Análise e aplicação de técnicas de otimização de código

Arthur M. Passos, Guilherme S. Reis, Fábio Lubacheski, Jean M. Laine

Faculdade de Computação e Informática
Universidade Presbiteriana Mackenzie (UPM) - São Paulo, SP– Brasil

arthurmoreirap@gmail.com, gui.sreis25@gmail.com,
fabio.lubacheski@mackenzie.br, jean.laine@mackenzie.br

Abstract. *In this project, code analysis and optimization techniques are presented, in order to demonstrate how to improve the performance of a program, either by reducing processing time, processing load or memory usage. In the course of the article, the importance of these techniques is explained and how much they affect the execution of a software when applied in real-world projects, comparing and explaining the results, which were mostly positive.*

Resumo. *Neste projeto são apresentadas técnicas de análise e otimização de código, com a finalidade de demonstrar como melhorar o desempenho de um programa, seja reduzindo tempo de processamento, carga de processamento ou uso de memória. No decorrer do artigo é explicada a importância dessas técnicas e o quanto elas afetam a execução de um software quando aplicadas nos projetos de mundo real, comparando e explicando os resultados, que na maioria, foram positivos.*

1. Introdução

1.1 Contextualização do problema

Ao longo da história da computação, a quantidade de linguagens, *frameworks* e ferramentas que são desenvolvidas têm aumentado, totalizando mais de 270 linguagens em uso atualmente (TIOBE Programming Community Index Definition, 2022). Dessa forma, o desenvolvimento de software está cada vez mais fácil e rápido. Contudo, comumente não é suficiente apenas resolver o problema; em diversas aplicações emerge a necessidade de construir uma solução que seja rápida ou que aproveite ao máximo o *hardware* disponível, demandando, assim, técnicas de otimização de desempenho.

Para mais, mesmo nos casos em que esses critérios não são intrínsecos à resolução do problema, é possível abordar o desempenho da aplicação como uma “moeda de troca” (LEISERSON; SHUN, 2018, palestra 1), de forma que o programa sempre terá um limite mínimo de velocidade e eficiência. Consequentemente, caso haja margem acima do limite, o desempenho pode ser negociado por funcionalidades e abstrações, interfaces melhores etc. Por outro lado, em casos operando no limite ou até abaixo dele são impedidos da adição desses recursos sem prejudicar o resultado final. Assim, a preocupação com tais critérios pode não apenas melhorar o que já está planejado ou em uso, mas também expandir o horizonte do que é possível adicionar em cada solução.

Atualmente existem diversos sistemas e *hardwares* para propósitos diferentes. Os sistemas embarcados, por exemplo, usados em eletrodomésticos ou equipamentos com funcionalidades únicas, são pensados especificamente para o sistema em que vão controlar. Em contraponto, os sistemas com arquiteturas MIPS ou x64, usados nos computadores e celulares, que possuem componentes separados e são mais genéricos,

de variados tamanho e marcas. Para cada sistema, o programa criado deve ser completamente diferente do outro para maximizar o potencial da máquina.

Além dos *hardwares*, a própria escolha da linguagem é de extrema importância. Cada uma possui a sua particularidade, desde o tempo em que ela está no mercado, até a forma de serem lidas pela máquina. As linguagens recentes possuem certas limitações comparadas às que estão a mais tempo em utilização. Os tipos de execução delas também importam: as interpretadas, de alto nível, possuem um interpretador antes de ser lida pelo compilador, diferente das compiladas, baixo nível, que não precisam. A forma que cada uma vai ser lida pelo interpretador e/ou compilador é diferente, considerando principalmente que não é o mesmo programa para todas as linguagens. Assim, da mesma forma que é necessário conhecer o *hardware*, também é importante saber sobre o fluxo que a linguagem irá executar para construir um programa ótimo.

1.2 Objeto da pesquisa

Na generalidade dos contextos de estudo de Ciência da Computação em nível de graduação, ou até de introdução no mercado de trabalho, o tema de desempenho tende a ser deixado em segundo plano. Essa falta de propagação do tema, somado a necessidade de uma compreensão sobre o fluxo completo da execução de um programa no *hardware* faz com que o conhecimento em desempenho seja extremamente específico, ocasionando os desenvolvedores introduzirem más práticas nos sistemas.

Mesmo nos casos em que utilizam-se *frameworks* e ferramentas que propõe melhorar o desempenho, ainda faz-se preciso analisá-los com cautela para extrair o máximo possível dessas soluções. Frente ao exposto, surgem oportunidades, e até demandas, para aplicação de técnicas de otimização.

Dessa forma, a pergunta que esta pesquisa busca responder é: como podemos identificar e abordar um problema de desempenho?

Para trabalhar a questão são estudadas formas de analisar e otimizar *softwares* para aplicar em alguns projetos exemplo, localizando trechos que possam conter oportunidades de otimização e comparando os resultados após a utilização das técnicas.

1.3 Objetivo

O presente trabalho tem como propósito comparar o resultado de diferentes técnicas de otimização para abordar problemas de desempenho, mostrando quando elas devem ser usadas e os benefícios que vão trazer.

A fim de cumprir com tal propósito, o trabalho trata de reunir e revisar técnicas de otimização de código visando os atuantes dos sistemas, desde os compiladores até a linguagem utilizada. As implementações vão ser em exemplos de códigos contidos em aplicações de mundo real.

Por fim, são comparados os resultados de cada técnica, apresentando os impactos positivos e negativos no âmbito de desempenho e uma análise desses resultados.

1.4 Justificativa

Os *softwares* hoje em dia são criados para resolver um problema do momento. A preocupação com o desempenho aparece quando o sistema como um todo passa a ter problemas, considerando a limitação do *hardware* até a sobrecarga de uso da aplicação.

Com isso, o presente trabalho trata de sumarizar técnicas e ferramentas de otimização mostrando abordagens diferentes para um problema de desempenho, servindo como material de apoio para uma introdução a essa área. A validação é pela contribuição com projetos existentes, e não simulados.

2. Referencial Teórico

Analisar aplicações é uma das atividades chave no trabalho de otimização. Uma das formas é utilizar ferramentas que traçam o comportamento do *software*, buscando encontrar os pontos em que otimizações tendem a ter o melhor ganho em termos de desempenho. Tal método é chamado *profiling* (LEISERSON; SHUN, 2018, palest 10).

Existem diversas formas de fazer o *profiling* da aplicação. Contudo, essas técnicas tendem a enviesar a análise por questões específicas de implementação. Dessa forma, cada *profiler* (ferramentas que fazem o *profiling*) deve ser usado com cautela e em aplicações específicas, conhecendo as limitações dele.

A forma mais básica para mensurar o desempenho de aplicações é utilizar bibliotecas ou chamadas de sistema que contabilizam o tempo de execução da aplicação; seja contabilizando a execução inteira ou trechos específicos. Apesar desse método não considerar métricas de hardware, como o uso de memória e GPU, e apenas o tempo de execução, isso tende a impactar diretamente o usuário final. Assim, pode ser uma estratégia eficiente para medir o impacto final de otimizações em programas ou trechos com início e fim, em contraste de aplicações de execução contínua.

