateria, a largura de rede e a CPU dos dispositivos móveis sempre foram uma limitação no desenvolvimento e na utilização dessas aplicações. Em busca de resolver esse problema, em 2007, Balan et al. fez um estudo sobre *cyber foraging* para dispositivos móveis [9]. Eles propunham que o dispositivo móvel realizasse a execução remota em um servidor próximo através de uma conexão sem fio. Para isso, desenvolveram um programa que usa *little languages*[11] capaz de modificar aplicações para que usem da técnica de *cyber foraging*.

Nessa mesma linha de trabalho, em 2011, Chun et al. criou CloneCloud, um sistema que particiona aplicações móveis e que é um ambiente de execução para aplicações não modificadas que estão rodando em uma máquina virtual em nível de aplicação. Ele permite que elas façam *offloading* de parte da execução para dispositivos clones, os quais estão operando em nuvem [10]. Através disso, CloneCloud foi capaz de acelerar em até 20 vezes a execução e diminuir em até 20 vezes a energia gasta no computador móvel.

Além dos mencionados, houve vários outros estudos sobre aumentar a capacidade dos dispositivos através da computação na nuvem [15], [16], [17]. Entretanto, esta também é limitada a condições do dispositivo e do ambiente, como largura de banda e latência, e, por isso, não deve ser a solução imediata nas aplicações, principalmente às que são sensíveis ao tempo. Tendo em vista isso, em 2015, Ra et al. criaram Odessa, ambiente de

execução que automaticamente e adaptativamente faz decisões de *offloading* e paralelismo para aplicações móveis interativas de percepção [8]. Seus experimentos mostraram que as decisões de *offloading* e o nível de paralelismo de dados ou de *pipelines* não podem ser determinados estatisticamente, mas precisam ser adaptadas em tempo de execução. Isso se deve pois a capacidade de resposta e acurácia variam drasticamente a depender da largura de banda, tamanho da entrada e outras características do dispositivo no tempo de execução.

Considerando as limitações do *offloading* computacional de dispositivos móveis para a nuvem, Hassan et al. fizeram um estudo sobre *offloading* e armazenamento em névoa [3]. Nas simulações feitas usando aplicativos computacionalmente intensos (Picaso [13], de reconhecimento facial, e DroidSlator [14], dicionário do Android) em diferentes ambientes, o resultado, como esperado, mostra que a proximidade desses serviços com o dispositivo móvel garantiu melhor performance em relação aos serviços em nuvem - o tempo de resposta e a energia consumida foram menores.

Apesar dos resultados dos estudos serem positivos, Hassan et al. também levanta desafios da computação em névoa - disponibilidade e segurança. Assim, concluímos que a depender da aplicação, considerar o uso de *offloading* para servidores na névoa pode ser uma opção para uma melhor performance em relação a computação local ou em nuvem.

Diante das diversas variáveis envolvidas quanto a fazer ou não o *offloading* e em qual servidor fazer, Yu et al. fizeram uma ferramenta de *offloading* dinâmico, usando *deep learning* [12]. Ela considera a sobrecarga local no dispositivo e as limitações dos recursos de comunicação e de computação da rede. Yu et al. abordaram o problema de decisão de *offloading* através de classificação multi-rótulo e implementaram um método de *Deep learning* supervisionado. Desse modo, obtiveram resultados 49.24% melhores do que não fazer *offloading*, 15.69% melhores do que só fazer o *offloading* e 11.18% melhores do que o *offloading* baseado numa classificação linear multi-rótulo.

3 Definições

3.1 Computação em nuvem

Computação em nuvem consiste na entrega de serviços sob demanda - de funcionalidades de aplicativos a armazenamento e poder de processamento - geralmente através da internet e com pagamento conforme o uso. Ela funciona através de uma *pool* de recursos virtual, em que o provedor realiza as requisições usando automatizações. A vantagem principal é a agilidade - a habilidade de aplicar computação abstraída, armazenamento e recursos de rede para cargas de trabalho conforme necessário e explorar uma abundância de serviços pré-construídos [19].

Um benefício de se usar computação em nuvem é que ela permite que empresas possam evitar o custo inicial e a complexidade de possuir e manter a própria infraestrutura de TI, e que possam, ao invés disso, simplesmente pagar pelo que usam e quando usam. Em troca, provedores de servidores em nuvem podem se beneficiar de obter economias significativas de escala, por oferecerem os mesmos serviços a uma ampla gama de clientes [18].

Os serviços oferecidos pelos servidores em nuvem são variados, como por exemplo, aplicações, armazenamento, rede, capacidade de processamento, processamento de lingua-

gem natural, inteligência artificial, entre outros.

3.2 Computação em névoa

Computação em névoa é uma infraestrutura computacional descentralizada na qual os dados, computação, armazenamento e aplicações estão localizadas entre a fonte de dados e a nuvem. Similar a computação em borda, a computação em névoa traz as vantagens e a capacidade da nuvem para mais perto de onde o dado é criado e usado [21].

