Capítulo 1

Mais Produtividade com LLMs, Engenharia de Prompt, Aprendizado de Máquina e GPUs

Ricardo dos Santos Ferreira
Departamento de Informática, Universidade Federal de Viçosa - ricardo@ufv.br

Resumo

Este minicurso apresenta uma abordagem prática para utilizar modelos de linguagem de grande escala (Large Language Models - LLMs) como ferramentas para aumentar sua produtividade. Através de múltiplos exemplos práticos e metodologias bem estruturadas, demonstraremos como elaborar prompts eficazes para desenvolver ferramentas de visualização interativas, interfaces dinâmicas responsivas, ferramentas de simulação e interpretadores especializados para linguagens de domínio específico. O minicurso aborda desde os fundamentos da engenharia de prompts, explorando o estado da arte atual das ferramentas baseadas em LLMs e identificando as palavras-chave e estratégias para o desenvolvimento de soluções para visualização de dados, uso do ambiente Google Colab aliado ao uso de JavaScript, C++ e CUDA. Desenvolvemos exemplos nas áreas de aprendizado de máquina, arquitetura de computadores e programação paralela em GPU.

1.1. Introdução

O avanço dos Modelos de Linguagem de Grande Escala (LLMs - *Large Language Models*) tem um impacto direto no desenvolvimento de ferramentas aplicadas ao contexto educacional e de pesquisa. Uma pesquisa recente conduzida pela Universidade de Elon revelou que 52% dos adultos americanos já incorporaram em suas atividades cotidianas modelos de linguagem (Elon University 2025), evidenciando a rápida penetração dessas tecnologias na sociedade contemporânea. A eficácia das respostas está intrinsecamente relacionada à qualidade da requisição (ou *prompt*) fornecida ao modelo.

Neste capítulo, exploraremos múltiplas dimensões do emprego de LLMs na produção de material educacional e ferramentas de apoio à pesquisa utilizando Google Colab, Python, JavaScript, CUDA, interfaces interativas e interpretadores. Através de exemplos práticos, demonstraremos como estes modelos podem ser integrados ao processo de desenvolvimento de recursos de visualização, validação, simulação e teste de código. Elaboramos também exemplos envolvendo aprendizado de máquina para ilustrar a metodologia. Todos os prompts estarão documentados para ilustrar as palavras-chave com maior probabilidade de sucesso na elaboração de diversos softwares de apoio.

Este minicurso é transversal e pode ser aplicado nas diversas áreas de ciência da computação ou outros domínios. Os temas trabalhados nos exemplos estão mais correlacionados à: ciência de dados, aprendizado de máquina e computação de alto desempenho, arquiteturas de computadores, arquiteturas avançadas, dedicadas e específicas, avaliação, medição e predição de desempenho.

Este capítulo está organizado da seguinte forma. A Seção 1.2 apresenta as ferramentas, incluindo as LLMs, linguagens e suas bibliotecas juntamente com o ambiente do Jupyter Notebook/Google Colab.

1.2. Ambiente, Ferramentas, Linguagens e Bibliotecas

1.2.1. Ferramentas de LLM

Nos últimos anos, observou-se um crescimento exponencial nos trabalhos usando os recursos disponibilizados pelas LLMs, conforme evidenciado por uma pesquisa (Joel et al. 2024) que demonstrou a publicação de mais de 27 mil estudos no período de 2020 a 2024. Contudo, ainda há oportunidades para investigar conceitos inovadores, como a criação de representações visuais estruturadas, incluindo diagramas em blocos e outras áreas (Zala et al. 2023; Al-Shetairy et al. 2024).

Neste minicurso iremos utilizar as ferramentas nas suas versões gratuitas ChatGPT, Copilot e Claude, que apresentaram melhor desempenho em geração de código (Lisboa et al. 2025; Ságodi et al. 2024; Almanasra and Suwais 2025), usando as linguagens Python e JavaScript (Godage et al. 2025) devido à sua popularidade e facilidades para geração de código em ambientes de navegadores. Iremos ilustrar alguns exemplos com Gemini e DeepSeek, que vêm evoluindo, apresentando bom desempenho em relação ao ChatGPT (Vyas and BHARDWAJ 2025).