Outra abordagem de análise de *softwares*, desenvolvida mais recente com propósito de expor trechos de maior carga de uso da aplicação, é a utilização do método de visualização hierárquica *Flame Graphs* (GREGG, 2011). Tal método é utilizado para visualizar pilhas de execução de um código, apresentando o quanto as funções em que as *threads* do sistema gastaram durante a execução do programa, como mostrado na Figura 1. Assim, facilita a localização dos métodos que podem causar mais impacto no sistema quando otimizados.

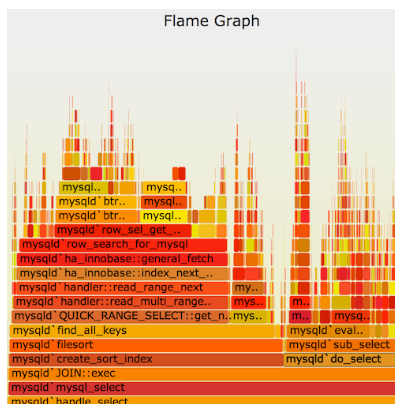

Figura 1: Pilha de execução de um banco de dados na forma de um *Flame Graph*. O gráfico é lido debaixo para cima, sendo a função mais inferior a mais externa, e

imediatamente acima dela as funções chamadas pela primeira função. A largura de cada chamada é relativa ao tempo de processamento utilizado.

Fonte: GREGG (2011)

A partir dos *profiling* é possível começar a fazer análises de código e inferências com o objetivo de buscar oportunidades de otimização de código. A sequência de palestras do MIT OpenCourseWare de tema *Performance Engineering of Software Systems* de Leiserson e Shun (2018) expõe conceitos base da área de otimização de código. Em especial na segunda palestra, na qual são apresentadas a definição de "trabalho" em desempenho e as *Bentley Rules* (Regras de Bentley) (BENTLEY, 1982). Tais regras são uma lista de técnicas de otimização genéricas. Contudo, várias dessas são antigas e são intrinsecamente dependentes da arquitetura da época. Assim, na palestra é proposto um novo conjunto de regras a partir do primeiro, dividido em 4 subgrupos: Estrutura de Dados, Laços (*Loops*), Lógica e Funções.

Além de otimizações a nível de código, também existem estruturas a nível de instrução no processador. Uma delas é o *Pipeline* (OSHANA, 2005). O intuito dessa estrutura é manter todas as partes do processador em uso, minimizando a ociosidade de qualquer região dele. Para isso ele é dividido em unidades e a partir do momento que uma instrução passa para a próxima unidade, a instrução seguinte na fila já começa a utilizar essa região mesmo que a instrução anterior não tenha acabado (Figura 2).

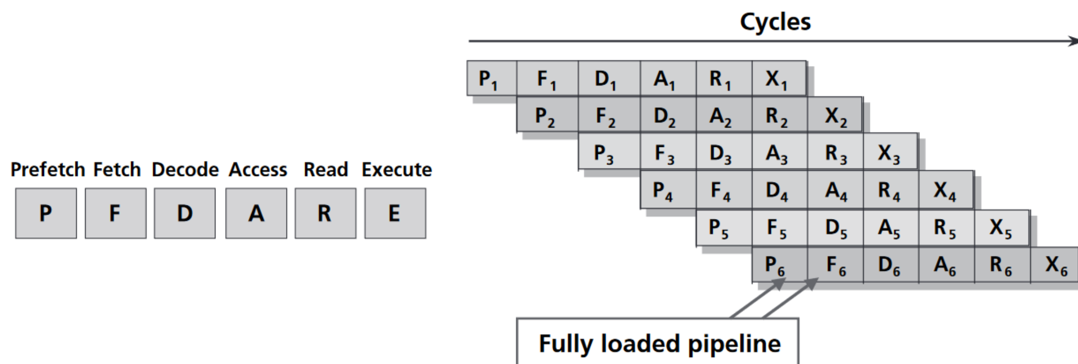

Figura 2. Exemplificação de alguns processos sendo executados utilizando o *Pipeline*.

Fonte: Oshana (2005)

Para que esse processo ocorra com fluidez é necessário que as instruções não dependam da informação de retorno de alguma instrução à sua frente na fila de execução. Caso tal dependência exista, o *Pipeline* será ignorado e será necessário esperar a finalização completa da instrução (*Stall*). A estratégia de programar com o foco no *Pipeline* é denominada programação *Pipeline Friendly*.

Um exemplo de uma estrutura *Pipeline Friendly* é o *Branch Prediction*, que aproveita ao máximo o *Pipeline* realizando previsões probabilísticas em ramificações do código, tais como "if" e "for". Dessa forma, o processador tenta estimar qual ramo será executado para evitar uma situação de *Stall*. A alteração do código para minimizar a

quantidade de ramos ou facilitar a predição do caminho deles é chamada *Branch optimization* (OSHANA, 2005).

O Singhal (2015) explica com cinco temas diferentes o ganho adquirido em desempenho com simples técnicas, mostrando com exemplo práticos e explicando quais benefícios são gerados, estes são: tempo de compilação, eliminação de repetição de código, código morto, realocação de código e redução de força.

Uma técnica específica para diminuir a quantidade de laços e ciclos em um código é usar o *Loop Fusion* ou *Jamming*. Para fazer a junção, os laços precisam estar usando as mesmas estruturas ou terem a mesma quantidade de passos e não podem ter uma dependência com o outro (Figura 3). A principal vantagem é diminuir a quantidade de ciclos que o compilador vai executar. No melhor caso pode reduzir pela metade (LEISERSON; SHUN, 2018, palestra 2).

```
for (int i = 0; i < n; ++i) {
  C[i] = (A[i] <= B[i]) ? A[i] : B[i];
}

for (int i = 0; i < n; ++i) {
  D[i] = (A[i] <= B[i]) ? B[i] : A[i];
}
```

```
for (int i = 0; i < n; ++i) {
  C[i] = (A[i] <= B[i]) ? A[i] : B[i];
  D[i] = (A[i] <= B[i]) ? B[i] : A[i];
}
```

Figura 3. Exemplo da aplicação de *Loop Fusion*. A direita o código original e a esquerda o código com a técnica aplicada.

Fonte: Leiserson e Shun (2018)

Outra técnica de otimização da classe de Laço é o *Hoisting* ou *Loop Invariant Code Motion* (OSHANA, 2005). Essa técnica tem como propósito remover cálculos redundantes de um laço extraindo operações que não dependem da iteração e permanecem constantes durante toda a execução dele (Figura 4). Essa técnica geralmente é aplicada pelo próprio compilador (Compiler Optimizations, 2012).

```
#include <math.h>

void scale(double *X, double *Y, int N) {
  for (int i = 0; i < N; i++) {
    Y[i] = X[i] * exp(sqrt(M_PI/2));
  }
}
```

```
#include <math.h>

void scale(double *X, double *Y, int N) {
  double factor = exp(sqrt(M_PI/2));
  for (int i = 0; i < N; i++) {
    Y[i] = X[i] * factor;
  }
}
```

Figura 4. Exemplo da aplicação de *Hoisting.* A cima o código original e abaixo o código com a técnica aplicada.