A metáfora da *névoa* vem do termo metereológico para uma nuvem próxima ao solo, assim como a computação em névoa se concentra nas bordas da rede.

O desenvolvimento de ferramentas de computação em névoa dá às organizações mais alternativas para processar dados onde for mais apropriado. Além disso, ela dá a possibilidade de criar conexões de rede com baixa latência entre dispositivos e *endpoints* de *analytics*. Por sua vez, essa arquitetura reduz a largura de banda necessária comparada ao caso em que o dado precisaria ser enviado para o centro de processamento de dados ou para a nuvem para ser processado [20]. A computação em nuvem pode também ser implantada por razões de segurança, pois ela permite segmentar o tráfego da largura de banda, e adicionar *firewalls* na rede para melhor segurança [22].

É importante notar que a computação em névoa complementa a computação em nuvem - a primeira permite *analytics* de curto prazo na borda, e a última realiza *analytics* de longo prazo, com uso intenso de recursos. Aplicações populares de computação em névoa incluem *smart grid*, cidades inteligentes, edifícios inteligentes e redes definidas por software (SDN) [23].

3.3 Offloading

Offloading em computação consiste na transferência de computações intensas para um processador separado, como um acelerador de hardware, ou uma plataforma externa, como em um *cluster* ou em uma nuvem. Ele possibilita o aumento da capacidade de processamento e elimina limitações de hardware, como armazenamento, processamento e energia. Por isso, é bastante usada por dispositivos móveis e inteligentes.

A implementação do *offloading* consiste em três partes: encontrar um servidor disponível; estabelecer conexão segura com o servidor; e o particionamento da aplicação. A maioria das técnicas de *offloading* requerem que os desenvolvedores manualmente programem as funções necessárias para executar em outro dispositivo. Entretanto, usar análises de código estático e dinâmico podem ser uma alternativa que resulta em mais adaptatividade para flutuações de rede e aumento de latência [24].

4 Metodologia

Nesse projeto, analisaremos o desempenho de um aplicativo que aplica um filtro sobre uma imagem em um dispositivo móvel Android.

4.1 Experimentos com CPU

Primeiro, uma biblioteca compartilhada foi desenvolvida em Java. A biblioteca é responsável por aplicar um filtro a uma imagem – um Bitmap é recebido como entrada, e um Bitmap é gerado como saída, onde cada pixel da imagem de entrada passa por uma transformação. A cor resultante do pixel é definida como a média dos três canais de cor RGB, em escala de cinza, efetivamente sendo uma espécie de filtro preto e branco.

A biblioteca foi desenvolvida de forma que pudesse operar tanto sobre as bibliotecas-padrão de imagem Android, tanto sobre aquelas de Java para PC. Ou seja, o filtro pode ser executado através de um mesmo algoritmo em ambos os ambientes, porém sobre as primitivas Bitmap que cada um deles implementa.

Após isso, implementamos um servidor em Java, utilizando Spring[1] – um framework de código aberto responsável por simplificar a implementação de uma API RESTful. Implementamos um *endpoint* `"POST: /image/"`, que recebe os *bytes* de um Bitmap, aplica o filtro através da biblioteca compartilhada e retorna na requisição os *bytes* do Bitmap resultante. Isso permite que o aplicativo Android possa fazer *offloading* do filtro através de uma simples requisição HTTP.

Por fim, foi desenvolvida uma aplicação em Kotlin, que possui uma imagem fixa, sobre a qual o filtro será aplicado; um botão para executar o filtro localmente (no dispositivo, através de uma chamada direta à biblioteca); um botão para executar o filtro remotamente (através de uma requisição HTTP para um servidor em nuvem); um botão para executar experimentos (vide Seção 5). A captura de tela da aplicação pode ser vista na Figura 1, e a implementação pode ser encontrada em [1].

A partir disso, conduzimos alguns experimentos em um dispositivo Google Pixel 3a. Foram definidos alguns cenários para análise:

1. O filtro é aplicado localmente, sem nenhum tipo de *offloading* – a CPU do dispositivo é responsável pelo processamento;
2. O filtro é *offloaded*, sendo aplicado por um servidor em nuvem, situado em São Paulo, no mesmo país em que o dispositivo está situado;
3. O filtro é *offloaded*, sendo aplicado por um servidor em nuvem, situado em Oregon (US), em um continente diferente de onde o dispositivo está situado.