1.2.2. Engenharia de Prompt

A construção de *prompts* pode ser realizada por meio de diversas técnicas, buscando fornecer instruções claras e objetivas (Chen et al. 2023). Para isso, recomenda-se o uso de delimitadores, como aspas ou chaves, que ajudam a distinguir as instruções dos exemplos ou trechos a serem aprimorados. Quanto às estratégias utilizadas, elas podem ser agrupadas em três categorias principais:

- **Zero-shot**: consiste em utilizar um *prompt* direto, sem o apoio de exemplos.
- **One-shot**: inclui um único exemplo para orientar a resposta desejada.
- **Few-shot**: apresenta múltiplos exemplos, oferecendo maior contexto e refinamento.

Um estudo recente apresentado em (Chen et al. 2023) exemplifica as principais abordagens:

- **Chain-of-Thought (CoT)**: decompõe problemas complexos em etapas menores, explicando o raciocínio em cada uma.
- **Least-to-Most Prompting**: transforma um problema complexo em subproblemas simples, resolvidos em sequência.
- **Golden Chain-of-Thought**: além da decomposição lógica, fornece explicações detalhadas sobre o raciocínio em cada etapa.
- **Generated Knowledge**: utiliza a capacidade da LLM para gerar informações úteis antes

de produzir a resposta final.

- **Tree of Thoughts (ToT)**: explora múltiplos caminhos de raciocínio, permitindo avaliações, avanços e retrocessos, com maior interatividade.
- **Catalogação de Prompts**: busca identificar padrões sistemáticos para auxiliar na elaboração de estratégias eficazes.
- **Otimização de Prompts**: usa a própria LLM para criar e ajustar prompts, melhorando sua precisão e relevância de forma automatizada.

Neste minicurso, baseado em experiências anteriores (Lisboa et al. 2025), optamos pela estratégia de manter os *prompts* curtos, com as palavras-chave importantes, em uma versão mais simplificada da técnica **Chain-of-Thought (CoT)**. Ao escolher uma LLM para apoiar o desenvolvimento de soluções baseadas em linguagem natural, enfrentamos o dilema entre utilizar uma plataforma comercial, ainda que gratuita, ou adotar uma alternativa de código aberto. Embora os modelos abertos ofereçam maior reprodutibilidade, transparência e controle sobre os parâmetros, optamos pela solução comercial devido à sua simplicidade de uso, integração facilitada e atualizações constantes que garantem desempenho competitivo e estabilidade.

Neste contexto, adotamos a geração de código realizada por meio de uma metodologia que decompõe problemas complexos em etapas menores. Essa abordagem favorece a criação de soluções modulares, portáveis e reutilizáveis, reduzindo a dependência de configurações específicas da LLM utilizada. Assim, mesmo sem acesso direto ao código-fonte do modelo LLM, conseguimos manter a clareza, a eficiência e a adaptabilidade dos códigos gerados. Todos os *prompts* estão inclusos no material do minicurso.

1.2.3. Jupyter Notebook e Google Colab

O Jupyter Notebook é um ambiente que funciona por meio do navegador, oferecendo um método eficiente para criar documentação que inclui tanto trechos de código quanto explicações no mesmo documento (Rule et al. 2019), utilizando dois tipos de células: texto e código.

As células de texto permitem, além do próprio texto, o uso de linguagem de marcação (*markdown*), imagens e outros recursos.

As células de código podem ser escritas em diversas linguagens de programação, além de executar e exibir os resultados gerados.

A biblioteca IPython constitui a base do Jupyter Notebook. O ambiente pode ser instalado localmente no computador do usuário ou em um servidor.

O IPython surgiu em 2007 (Pérez and Granger 2007) como um ambiente interativo para execução no navegador, com suporte à visualização de dados, e encapsula o sistema operacional subjacente.

Os usuários podem navegar pelo sistema de arquivos com comandos Unix/Linux, adicionando o caractere prefixo "!" para executar operações de linha de comando. Um Jupyter Notebook usa o IPython como camada de virtualização para interagir com o sistema de arquivos.