Fonte: Leiserson e Shun (2018)

Uma outra oportunidade de otimização da classe de Lógica surge de um erro comum na programação de cálculos extensos que utilizam um resultado que já tenha sido utilizado. Dessa forma a técnica se baseia em evitar o cálculo redundante extraindo a expressão e utilizando o resultado todas as vezes necessárias. Essa técnica é denominada *Common-sub expression elimination* (Figura 5) .

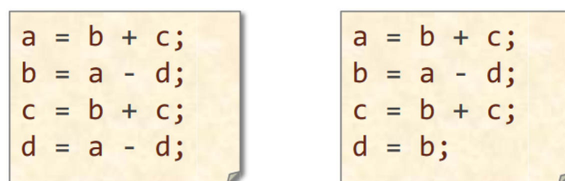

Figura 5. Exemplo da aplicação de *Common-sub expression elimination*. A direita o código original e a direita o código com a técnica aplicada.

Fonte: Leiserson e Shun (2018)

Existem outras técnicas complementares, além das apresentadas, mas por restrição de espaço no artigo não serão comentadas. Várias dessas são descritas em Oshana (2005).

3. Metodologia

Com o objetivo de realizar um processo coeso e consistente de otimização de código, o presente trabalho é constituído de cinco etapas principais que serão detalhadas ao longo desta seção.

- Pesquisa do referencial,
- Escolha dos programas para estudo,
- *Profiling* e análises,
- Aplicação das técnicas,
- Análise dos resultados.

3.1 Pesquisa do referencial

Nesse primeiro passo foi realizada a pesquisa de referências para técnicas de análise e otimização de código. Foi feita uma busca geral sobre como abordar problemas de desempenho: quais métricas é possível utilizar para cada caso, quais ferramentas e técnicas aplicar, a forma correta de utilizá-las e como medir o ganho de desempenho após otimização.

3.2 Escolha dos programas para estudo

Nesta segunda etapa, foram selecionados com potencial ou necessidade de otimizações de desempenho para, assim, aplicar as técnicas abordadas. Esses projetos são:

- Stars And Exoplanets (2022): o projeto tem o propósito de "calcular a curva de luz de uma estrela com possíveis exoplanetas em sua órbita através de programação Python híbrida para programação C usando Pipeline em tempo de execução."
- Catch Fly - The Escape: um jogo *hyper casual* criado para iOS e tvOS na linguagem Swift, usando o framework nativo SpriteKit.

3.3 *Profiling* e análises

Com os códigos de base selecionados, são feitos o *profiling* dessas aplicações, utilizando as ferramentas de acordo com as tecnologias, e uma análise de código fonte, buscando fazer uma breve triagem, localizando os códigos que mais oferecem oportunidades de otimização. Ademais, esse processo auxilia conhecer os projetos, a regra de negócio e qual métrica é prioridade em dado contexto.

3.4 Aplicação das técnicas

Após analisar as aplicações, algumas técnicas de otimização, explicadas no referencial, foram aplicadas e o resultado avaliado. Essa etapa é o foco principal do projeto, uma vez que, além do trabalho de aplicar as técnicas, é necessário realizar novos *profiling* e análises e checar resultados parciais.

3.5 Análise dos resultados

Por fim, esta etapa consiste em documentar o processo, buscando detalhar os resultados obtidos após a aplicação das técnicas e explicando o processo que foi seguido.

4. Aplicações

Essa seção tem como finalidade apresentar a aplicação dos procedimentos metodológicos descritos na seção anterior nos projetos selecionados.

4.1 Stars And Exoplanets

Essa sub-seção trata do processo aplicado à aplicação Stars And Exoplanets.

4.1.1 *Profiling* e análises

Como a aplicação possui uma interface simples, primeiramente foi alterado o código para minimizar a necessidade de interação com o usuário, injetando valores na execução.

Em seguida, foi feita uma execução do código gerando um arquivo de *profiling* com o cProfile do Python, capturando o tempo de execução de cada chamada de função. Durante o *profiling* também foi utilizado um parâmetro que contabiliza apenas o tempo que as funções ocuparam espaço no processador, desconsiderando tempo na fila na fila de espera. Esse arquivo serve como interface para diversas outras ferramentas de análise. Nesse caso ele foi utilizado para gerar um *Flame Graph* (Figura 6).

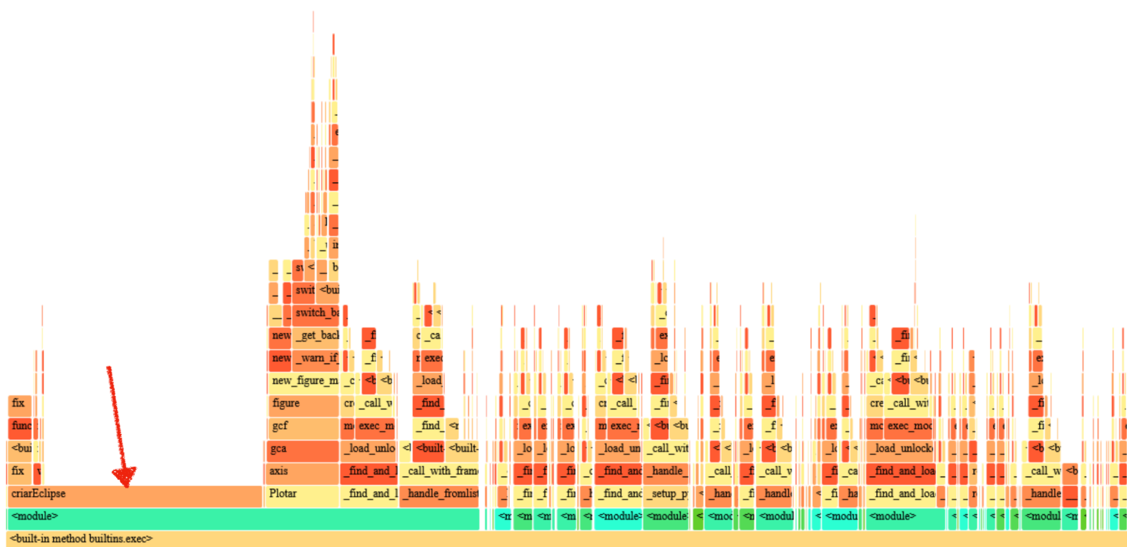

Figura 6. *Flame Graph* produzido pela execução do Stars And Exoplanets. Na parte inferior esquerda da imagem gerada é visível que a função “criaEclipse” ocupa uma parcela considerável do programa, dados os métodos não internos da aplicação.

A partir desse *gráfico* é possível identificar que um dos métodos que mais se destaca é o “criarEclipse”. Assim, encontrar uma otimização para esse trecho pode reduzir consideravelmente a carga de execução do programa; e consequentemente o tempo de execução.