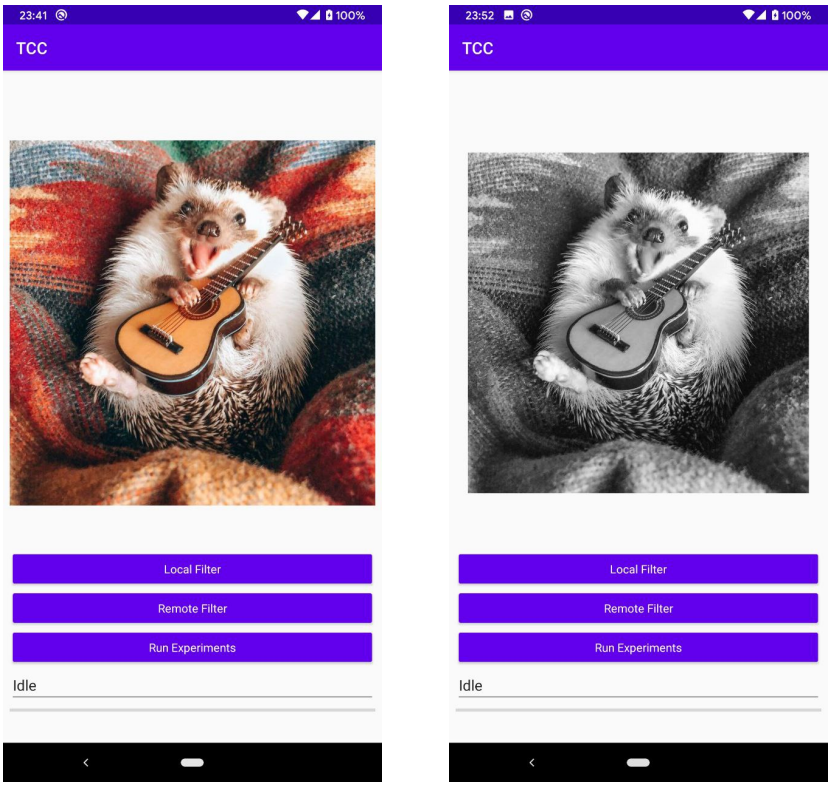
Além disso, para cada um dos cenários, aplicamos o filtro em versões redimensionadas da imagem original, variando entre 180p e 1080p.

O modelo de instância do Google Cloud que foi escolhido para rodar os servidores foi o `n1-highcpu-4`, que provê 4 *virtual CPUs* e 3.6 GB de memória RAM.

A escolha dos cenários visa exercitar nos experimentos as particularidades dos métodos de *offloading* e de computação em névoa, como:

- Latência incorrida pela distância cliente-servidor;
- Tráfego de rede incorrido pela transmissão das imagens;

[1]Spring. https://spring.io/



(a) Imagem antes de aplicar o filtro. (b) Imagem depois de aplicar o filtro.

Figura 1: Captura de tela da aplicação implementada.

- Consumo de bateria incorrido pelo processamento no dispositivo.

Nos cenários aplicáveis, procuramos computar métricas como: latência de rede, tempo real gasto durante a aplicação do filtro, tempo de CPU gasto e consumo de bateria. Todas as métricas de tempo foram computadas através de um código automatizado responsável por executar os experimentos pré-definidos em um arquivo de configuração.

Já o consumo de bateria foi computado através do comando `batterystats` do *Android Debug Bridge*[2], uma ferramenta provida junto ao kit de desenvolvimento Android que permite que certos dados de execução sejam capturados durante a execução do programa. Tais dados de consumo de bateria são estimativas do consumo real providas pela ferramenta, pois o dispositivo Android utilizado não suporta a captura de métricas exatas.

4.2 Experimentos com GPU

Para fazermos experimentos com GPU, realizamos algumas incrementações no código que foi desenvolvido para a CPU. Implementamos um filtro preto e branco no dispositivo, usando

[2]Android Debug Bridge. https://developer.android.com/studio/command-line/adb

OpenGL, através da biblioteca GPUImage[3]. Para o servidor, implementamos o mesmo filtro através de um *shader* em OpenGL.

A partir disso, conduzimos alguns experimentos no mesmo dispositivo (Google Pixel 3a) com os seguintes cenários:

1. O filtro é aplicado localmente, sem nenhum tipo de *offloading* – a GPU do dispositivo é responsável pelo processamento;

2. O filtro é *offloaded*, sendo aplicado por um servidor em nuvem, situado em São Paulo, no mesmo país em que o dispositivo está situado;

3. O filtro é *offloaded*, sendo aplicado por um servidor em nuvem, situado em Oregon (US), em um continente diferente de onde o dispositivo está situado.

Para cada um desses cenários, aplicamos o filtro em versões redimensionadas da imagem original, variando entre 180p e 2160p.

O modelo de instância do Google Cloud que foi escolhido para executar os servidores foi o `n1-highcpu-4`, que provê 4 *virtual CPUs* Intel Skylake, 16GB de memória RAM e GPU Intel Graphics HD.

Nos cenários aplicáveis, procuramos computar métricas como: latência de rede, tempo real gasto durante a aplicação do filtro e tempo de GPU gasto. Todas as métricas de tempo foram computadas através de um código automatizado responsável por executar os experimentos pré-definidos em um arquivo de configuração.

5 Avaliação

Nesta Seção, avaliaremos os resultados obtidos a partir dos experimentos descritos na Seção 4.

5.1 Análise do tempo total de execução

5.1.1 Experimentos com CPU

Ao executarmos diversas iterações dos diferentes cenários para diferentes tamanhos de imagem, obtivemos os valores da Tabela 1, e fizemos algumas análises sobre os resultados dos experimentos.

A partir dessa tabela, plotamos um gráfico que pode ser visto na Figura 2.