Em 2017, a Google iniciou o oferecimento de um Jupyter Notebook com versões gratuitas na sua nuvem, denominado Google Colaboratory ou Google Colab. O Colab permite que um Jupyter Notebook execute em processadores de alto desempenho e aceleradores de hardware (GPUs e TPUs). Portanto, o Colab pode hospedar e fornecer aos estudantes as facilidades do

Jupyter Notebook sem a necessidade de instalações locais. O Colab é integrado ao Google Drive e GitHub, permitindo acesso fácil e gratuito a dados e códigos compartilhados.

Embora os exemplos disponíveis de Colab em várias áreas do conhecimento ofereçam recursos ricos para apresentar código e documentar trabalho em um único ambiente, a maior parte dos exemplos carece de mais explicações, já que seu foco não é o ensino (Rule et al. 2018). Este capítulo busca promover ideias para implementar ferramentas com auxílio dos modelos de linguagem LLMs usando o Colab. Mas todas as soluções podem também executar localmente se o usuário preferir instalar o Jupyter Notebook em seu computador ou em um ambiente de servidor com JupyterHub. A Tabela 1.1 apresenta as vantagens do Colab em comparação com um Jupyter Notebook instalado localmente.

Tabela 1.1. Comparação entre Jupyter Notebook e Google Colab

Jupyter Notebook	**Google Colab**
Principais Vantagens	
Execução totalmente local - maior controle sobre dados e privacidade	Acesso gratuito a GPUs e TPUs
Não depende de conexão à internet após instalação	Não requer instalação - executa direto no navegador
Personalização completa do ambiente de desenvolvimento	Colaboração em tempo real com outros usuários
Integração direta com sistema de arquivos local	Integração nativa com Google Drive
Suporte a múltiplos kernels (Python, R, Scala, etc.)	Ambiente pré-configurado com principais bibliotecas
Extensões e plugins personalizáveis	Compartilhamento fácil de notebooks via link
Sem limitações de tempo de execução	Sincronização automática na nuvem
Ideal para trabalhar com dados sensíveis	Ideal para prototipagem rápida e aprendizado

1.2.4. Interatividade

O IPython oferece uma gama de opções de botões de interface com a biblioteca *ipywidgets* e vários modos de interação com código Python. Devido à popularidade do uso de *ipywidgets*, a maioria dos modelos de LLM gera código com sucesso. Nosso primeiro exemplo desta seção irá avaliar a *DeepSeek* com um prompt para mostrar as várias opções de botões e entrada de dados de interface.

O *prompt* a seguir ilustra um exemplo para exploração das opções de interface. O resultado da execução do código gerado está ilustrado na Figura 1.1. Podemos observar campos para vários tipos de entrada: strings com a opção *Text*, texto com a opção *Textarea*, números inteiros ou float com botões deslizantes com *IntSlider* ou *FloatSlider*, lista de opções com *Dropdown*, ativar uma opção com *Checkbox* e botões com texto e cores com *Button*. O layout do grid de "botões"é controlado pelos recursos *VBox* e *HBox*. Usando estas palavras-chave é possível gerar rapidamente uma interface para o seu código.

Prompt de Ipywidgets com DeepSeek

Vamos agora mostrar o potencial dos ipywidgets em Python com o Google Colab. Fazer um exemplo de código com string, janela para editar texto, checkbox, slider, dropdown e botão para submeter.