Com uma breve análise de código, foi possível identificar que o método em questão possui um laço principal no qual faz uma chamada a uma DLL escrita em C (Figura 7).

```
for i in range(0,len(rangeloop)):
    x0 = xplan[i]
    y0 = yplan[i]

    ### adicionando luas ###
    xm = x0-self.xxm[i]
    ym = y0-self.yym[i]

    self.curvaLuz[rangeloop[i]]=my_func.curvaLuzLua(x0,y0,xm,ym,self.Rmoon,self.tamanhoMatriz,raioPlanetaPixel,em,kk2,maxCurvaLuz)

    if(plota and self.curvaLuz[rangeloop[i]] != 1 and numAux<200):
        plan = np.zeros(tamanhoMatriz*tamanhoMatriz)+1.
        ii = np.where(((kk/tamanhoMatriz-y0)**2+(kk-tamanhoMatriz*np.fix(kk/tamanhoMatriz)-x0)**2 <= raioPlanetaPixel**2))
        ll = np.where((kk/tamanhoMatriz-ym)**2+(kk-tamanhoMatriz*np.fix(kk/tamanhoMatriz)-xm)**2 <= self.Rmoon**2)
        plan[ii]=0.
        plan[ll]=0.
        plan = plan.reshape(self.tamanhoMatriz, self.tamanhoMatriz)     # posicao adicionada na matriz
        plt.axis([0,self.Nx,0,self.Ny])
        im = ax1.imshow(self.estrelaManchada*plan,cmap="hot", animated = True)
        ims.append([im])                                                # armazena na animação os pontos do grafico (em imagem)
        numAux+=1
    plota = not(plota)
```

Figura 7. Laço principal da função “criaEclipse”. Na linha 291 possui a chamada para a DLL de nome “curvaLuzLua”.

Dadas questões encontradas, a fim de complementar a métrica de tempo de CPU exposta no *Flame Graph*, é feita uma contagem do tempo de execução da função

"criaEclipse". Separadamente, foi contabilizado o tempo do laço principal e a chamada da DLL. Os tempos médios de 100 execuções estão expostos na Tabela 1.

	Tempo de execução	
Trecho monitorado	**Em segundos**	**Proporção**
Código completo	4,070	100%
Método "criaEclipse"	3,960	97,3%
Laço principal da "criaEclipse"	2,997	73,6%
Chamada da DLL acumulado	0,651	16,1%

Tabela 1: Tempos médios de 100 execuções de trechos selecionados. Vale notar que o tempo da chamada à DLL dentro do laço foi agregado por soma, dado que esse trecho está dentro de um laço e o objetivo é encontrar o tempo total de execução.

Com tais médias é possível identificar que a maior parte da execução se passa dentro do laço do "criaEclipse". Contudo, não é na chamada do código em C, e sim no trecho escrito em Python. Dessa forma, esse laço é um trecho que apresenta uma chance relevante de impactar a aplicação significativamente em caso de otimizações.

4.1.2 Aplicação das técnicas

A fim de padronizar as comparações, em todos os testes (incluindo a análise inicial) foram executadas 100 vezes o programa com os mesmos parâmetros:

- "Não" para calcular semi eixo orbital do planeta através da 3ª Lei de Kepler;
- 0,028 para Semi eixo (em UA);
- 1 minuto para intervalo de tempo.

4.1.2.2 Segunda otimização

Posto tais análises, avaliando o código do laço é possível identificar várias operações duplicadas e contas constantes, isto é, aquelas que não mudam de uma iteração do laço para outra (Figura 8).

```python
for i in range(0,len(rangeloop)):
    x0 = xplan[i]
    y0 = yplan[i]

    ### adicionando luas ###
    xm = x0-self.xxm[i]
    ym = y0-self.yym[i]

    self.curvaLuz[rangeloop[i]]=my_func.curvaLuzLua(x0,y0,xm,ym,self.Rmoon,self.tamanhoMatriz,raioPlanetaPixel,em,kk2,maxCurvaLuz)

    if(plota and self.curvaLuz[rangeloop[i]] != 1 and numAux<200):
        plan = np.zeros(tamanhoMatriz*tamanhoMatriz)+1.
        ii = np.where(((kk/tamanhoMatriz-y0)**2+(kk-tamanhoMatriz*np.fix(kk/tamanhoMatriz)-x0)**2 <= raioPlanetaPixel**2))
        ll = np.where((kk/tamanhoMatriz-ym)**2+(kk-tamanhoMatriz*np.fix(kk/tamanhoMatriz)-xm)**2 <= self.Rmoon**2)
        plan[ii]=0.
        plan[ll]=0.
        plan = plan.reshape(self.tamanhoMatriz, self.tamanhoMatriz)     # posicao adicionada na matriz
        plt.axis([0,self.Nx,0,self.Ny])
        im = ax1.imshow(self.estrelaManchada*plan,cmap="hot", animated = True)
        ims.append([im])                                                # armazena na animação os pontos do grafico (em imagem)
        numAux+=1
    plota = not(plota)
```

Figura 8. Código com fonte com cálculos que não variam no laço sublinhado.

Dessa forma, é possível aplicar a técnica de *Common subexpression elimination* eliminando os cálculos duplicados. Além disso, por essas operações estarem dentro de um laço e não dependerem das iterações, também é possível extrair esses cálculos constantes do laço aplicando o *Hoisting* (Figura 9).

```python
kkTamanhoMatriz = kk/tamanhoMatriz
npFixKkTamanhoMatriz = kk-tamanhoMatriz*np.fix(kkTamanhoMatriz)
rMoonSqr = self.Rmoon**2
raioPlanetaPixerSqr = raioPlanetaPixel**2
tamanhoMatrizSqr = tamanhoMatriz**2

for i in range(0,len(rangeloop)):
    x0 = xplan[i]
    y0 = yplan[i]

    ### adicionando luas ###
    xm = x0-self.xxm[i]
    ym = y0-self.yym[i]

    self.curvaLuz[rangeloop[i]]=my_func.curvaLuzLua(x0,y0,xm,ym,self.Rmoon,self.tamanhoMatriz,raioPlanetaPixel,em,kk2,maxCurvaLuz)

    if(plota and self.curvaLuz[rangeloop[i]] != 1 and numAux<200):
        plan = np.zeros(tamanhoMatrizSqr)+1.
        ii = np.where(((kkTamanhoMatriz-y0)**2 + (npFixKkTamanhoMatriz-x0)**2 <= raioPlanetaPixerSqr))
        ll = np.where((kkTamanhoMatriz-ym)**2 + (npFixKkTamanhoMatriz-xm)**2 <= rMoonSqr)
        plan[ii]=0.
        plan[ll]=0.
        plan = plan.reshape(self.tamanhoMatriz, self.tamanhoMatriz)     # posicao adicionada na matriz
        plt.axis([0,self.Nx,0,self.Ny])
        im = ax1.imshow(self.estrelaManchada*plan,cmap="hot", animated = True)
        ims.append([im])                                                # armazena na animação os pontos do grafico (em imagem)
        numAux+=1
    plota = not(plota)
```

Figura 9. Código com fonte com variáveis extraídas acima do laço.