Podemos notar os impactos, em termos de latência, que optar por *offloading* causam. Em particular, quando a imagem a ser processada é pequena, a sobrecarga incorrida pelo *offloading* pode tornar indesejável o uso da técnica. Porém, para imagens maiores, a latência tende a diminuir e, se o servidor estiver perto o suficiente do dispositivo (na borda), pode-se obter até latências menores usando *offloading* (vide valores para imagens a partir de 720p). Isso provavelmente se deve ao fato da máquina da nuvem ser muito mais poderosa que o dispositivo móvel.

[3]Android GPUImage https://github.com/cats-oss/android-gpuimage

Tabela 1: Valores referentes ao tempo médio de execução do filtro (em milissegundos) para cada cenário (coluna) e tamanho de imagem (linha), ao longo de 200 execuções do filtro.

	Local	São Paulo	Oregon (US)
180p	**70.21± 0.1**	339.08 ± 33.5	1153.26 ± 160.1
360p	**276.57 ± 0.2**	748.36 ± 51.4	2781.07 ± 246.6
720p	2384.85 ± 45.7	**1983.57 ± 173.2**	4302.90 ± 519.6
1080p	5426.86 ± 53.0	**4089.72 ± 165.9**	8544.29 ± 531.4

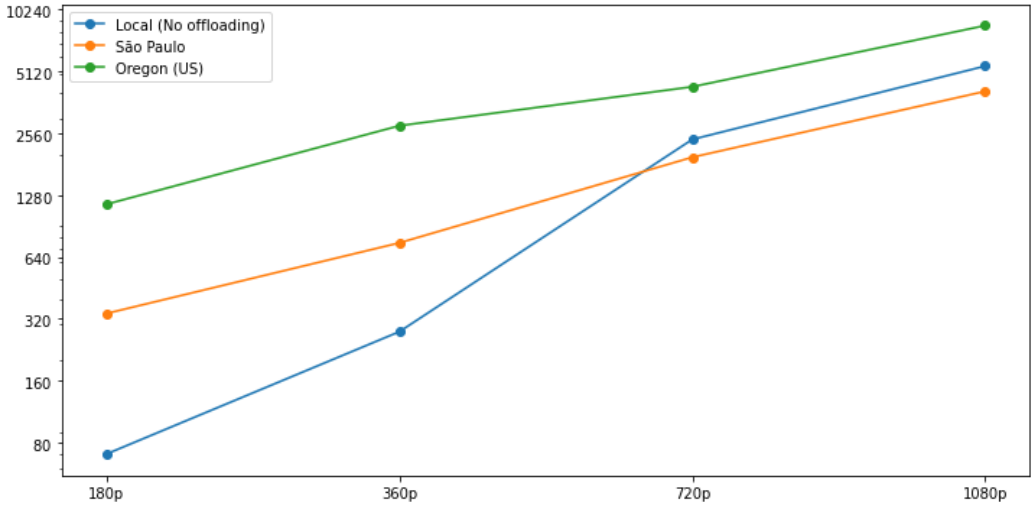


Figura 2: O tempo médio consumido (milissegundos) pela execução do filtro para cada cenário em relação ao tamanho de imagem (ambos em escala logarítmica), ao longo de 200 execuções do filtro na CPU.

5.1.2 Experimentos com GPU

Para o caso com GPU, executamos diversas iterações dos diferentes cenários e tamanhos de imagem e obtivemos os valores da Tabela 2.

Tabela 2: Valores referentes ao tempo médio de execução do filtro (em milissegundos) para cada cenário (coluna) e tamanho de imagem (linha), ao longo de 200 execuções do filtro.

	Local	São Paulo	Oregon
180p	**14.96 ± 0.9**	390.00 ± 65.5	835.23 ± 99.2
360p	**19.07 ± 0.6**	836.52 ± 81.6	2552.53 ± 312.8
540p	**29.21 ± 0.8**	1338.04 ± 68.0	3543.47 ± 589.3
720p	**33.00 ± 1.3**	1910.42 ± 84.5	3930.15 ± 150.5
1080p	**66.12 ± 1.9**	3037.98 ± 170.8	5327.73 ± 124.9
1440p	**105.05 ± 2.4**	3842.22 ± 82.7	8464.65 ± 422.7
2160p	**219.89 ± 3.1**	6244.68 ± 139.4	12137.42 ± 565.9

A partir dessa tabela, plotamos um gráfico, como pode ser visto na Figura 3.