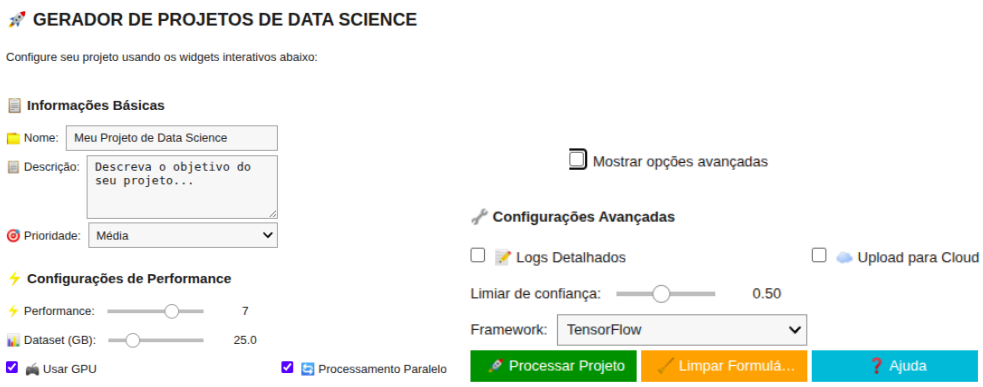

Figura 1.1. Exemplo de interface com vários botões e caixas de texto interativas da biblioteca ipywidgets. Para acesso ao exemplo clique aqui.

Outro ponto importante é a maneira como a interface interage com o código. Enquanto os *widgets* definem os elementos visuais e suas propriedades, o potencial do *IPython* está na forma como o comportamento desses componentes é conectado à lógica do programa.

Essas conexões permitem que o usuário modifique variáveis, execute cálculos ou dispare funções sem precisar editar o código diretamente, um recurso com aplicações didáticas e exploratórias, especialmente no *Google Colab*.

De modo geral, existem três abordagens principais para conectar widgets e funções em Python. O método *on_click* é o mais direto: associa uma ação específica à interação do usuário com um botão. Quando o botão é pressionado, a função vinculada é executada, permitindo, por exemplo, atualizar valores, limpar campos ou iniciar cálculos. Esse padrão é bastante útil em situações em que a ação do usuário deve ocorrer de forma controlada e explícita, como na execução de uma simulação ou na confirmação de uma escolha.

O método *interact*, por sua vez, é mais dinâmico e automatiza parte desse processo. Ele observa continuamente os valores dos componentes de interface e chama a função correspondente sempre que qualquer valor é alterado. Isso o torna ideal para experimentação interativa e ajustes rápidos de parâmetros em tempo real, muito usado em visualizações de dados ou no ajuste de hiperparâmetros de modelos de aprendizado de máquina. No entanto, para processos mais custosos ou que exijam múltiplas alterações antes de um resultado final, o *interact* pode tornar-se ineficiente, pois dispara a função repetidamente a cada modificação.

A terceira opção, o método *observe*, oferece um controle mais granular. Ele permite observar uma ou mais propriedades de um widget e reagir apenas quando um evento específico ocorre. Diferente do *interact*, não há necessidade de uma associação explícita a uma função: o desenvolvedor define manualmente quais alterações devem ser monitoradas e como tratá-las. Essa abordagem é particularmente útil em interfaces mais complexas, onde múltiplos elementos precisam reagir de maneira coordenada, ou quando se deseja controlar com precisão o fluxo de execução entre interações do usuário.

O trecho de código Python que ilustra as três opções também foi gerado com a *DeepSeek*, e as interfaces resultantes estão ilustradas na Figura 1.2.

Cada exemplo demonstra uma filosofia distinta de interação entre interface e lógica, e compreender essas diferenças é essencial para projetar experiências interativas eficientes e responsivas em notebooks Jupyter.

Response - DeepSeek

Exemplos de Interações Ipywidgets e código Python

```python
# on_click
def quando_clicar(botao):
    print(f"Bot o clicado! Contador: {contador.value}")
    contador.value += 1
botao = widgets.Button(description="Clique aqui!", button_style='success')
botao.on_click(quando_clicar)
# interact
@interact(nome="Jo o",
    idade=widgets.IntSlider(value=25, min=0, max=100, description="Idade:"),
    python_expert=False,
    cor=widgets.Dropdown(options=['Vermelho', 'Verde', 'Azul'], description="Cor:")
)
def cumprimentar(nome, idade, python_expert, cor):
...
# observe
def quando_mudar(change):
    print(f"Valor mudou: {change['old']}      {change['new']}")
    resultado.value = f"Resultado: {slider.value * 2}"
    barra_progresso.value = slider.value
slider = widgets.IntSlider(value=5, min=0, max=10, description="Valor:")
resultado = widgets.Label(value="Resultado: 10")
barra_progresso = widgets.IntProgress(value=5, min=0, max=10)
slider.observe(quando_mudar, names='value')
```