A partir de tais modificações o código foi uma nova análise.

	Tempo de execução (em segundos)	
Códigos	**Sem alteração**	**Otim. 01**
Código completo	4,070	2,902
Método “criaEclipse”	3,960	2,791
Laço principal da “criaEclipse”	2,997	1,828
Chamada da DLL acumulado	0,651	0,653

Tabela 2. Tempos médios de 100 execuções de trechos selecionados, após aplicação do *Common subexpression elimination* e *Hoisting*.

Com a nova média do tempo de execução é possível perceber que houve uma diminuição no tempo de execução, obtendo um ganho de aproximadamente 28,6% de tempo de execução.

Apesar de ser uma técnica básica que pode ser aplicada pelo compilador, o que causou o impacto positivo nesse caso se deve principalmente ao fato de Python ser uma linguagem interpretada, limitando as possibilidades de otimização automática.

4.1.2.2 Segunda otimização

A segunda otimização foi verificar o código fonte da DLL em C e observou-se que os cálculos extraídos na Figura 9 são também são realizados (Figura 10).

```
double curvaLuzLua(
  double x0, double y0, double xm, double ym, double rMoon, int tamanhoMatriz, int raioPlanetaPixel,
  double* estrelaManchada, double* kk, double maxCurvaLuz)
{
    double valor = 0;
    int i;

#pragma omp parallel for reduction(+:valor)
    for(i=0; i<tamanhoMatriz*tamanhoMatriz; i++) {
        if( (pow((kk[i]/tamanhoMatriz-y0), 2) +
            pow((kk[i]-tamanhoMatriz * floor(kk[i]/tamanhoMatriz)-x0), 2) > pow(raioPlanetaPixel,2)) &&
            (pow((kk[i]/tamanhoMatriz-ym), 2) + pow((kk[i]-tamanhoMatriz * floor(kk[i]/tamanhoMatriz)-xm), 2) > pow(rMoon, 2))
            ) {
            valor += estrelaManchada[i];
        }
    }

    valor = valor/maxCurvaLuz;
    return valor;
}
```

Figura 10. Código com fonte da DLL em C com os cálculos redundantes indicados.

Assim, por mais que a maioria dos compiladores de C sejam excelentes em realizar *Common subexpression elimination* automaticamente (Compiler Optimizations. 2012), como esse cálculo é já realizado no trecho em Python, há um potencial de evitar essa redundância aplicando a técnica *hoisting* novamente (Figura 11).

```c
double curvaLuzLua(
    double x0, double y0, double xm, double ym, double rMoonSqr, int tamanhoMatriz,
    int raioPlanetaPixelSqr, double *estrelaManchada, double *kk, double maxCurvaLuz)
{
    double valor = 0;
    int i;

#pragma omp parallel for reduction(+:valor)
    for(i=0; i<tamanhoMatriz*tamanhoMatriz; i++) {
        if((pow((kk[i]/tamanhoMatriz-y0),2) +
            pow((kk[i]-tamanhoMatriz*floor(kk[i]/tamanhoMatriz)-x0),2) > raioPlanetaPixelSqr) &&
            (pow((kk[i]/tamanhoMatriz-ym),2) + pow((kk[i]-tamanhoMatriz*floor(kk[i]/tamanhoMatriz)-xm),2) > rMoonSqr)
            ){
            valor += estrelaManchada[i];
        }
    }

    valor = valor/maxCurvaLuz;
    return valor;
}
```

Figura 11. Código com fonte da DLL em C após a extração dos cálculos, sendo passados como parâmetros.

A partir disso, foi feita uma nova análise (Tabela 3).

	Tempo de execução (em segundos)		
Códigos	**Sem alteração**	**Otim. 01**	**Otim. 02**
Código completo	4,070	2,902	2,890
Método “criaEclipse”	3,960	2,791	2,779
Laço principal da “criaEclipse”	2,997	1,828	1,816
Chamada da DLL acumulado	0,651	0,653	0,642

Tabela 3. Tempos médios de 100 execuções de trechos selecionados após a extração das variáveis na DLL.

Dado que cálculos na DLL tendem a apresentar um custo reduzido em relação a Python, a otimização apresentou um menor que a primeira aplicação da técnica, aproximadamente 0,4% , 29% em relação ao código inicial. Para mais, apesar do cálculo ser definido no interior de um laço no Python e dentro de outro na linguagem C, o compilador tende a aplicar essa técnica. Assim, o cálculo é realizado na DLL apenas uma vez por chamada.

4.1.2.2 Terceira otimização

Ainda no contexto do trecho em C, é possível observar a oportunidade teórica para aplicação da técnica de *Branch Optimization* (Figura 12).

```
double curvaLuzLua(
    double x0, double y0, double xm, double ym, double rMoonSqr, int tamanhoMatriz,
    int raioPlanetaPixelSqr, double *estrelaManchada, double *kk, double maxCurvaLuz)
{
    double valor = 0;
    int i;

#pragma omp parallel for reduction(+:valor)
    for(i=0; i<tamanhoMatriz*tamanhoMatriz; i++) {
        if((pow((kk[i]/tamanhoMatriz-y0),2) +
            pow((kk[i]-tamanhoMatriz*floor(kk[i]/tamanhoMatriz)-x0),2) > raioPlanetaPixelSqr) &&
            (pow((kk[i]/tamanhoMatriz-ym),2) + pow((kk[i]-tamanhoMatriz*floor(kk[i]/tamanhoMatriz)-xm),2) > rMoonSqr)
            ){
            valor += estrelaManchada[i];
        }
    }

    valor = valor/maxCurvaLuz;
    return valor;
}
```

Figura 12. Código sinalizando a condição e a agregação extraída.

Essa aplicação deve-se quando há uma agregação no interior de um bloco condicional. Assim, ela é extraída e adicionada à uma multiplicação (Figura 13), uma vez que em C, a condição do bloco retorna um valor inteiro (1 ou 0). Dessa forma, a agregação só será realizada quando a condição é verdadeira (1), já que no caso contrário, a multiplicação por zero resulta no mesmo.

```
#pragma omp parallel for reduction(+:valor) {
    for(i=0;i<tamanhoMatriz*tamanhoMatriz;i++) {
        valor += estrelaManchada[i] * ((pow((kk[i]/tamanhoMatriz-y0),2) +
          pow((kk[i]-tamanhoMatriz*floor(kk[i]/tamanhoMatriz)-x0),2) > raioPlanetaPixelSqr) &&
          (pow((kk[i]/tamanhoMatriz-ym),2) + pow((kk[i]-tamanhoMatriz*floor(kk[i]/tamanhoMatriz)-xm),2) > rMoonSqr));
    }

    valor = valor/maxCurvaLuz;
    return valor;
}
```

Figura 13. Trecho com a agregação extraída e substituindo bloco condicional por multiplicação.

Por essa técnica remover uma ramo de possibilidade de fluxo de código, facilita a execução no *pipeline*. A partir da nova execução podemos ver a análise na Tabela 4.