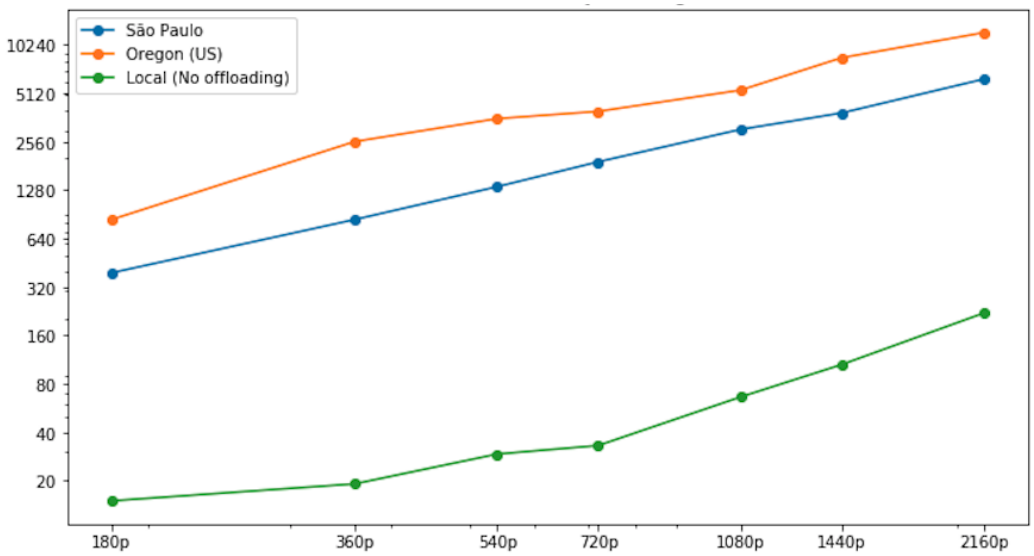


Figura 3: O tempo médio consumido (milissegundos) pela execução do filtro para cada cenário em relação ao tamanho de imagem (ambos em escala logarítmica), ao longo de 200 execuções do filtro na GPU.

Podemos notar uma grande discrepância na latência entre a execução do filtro localmente e em nuvem. Para todos os tamanhos de imagens testados, o processamento local foi muito mais vantajoso do que o *overhead* de mandar as imagens para a nuvem e a latência de rede. É importante ressaltar que a GPU do Pixel 3a (Adreno 615) é focado em processamento de imagem, então a sua performance é melhor em relação a outros dispositivos *low-end*.

Além disso, ao compararmos as latências médias de *offloading* com CPU e GPU, observamos que os valores com GPU são, em geral, menores do que com CPU. Assim, para processamentos mais pesados, que exigem mais computadores ou paralelismo, como o *deep learning*, é mais vantajoso fazer *offloading* e executar com GPU ao invés da CPU.

5.2 Análise do consumo de CPU

Uma outra métrica interessante quando se fala em *offloading* é o consumo de CPU, que está correlacionado ao consumo de bateria – é comum aplicações móveis serem desenvolvidas com a otimização do consumo de CPU em mente e, em alguns casos, o *offloading* se mostra como uma estratégia válida. A Figura 4 mostra como o consumo de CPU varia a depender do cenário e do tamanho da imagem processada.

Levando em consideração apenas o tempo de processamento da CPU do dispositivo Android, observamos que, para imagens de tamanho 180p, o tempo médio de processamento é equivalente ao tempo de se fazer o *offloading*. Além disso, para imagens menores do que 180p, é mais rápido executar o filtro localmente. Isso se deve por causa do *overhead* causado por transportar os dados da imagem para a realização do processamento remoto –

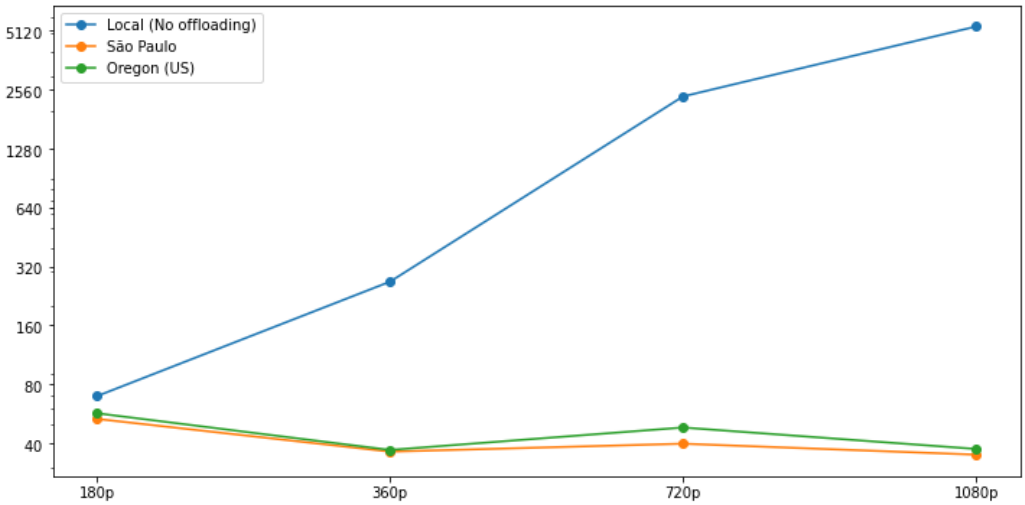


Figura 4: O tempo médio de processamento da CPU do dispositivo Android (milissegundos) na execução do filtro para cada cenário em relação ao tamanho de imagem (ambos em escala logarítmica), ao longo de 200 execuções do filtro.

simplesmente processar a imagem localmente é mais rápido que enviar os dados pela rede. Entretanto, conforme o tamanho da imagem cresce, é possível perceber que o *offloading* se torna mais vantajoso, já que o consumo de CPU cresce com o tamanho da imagem.