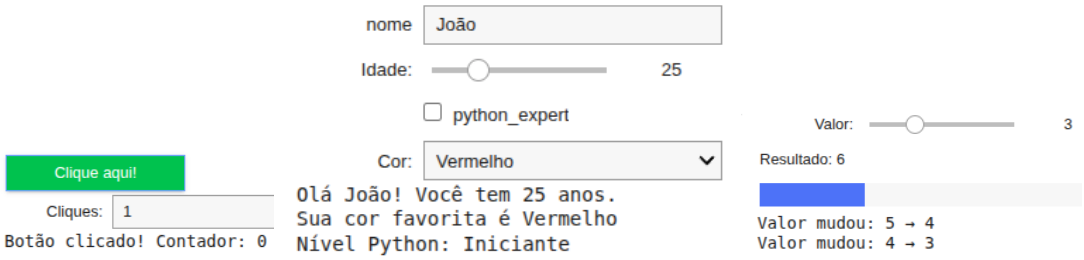

Figura 1.2. Interfaces: *on_click*, *interact*, *observe*. **Para acesso ao exemplo clique aqui.**

A Tabela 1.2 mostra um resumo e indica que existem outras opções:

- **interact_manual**: Adiciona um botão para controle de execução.
- **interactive**: Oferece mais controle sobre layout e widgets.
- **on_submit**: Para capturar quando a tecla Enter é pressionada.

Tabela 1.2. Casos de Uso Recomendados

Método	Melhor Aplicação
on_click	Submissão de formulários, ações de confirmação, processos que requerem intenção explícita
interact	Exploração rápida de dados, protótipos, demonstrações interativas, ensino
observe	Interfaces responsivas, atualizações em tempo real, cálculos instantâneos
interactive	Aplicações com layout customizado, interfaces complexas, produção
on_submit	Campos de busca, entradas de texto que devem processar dados ao pressionar Enter

Além do Ipywidgets, podemos usar o pacote *Gradio* (Ferreira et al. 2024), que possibilita a separação entre a interface e o código. O *Gradio* permite criar interfaces gráficas simples e interativas para funções Python. Com poucas linhas de código, você pode transformar qualquer função Python em uma aplicação web acessível, facilitando testes, demonstrações e compartilhamento de projetos. Ele suporta diversos tipos de entrada e saída, como texto, imagem, áudio e vídeo, e é amplamente usado para prototipagem rápida e integração com plataformas como Hugging Face. Antes da explosão das LLMs, a interface *Gradio* se popularizou com vários exemplos gratuitos de ferramentas de imagem com aprendizado profundo. Entretanto, após a explosão do uso das LLMs e a escassez de recursos de GPU em servidores, a maioria das demonstrações foi desativada.

Além das facilidades de interfaces, mais flexíveis e com mais recursos que os *ipywidgets*, o *Gradio* com acesso remoto permite isolar a implementação da ferramenta e sua interface. Outro aspecto é monitorar o uso das ferramentas, pois pode gerenciar de forma transparente várias conexões e gerar estatísticas de uso ou mesmo monitorar atividades dos estudantes (Ferreira et al. 2024).