	Tempo de execução (em segundos)			
Códigos	**Sem alteração**	**Otim. 01**	**Otim. 02**	**Otim. 03**
Código completo	4,070	2,902	2,890	7,788
Método “criaEclipse”	3,960	2,791	2,779	7,675
Laço principal da “criaEclipse”	2,997	1,828	1,816	6,714

Chamada da DLL acumulado	0,651	0,653	0,642	0,758

Tabela 4. Tempos médios de 100 execuções de após aplicar a *Branch optimization*.

Com os novos resultados é possível perceber que a técnica não reduziu o tempo, e sim aumentou consideravelmente em 169,5%. Esse é um exemplo de que as técnicas aplicadas sem cuidado podem ser inúteis ou até prejudiciais ao fluxo do código. Nesse caso, a alteração do código impediu otimizações do próprio compilador, gerando esse aumento no tempo total.

4.2 Catch Fly - The Escape

Essa sub-seção trata do processo aplicado ao jogo Catch Fly - The Escape.

4.2.1 *Profiling* e análises

Como o aplicativo Catch Fly foi criado usando SpriteKit, o *framework* nativo da Apple que vem junto com a linguagem Swift, no primeiro momento foi estudado a forma que ele funciona. Entendendo que o framework usa o *design pattern State*, que se baseia em máquina de estado finito, foi identificado as funções principais que são chamadas quando há uma mudança de estado (Figura 14).

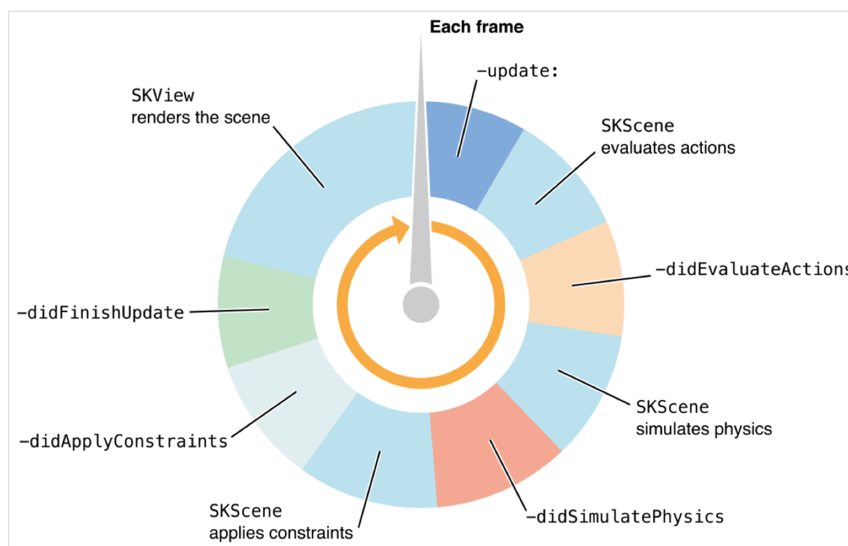

Figura 14. Ciclo de execução de um frame no *Sprite Kit.*

Reconhecendo as funções do ciclo do SpriteKit, um dos principais métodos usados para a execução do código é o "update". Como mostrado na Figura 14, essa função é chamada a cada ciclo do frame. A quantidade de frames por segundo (FPS) máxima é de 60, sendo que, quanto mais próximo o FPS da aplicação a esse número, melhor é o seu desempenho, aumentando a sensação de fluidez durante o jogo.

Ao verificar o código, o objetivo inicial foi identificar o trecho responsável que acontece durante o jogo e encontrar a função *update* (Figura 15) para poder entender o que acontece a cada frame.

```swift
override func update(_ currentTime: TimeInterval) {
    self.currentTime = currentTime
    let outOfTheScreenNodes = children.filter { node in
        gameLogic.passedObstacles(node: node)
    }

    for node in outOfTheScreenNodes {
        node.physicsBody = nil
    }

    moveObstacle()
    gameLogic.moveBackground()
    removeChildren(in: outOfTheScreenNodes)
    gameLogic.update(currentTime: currentTime)
}
```

Figura 15. Função *update* do código referente ao jogo.

Analisando a função (Figura 15), foi possível identificar as ações principais que ela executa (Figura 16), sendo elas: a remoção dos elementos que não estão visíveis na tela, movimentação dos elementos na tela e cálculo dos tempos que cada componente possui.

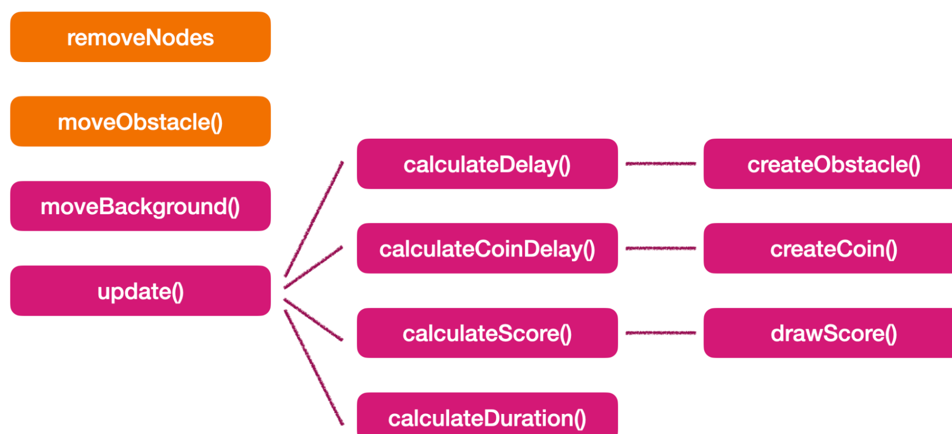

Figura 16. Ações e métodos que acontecem quando a função *update* é executada.

A partir dessas observações foi feita uma análise para entender como que a aplicação estava se comportando e o quanto estava consumindo de memória e de GPU.

Código	FPS (SpriteKit)	FPS (HUD)	Memória (mB)	GPU (%)	Nodes
Jogo	58,8708	59,54412	141,95	1,129	10,4

Tabela 5. Dados de uma partida sem alteração.

4.2.2 Aplicação das técnicas

Para manter o mais próximo de uma situação real, todas as análises foram feitas cinco vezes para cada modificação, incluindo a análise inicial. Foi usado um dispositivo

com as mesmas configurações. Em todas as execuções foram realizados o seguinte fluxo: remoção da versão anterior, instalação de uma nova (para garantir que nenhum *cache* fique salvo), iniciar o aplicativo como se fosse a primeira vez, fazendo o tutorial e em seguida executar a partida. Os valores obtidos para as análises foram dos 50 primeiros segundos de jogo.

No total foram usados cinco valores para as métricas: cálculo do FPS feito pelo *framework* (SpriteKit), cálculo do FPS feito pelo HUD (*heads-up display* - tela de alerta) do iPhone, consumo de memória, uso da GPU e a quantidade de *nodes* na tela.