É importante notar que o consumo de CPU é semelhante nos dois cenários onde é feito *offloading*, já que a escolha do servidor pouco influencia no consumo de CPU do dispositivo.

5.3 Análise da latência de comunicação

Para analisarmos como a escolha do servidor impacta no tempo total de execução, mensuramos a latência de comunicação entre o dispositivo e o servidor – o tempo entre a requisição ser enviada pelo dispositivo e ser recebida pelo servidor. Os valores obtidos estão representados na Figura 5.

Através da Figura 5, observamos que a distância entre o servidor e o dispositivo é proporcional à latência. Assim, para melhor performance do aplicativo, o ideal seria a utilização de servidores em nuvem mais próximos do usuário ou de servidores em névoa.

Para analisarmos a contribuição da latência de comunicação para a latência total com *offloading*, criamos os gráficos de área representados na Figura 6, para São Paulo e Oregon, respectivamente.

Observamos que a latência total é majoritariamente originada da latência de comunicação para ambos os casos. Isso é ainda mais acentuado para o servidor em Oregon, que até para imagens grandes (de 2160p), a latência total foi causada predominantemente pela latência de rede.

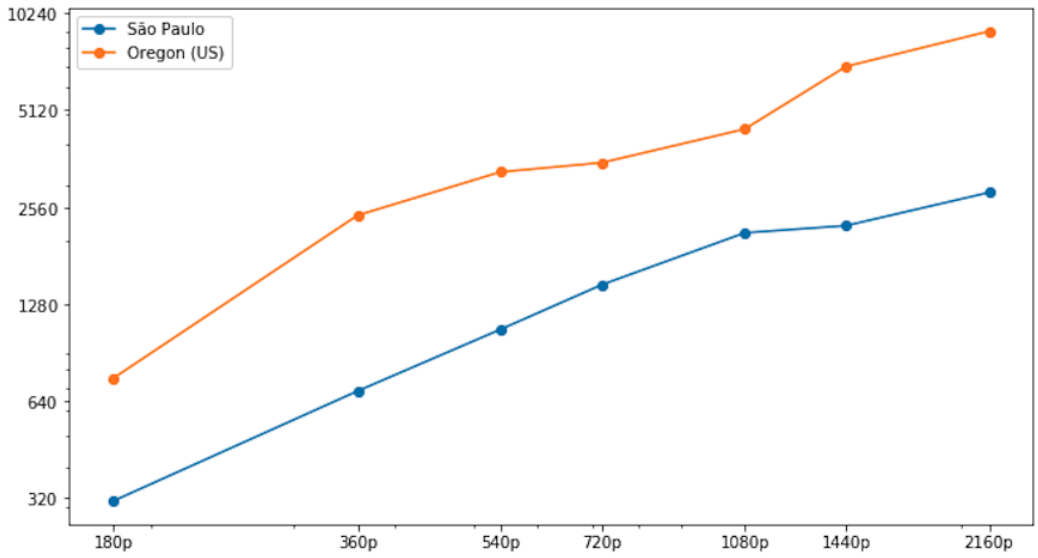


Figura 5: A latência média de comunicação (milissegundos) incorrida pelo *offloading* em relação ao tamanho de imagem (ambos em escala logarítmica), ao longo de 200 execuções do filtro.

5.4 Análise do consumo de bateria

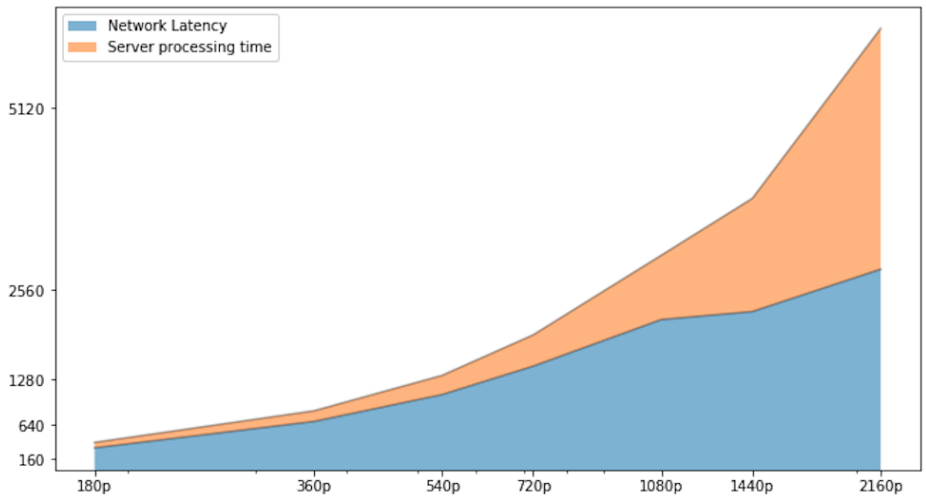
5.4.1 Experimentos com CPU

Em adição aos experimentos acima, realizamos um experimento sobre o consumo de bateria, uma métrica específica de dispositivos móveis. Como o experimento teve de ser realizado manualmente, usamos somente dois tamanhos diferentes de imagem, e os cenários local e com servidor em São Paulo. Obtivemos os resultados mostrados na Tabela 3.