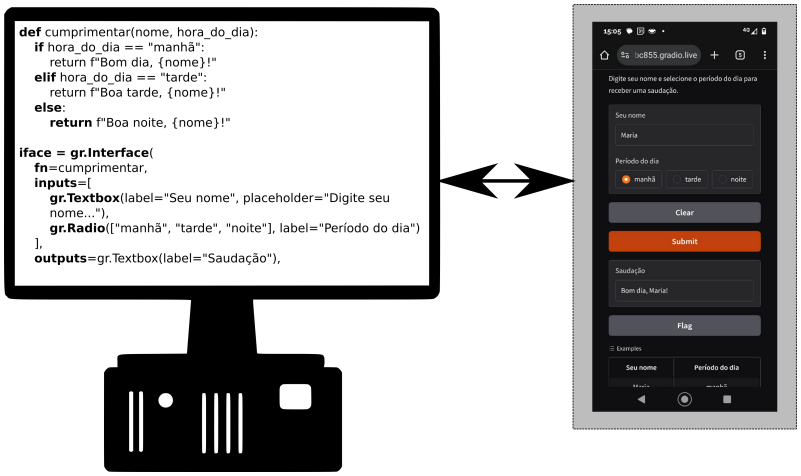

Figura 1.3. Acesso remoto com dispositivo móvel ao Google Colab via Gradio e Hugging Face. Para acesso ao exemplo clique aqui.

A Figura 1.3 ilustra um exemplo simples de código que é executado no Google Colab do desenvolvedor e acessado remotamente por um dispositivo móvel, no exemplo um celular, através

de uma URL disponibilizada pelo Hugging Face automaticamente quando executamos o código no Google Colab. O código exemplo foi gerado pelo DeepSeek com um simples prompt: “agora vamos usar a interface Gradio para dar exemplos”.

1.2.5. Comandos Mágicos

O Jupyter Notebook permite a criação de comandos mágicos. Estes comandos irão executar um código para processar o conteúdo da célula. Desta maneira, é possível executar qualquer linguagem, desde que o compilador da linguagem esteja instalado e não tenha um uso complexo de interface gráfica. Existem vários comandos que já estão pré-instalados. O comando mágico começa com % (para comandos de linha) ou %% (para comandos de célula inteira). As tarefas mais comuns incluem medir tempo de execução, executar código em outra linguagem, manipular arquivos, entre outros.

Podemos criar comandos novos. Para ilustrar, avaliamos a capacidade da *DeepSeek* na geração de um comando mágico para compilar a linguagem C. Usando um *prompt* bem simples “Criar comando mágico para C no Colab”, geramos um resultado correto e satisfatório como ilustrado na Figura 1.4.

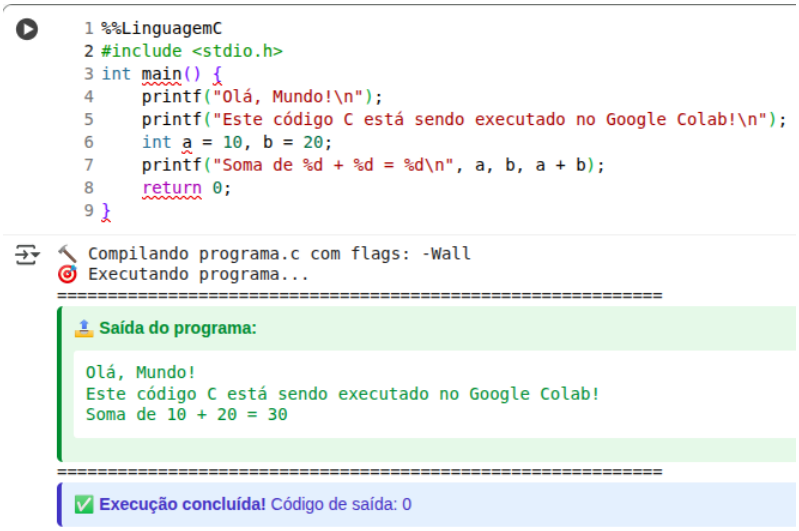

Figura 1.4. Criação de um comando mágico para compilar e executar um código C. Para acesso ao exemplo clique aqui.

O pacote *Cad4U* (Canesche et al. 2021) apresenta vários exemplos que foram desenvolvidos para arquitetura de computadores, incluindo a compilação de várias linguagens como Verilog/VHDL, ferramentas como Valgrind e gem5, além de comandos específicos como *print_verilog*, que usa o pacote *Yosys* para desenhar o código Verilog. Os comandos mágicos podem encapsular a instalação de ferramentas e simplificar o ensino ou uso de scripts para ferramentas de pesquisa.

1.2.6. Linguagens

Como visto na seção anterior, é possível instalar suporte para várias linguagens no Jupyter Notebook ou Google Colab, além de ser possível associar um comando mágico para simplificar a compilação e execução do código.