4.2.2.1 Primeira otimização

A primeira modificação feita foi mudando o momento em que a ação de remover os *nodes* (componentes do jogo do *framework*) que não estão mais visíveis na tela é executada. Essa ação não é necessária fazer 60 vezes por segundo, já que os componentes do jogo ficam na tela por pelo menos um segundo.

A nova chamada dessa ação foi feita quando um novo objeto é criado (Figura 17). Como o cálculo de quando um novo obstáculo precisa ser gerado já está sendo realizado, ao adicionar um novo *node* na tela, aproveita o mesmo momento para remover aqueles que não estão mais visíveis.

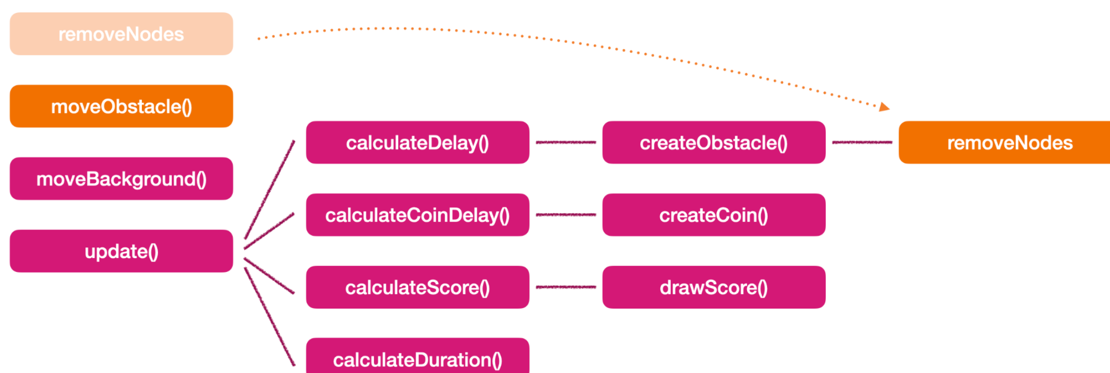

Figura 17. Novo momento em que a ação de remover os *nodes* é executada.

A partir dessa simples modificação de chamada (Figura 17), no período da partida, a função passou a ser executada em uma média de 30 vezes, o que antes estava eram 3000 vezes (60 frames por segundo por 50 segundos). Assim, foi feita uma análise como mostra na Tabela 6.

Código	FPS (SpriteKit)	FPS (HUD)	Memória (mB)	GPU (%)	Nodes
Jogo	58,8708	59,54	141,9	1,129	10,4
Otim. 01	58,8712	59,58	141,9	1,136	11

Tabela 6. Análises das partidas sem nenhuma otimização e com a primeira otimização

Analisando os dados da é possível ver um aumento na média do FPS. Como a remoção não está sendo feita sempre, a quantidade de *nodes* na tela aumentou, o que resultou no aumento do uso da GPU. Apesar disso, o consumo de memória não teve alteração.

4.2.2.2 Segunda otimização

A segunda otimização foi feita na função *moveObstacle*, responsável por mover os objetos (obstáculos e moedas) na tela. A chamada dela é necessária em todos os frames que acontecem, já que a posição dela é atualizada em cada um.

Analisando a *moveObstacle* (Figura 18) foi identificado que há dois laços: uma filtragem a partir de uma lista (que contém todos os componentes da tela) e o acesso de cada item dessa lista filtrada para então mover os elementos.

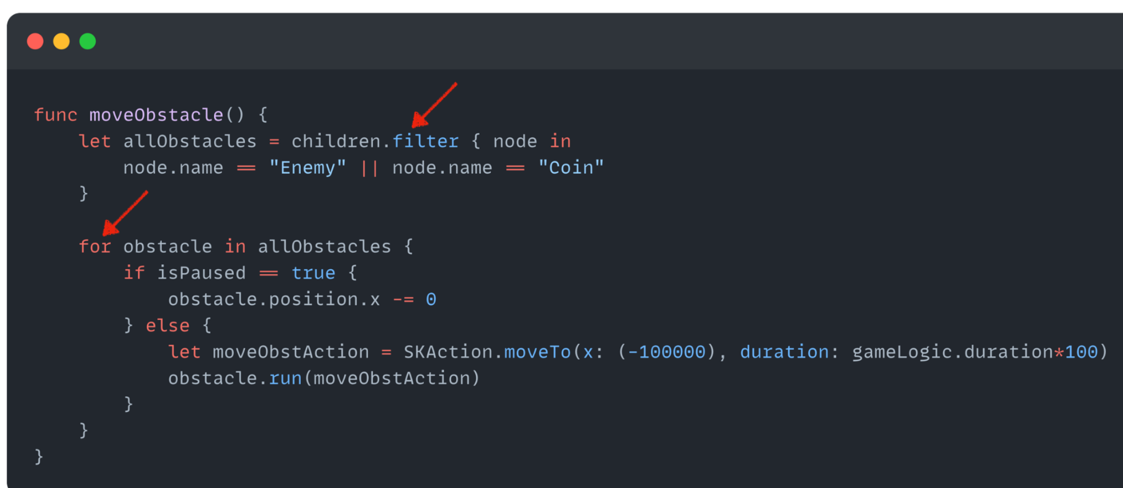

Figura 18. Função *moveObstacle* sem nenhuma alteração com indicação dos laços.

Como os dois laços acabam usando o mesmo vetor, é possível realizar o *loop fusion*. Além da fusão, outro fator que pode ser modificado é a condição do filtro. O atributo *name* é do tipo *optional*. Na linguagem Swift, os tipos *optionals* são variáveis que podem ser nulas. Além disso, os valores possíveis para esse atributo são: *nil* (nulo), *fly* (a mosca que o usuário controla), *enemy* (os obstáculos) e *coins* (as moedas).

A partir disso, é interessante verificar se o atributo *name* existe primeiro antes de comparar os valores. Certificando que existe, faria a verificação de qual componente está sendo acessado, que no caso os que são diferentes de mosca (Figura 19).

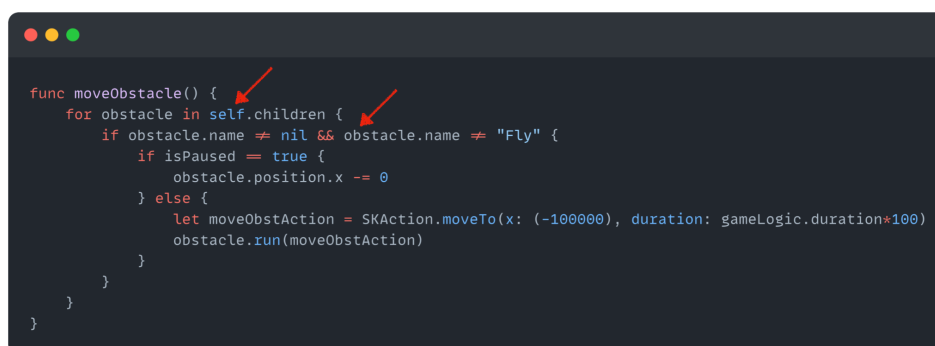

Figura 19. Função *moveObstacle* com *loop fusion* e alteração na condição de filtro.