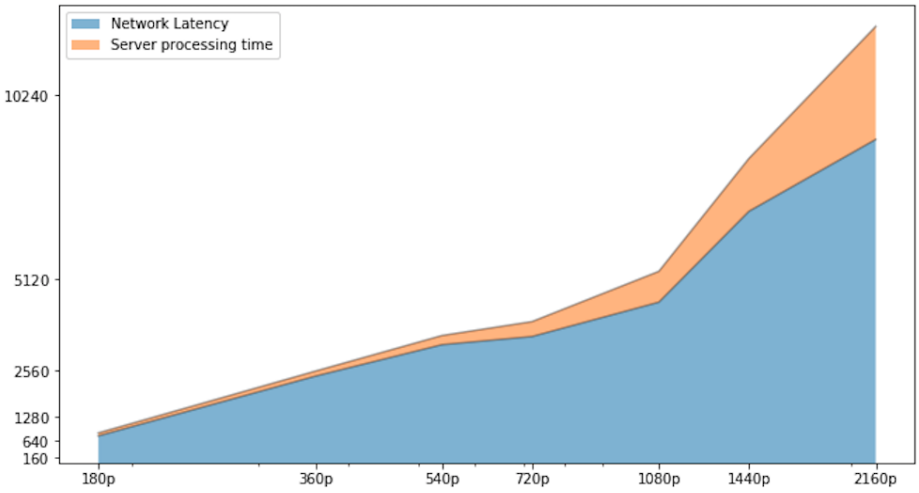
Tabela 3: Valores referentes ao consumo médio de bateria (em microampere hora – μAh) para cada cenário (coluna) e tamanho de imagem (linha), ao longo de 200 execuções do filtro.

	Local	São Paulo
128p	**2.439**	9.377
720p	101.209	**37.854**

Podemos notar que existe um *overhead* no consumo de bateria quando é feito *offloading*, causado pelo uso do componente de Wi-Fi do dispositivo. Conforme o tamanho da imagem cresce, existe uma melhora no consumo de bateria. Conforme discutido nos experimentos anteriores, isso está relacionado à economia no consumo de CPU que o *offloading* causa. Isso mostra que, no caso do filtro analisado, o aumento no custo de executar o algoritmo sobrepõe o *overhead* causado pelo componente de rede.



(a) Servidor em São Paulo.



(b) Servidor em Oregon.

Figura 6: A latência média por execução (milissegundos) fazendo-se *offloading* para cada servidor em relação ao tamanho da imagem em escala logarítmica, ao longo de 200 execuções do filtro.

5.4.2 Experimentos com GPU

Por causa da limitação das ferramentas do Android, não foi possível fazer a análise de bateria nos experimentos com GPU. Porém, podemos assumir que o consumo de bateria é proporcional aos valores obtidos para a CPU.

6 Conclusão

O objetivo desse trabalho foi analisar cenários em que o *offloading* pode ser aplicado em dispositivos móveis e averiguar se seu uso é vantajoso ou desvantajoso. Para isso, implementamos uma aplicação para dispositivos Android que aplica um filtro em uma imagem usando CPU e GPU. A partir dela, fizemos experimentos para diversos tamanhos de imagem em três cenários distintos: local, servidor em nuvem em São Paulo e em Oregon.

Com os dados obtidos a partir dos experimentos, vimos que para computações que usam a GPU do dispositivo, a execução local é mais vantajosa do que fazer o *offloading* para todos os tamanhos de imagem testados. No caso em que usamos a CPU, tanto na análise de tempo médio total, como na de consumo de CPU, o *offloading* é vantajoso apenas para os casos em que a computação é intensiva de dados (imagens maiores). Caso contrário, o *overhead* de transportar os dados pela rede e a latência de comunicação fazem com que o uso de *offloading* seja indesejável para a performance da aplicação.

Além disso, analisamos o consumo de bateria e vimos que nos casos em que a aplicação usa *offloading* há um *overhead* causado pelo uso do componente de Wi-Fi do dispositivo. Esse comportamento muda, no entanto, para imagens maiores, em que o consumo de bateria é menor ao se fazer *offloading* do que ao executar a computação localmente.

Portanto, mostramos algumas características importantes para se decidir uma política de *offloading* – conjunto de regras que decide se um certo processo deve ser *offloaded* ou não. Mais especificamente, mostramos como a variação dessas características, como tamanho da entrada a ser processada e distância cliente-servidor, impacta as métricas de execução como latência e consumo de bateria. Essa análise mostra algumas conclusões simples que podem ser utilizadas na construção de políticas de *offloading* mais eficientes para aplicações móveis.