Por fim, a técnica que é possível ser aplicada é a *hoisting*. Ela vai ser aplicada em dois casos. O primeiro caso é na condicional estática. A verificação *isPaused* == *true* não é alterada dentro do laço, sendo assim, uma condição que nunca vai mudar dentro do escopo. A aplicação do *Hoisting* em condicionais gera uma fissão do laço.

Além disso, essa condição é menos executada, já que, para estar de acordo com ela, o jogo precisa estar no modo *pause*. Invertendo ela, o primeiro trecho passa a ser o mais importante, considerando a execução do *if* e na leitura do código em uma possível manutenção futura.

```swift
func moveObstacle() {
    if !isPaused == true {
        for obstacle in self.children {
            if obstacle.name != nil && obstacle.name != "Fly" {
                let moveObstAction = SKAction.moveTo(x: (-100000), duration: gameLogic.duration*100)
                obstacle.run(moveObstAction)
            }
        }
    } else {
        for obstacle in self.children {
            if obstacle.name != nil && obstacle.name != "Fly" {
                obstacle.position.x -= 0
            }
        }
    }
}
```

Figura 20. *moveObstacle* com *loop fusion*, alteração na condição e *hoisting*.

O segundo caso é na variável estática. A conta passada por parâmetro na função *moveTo* é um cálculo que não muda dentro do escopo, sempre vai ter o mesmo resultado. Aplicando o *hoisting* removemos essa multiplicação para fora do laço.

```swift
func moveObstacle() {
    if !isPaused == true {
        let duration = gameLogic.duration*100

        for obstacle in self.children {
            if obstacle.name != nil && obstacle.name != "Fly" {
                obstacle.run(
                    SKAction.moveTo(x: (-100000), duration: duration)
                )
            }
        }
    } else {
        for obstacle in self.children {
            if obstacle.name != nil && obstacle.name != "Fly" {
                obstacle.position.x -= 0
            }
        }
    }
}
```

Figura 21. Função *moveObstacle* com *loop fusion,* alteração na condição de filtro, *Hoisting* na condição e *Hoisting* na variável.

Após a aplicação dessas técnicas, com a nova função *moveObstacle* (Figura 21) foi feita uma nova análise.

Código	FPS (SpriteKit)	FPS (HUD)	Memória (mB)	GPU (%)	Nodes
Jogo	58,8708	59,54	141,9	1,129	10,4
Otim 01	58,8712	59,58	141,9	1,136	11
Otim 02	59,8756	59,60	142	1,125	10,9

Tabela 7. Análises das partidas com todas as otimizações.

Como a função *moveObstacle* é executada 3000 vezes, com a aplicação das técnicas percebemos que a GPU trabalha menos comparado aos dois casos anteriores e ainda sim apresentou o maior FPS. Apesar disso, o consumo de memória teve um aumento de apenas 0,1mb.

5. Considerações finais

Os dois softwares selecionados para análise e aplicação das técnicas foram criados de formas diferentes. Por conta desse contraste, as abordagens para análises também foram divergentes.

O projeto Stars And Exoplanets foi desenvolvido sem uma arquitetura base e utilizando majoritariamente Python, além de alguns módulos em C. Ademais, essa aplicação é um utilitário de fluxo sequencial, que inicia e termina em cada utilização realizada pelo usuário final. Dessa forma, foi utilizado o *frame graph* para encontrar o trecho de computação principal e então focou-se na métrica que mais impacta o usuário, que é o tempo de execução.

Com a aplicação das técnicas obteve-se um ganho de 28,6%. Esse ganho se dá principalmente pelo fato da linguagem ser interpretada e muitas das técnicas não serem implementadas no interpretador do Python.

A linguagem C é uma das mais antigas e usadas na atualidade. Por conta disso, ela vem sendo melhorada a mais tempo. Muitas dessas técnicas que foram aplicadas já são executadas no compilador e de forma mais prática ao sistema.

No caso do projeto Catch Fly, a sua criação foi baseada em um *framework*. A linguagem Swift que foi usada também é compilada, porém a origem dela é mais recente comparada a linguagens clássicas como C, tendo as otimizações do compilador diferentes e inferiores. Por usar o *SpriteKit*, a análise do código fonte teve como objetivo principal entender como funciona a sua arquitetura.

Entendendo o seu funcionamento, uma das otimizações implementadas que teve um ganho de desempenho foi mover a chamada de uma função, sem a aplicação de uma

técnica específica. Além disso, também foram aplicadas algumas técnicas em um dos métodos que era executado em todos os ciclos.

Com essas otimizações houve um aumento na média do FPS do jogo em 0,06 com a GPU trabalhando menos. Apesar dessas vantagens teve um aumento da memória de 0,1 mb. Mesmo sendo pequenas, considerando que a linguagem é compilada e o jogo é simples, não exigindo tanto do dispositivo, foi um ganho observado. Além do desempenho, a legibilidade do código ficou melhor, facilitando a leitura.

Cada projeto tem a sua construção singular e métricas específicas que impactam o usuário, afinal cada um tem o seu objetivo e seu sistema de execução. Mesmo que os *hardwares* estejam sempre em evolução e cada vez mais rápido, aplicar as técnicas não só tem o ganho de desempenho mas também deixa o código mais legível, dando espaço para futuras manutenções e aumentando a qualidade do software.

Referências

BENTLEY, Jon. Writing Efficient Programs (Prentice-Hall Software Series). Prentice Hall Ptr. 1982.

Catch Fly - The Escape. 2022. Disponível em: https://github.com/rebeccamello/Catch-Fly. Acesso em 11, nov, 2022.

Compiler Optimizations. 2012. Disponível em: http://compileroptimizations.com/. Acesso em 10, out, 2022.

DARTE, Alaim; HUARD, Guilherme. New Results on Array Contraction. IEEE International Conference on Application-Specific Systems, Architectures and Processors. 2002.

GREGG, Brendan. Flame Graphs. Brendan's site. 2011. Disponível em: https://www.brendangregg.com/flamegraphs.html. Acesso em 02, jun, 2022.

LEISERSON, Charles; SHUN, Julian. 6.172 Performance Engineering of Software Systems. MIT OpenCourseWare. 2018. Disponível em: https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/pages/syllabus/. Acesso em 27, mai, 2022.

OSHANA, Robert. DSP Software Development Techniques for Embedded and Real-Time Systems. Newnes. 2005.

Stars And Exoplanets. Transit-Model-CRAAM. 2022. Disponível em: https://github.com/Transit-Model-CRAAM/pipelineMCMC. Acesso em 2, out, 2022.

SINGHAL, Akshay. Code Optimization Techniques. Gate Vidyalay. 2015. Disponível em: https://www.gatevidyalay.com/code-optimization-techniques. Acesso em 2, jun, 2022.

TIOBE Programming Community Index Definition. 2022. Disponível em: https://www.tiobe.com/tiobe-index/programminglanguages_definition/. Acesso em 02 novembro 2021.