Como trabalho futuro, outras métricas podem ser analisadas, como consumo de dados móveis – caso o celular esteja conectado a uma rede móvel fornecida pela operadora de celular – caso o filtro seja executado pelo componente gráfico do dispositivo. Outras características também podem ser exploradas, como a velocidade da conexão – que pode variar entre Wi-Fi e rede móvel, por exemplo –, e a quantidade de memória RAM disponível – que é, em particular, limitada em dispositivos móveis.

Referências

[1] S. E. Kim. Repositório Git do projeto. [S.I.] 2020. Disponível em: https://github.com/seongeunkim/offloading-analysis.

[2] K. Akherfi, M. Gerndt and H. Harroud, *Mobile cloud computing for computation offloading: Issues and challenges,* Applied Computing and Informatics (2018).

[3] M. A. Hassan, M. Xiao, Q. Wei and S. Chen, *Help Your Mobile Applications with Fog Computing,* 2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking - Workshops (SECON Workshops), Seattle, WA, 2015, pp. 1-6, doi: 10.1109/SECONW.2015.7328146.

[4] M. Satyanarayanan, P. Bahl, R. Caceres and N. Davies. *The case for VM-based cloudlets in mobile computing,* IEEE Pervasive Computing, volume 8(4) (2009).

[5] M. A. Hassan and S. Chen. *An investigation of different computing sources for mobile application outsourcing on the road,* Proc. of Mobilware (2011).

[6] M. A. Hassan, K. Bhattarai and S. Chen. *Virtually unifying personal storage for fast and pervasive data accesses,* Mobile Computing, Applications, and Services, pages 186–204. Springer (2013).

[7] L. Lin, X. Liao, H. Jin and P. Li. *Computation offloading toward edge computing,* Proceedings of the IEEE, vol. 107, no. 8, pp. 1584-1607, Aug. 2019, doi: 10.1109/JPROC.2019.2922285.

[8] Ra, Moo-Ryong & Sheth, Anmol & Mummert, Lily & Pillai, Padmanabhan & Wetherall, David & Govindan, Ramesh. (2011). Odessa: Enabling interactive perception applications on mobile devices. 43-56. 10.1145/1999995.2000000.

[9] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. 2007. *Simplifying cyber foraging for mobile devices*. In Proceedings of the 5th international conference on Mobile systems, applications and services (MobiSys '07). Association for Computing Machinery, New York, NY, USA, 272–285. DOI:https://doi.org/10.1145/1247660.1247692.

[10] Chun, Byung-Gon & Ihm, Sunghwan & Maniatis, Petros & Naik, Mayur & Patti, Ashwin. (2011). *CloneCloud: Elastic execution between mobile device and cloud*. EuroSys'11 - Proceedings of the EuroSys 2011 Conference. 301-314. 10.1145/1966445.1966473.

[11] Bentley, J. *Little Languages,* Communications of the ACM,29(8):711–721, 1986

[12] S. Yu, X. Wang and R. Langar, *Computation offloading for mobile edge computing: A deep learning approach,* 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), Montreal, QC, 2017, pp. 1-6, doi: 10.1109/PIMRC.2017.8292514.

[13] Picaso. http://code.google.com/p/picaso-eigenfaces/

[14] Droidslator. http://code.google.com/p/droidslator/

[15] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: making smartphones last longer with code offload. In Proceedings of the 8th international conference on Mobile systems, applications, and services (MobiSys '10). Association for Computing Machinery, New York, NY, USA, 49–62. DOI:https://doi.org/10.1145/1814433.1814441

[16] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. 2012. COMET: code offload by migrating execution transparently. In Proceedings

of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12). USENIX Association, USA, 93–106.

[17] SME Storage. http://smestorage.com/.

[18] What is cloud computing? Everything you need to know about the cloud explained. Visited on August 4th, 2020. https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-about-the-cloud/

[19] What is cloud computing? Everything you need to know now. Visited on August 4th, 2020. https://www.infoworld.com/article/2683784/what-is-cloud-computing.html

[20] What is fog computing? Connecting the cloud to things. Visited on August 4th, 2020. https://www.networkworld.com/article/3243111/what-is-fog-computing-connecting-the-cloud-to-things.html

[21] fog computing (fog networking, fogging). Visited on August 4th, 2020. https://internetofthingsagenda.techtarget.com/definition/fog-computing-fogging

[22] What is fog computing? Visited on August 4th. https://www.techradar.com/news/what-is-fog-computing

[23] Software-Defined Networking (SDN) Definition. https://www.opennetworking.org/sdn-definition/

[24] F. Khodadadi, A.V. Dastjerdi, R. Buyya, *Chapter 1 - Internet of Things: an overview,* Editor(s): Rajkumar Buyya, Amir Vahid Dastjerdi, Internet of Things, Morgan Kaufmann, 2016, Pages 3-27, ISBN 9780128053959, https://doi.org/10.1016/B978-0-12-805395-9.00001-0. (http://www.sciencedirect.com/science/article/pii/B9780128053959000010)