Nesta seção, iremos ilustrar o uso de JavaScript gerado pelas LLMs para criar interfaces de ferramentas e pequenas demonstrações. A maior parte deste capítulo faz uso de Python e do

suporte nativo no Jupyter Notebook. Porém, para usar a maioria das linguagens, incluindo Python, é necessário conectar o notebook ao servidor, seja através do Google Colab, servidor local na sua rede ou diretamente instalado no seu computador. Entretanto, a linguagem JavaScript oferece suporte para execução no navegador, mesmo dentro do Google Colab, sem a necessidade de conexão. Pode também executar no navegador de seu celular para exemplos mais simples.

Devido à sua popularidade e disponibilidade de código, as LLMs são capazes de gerar código JavaScript com facilidade, assim como Python (Godage et al. 2025). Das LLMs avaliadas, a LLM Claude apresenta o melhor desempenho para JavaScript.

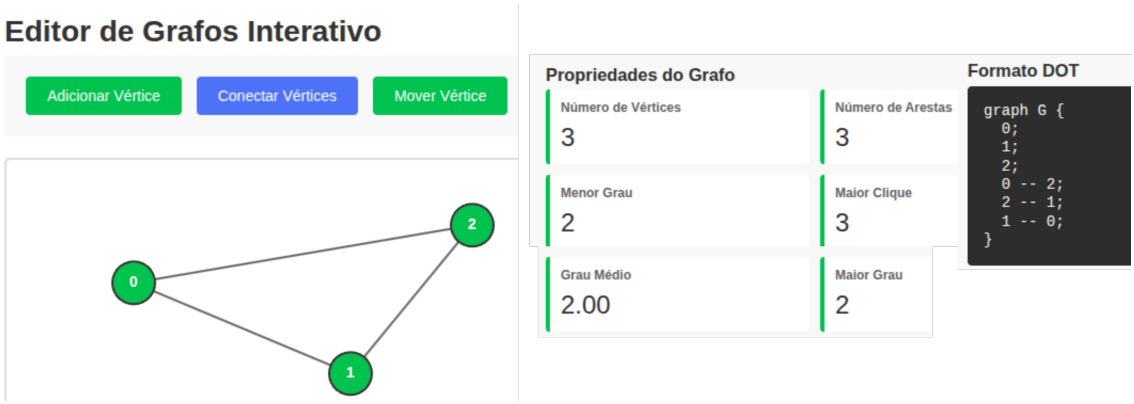

Figura 1.5. Editor de Grafos para exemplos com JavaScript usando a LLM Claude. Para acesso ao exemplo clique aqui.

Usando o prompt abaixo para especificar um editor de grafos, a ferramenta Claude criou sem dificuldade um código funcional em JavaScript, como ilustra a Figura 1.5.

Prompt de Editor de Grafos com Claude em JavaScript

Fazer um código JavaScript para um editor de grafos onde é possível adicionar e mover vértices, conectar vértices. Colocar também um botão para medir as propriedades do grafo: grau médio, maior grau, menor grau, maior clique, número de vértices e número de arestas. Mostrar também o grafo descrito no formato dot. O editor deve executar dentro do ambiente Google Colab.

Outro exemplo desenvolvido foi um editor e simulador de máquina de estados com apenas um prompt e uma tentativa.

1.2.7. Visualização

Existem várias opções para visualização de gráficos, grafos, vídeos e animações que podem ser facilmente adicionadas ao Google Colab. Iremos ilustrar alguns exemplos com auxílio das LLMs para geração de código.

Iremos começar com a visualização de grafos. Embora a biblioteca Matplotlib desenhe grafos, o mais aconselhável é usar a biblioteca Graphviz para visualização e a biblioteca NetworkX para os algoritmos.

Vamos ilustrar um exemplo simples de um gerador com o algoritmo da árvore geradora mínima. Usamos o seguinte *prompt*:

Prompt de gerador de Grafos e algoritmo de árvore geradora com ChatGPT

Usando Google Colab, NetworkX e Graphviz para desenhar, fazer um gerador de grafos aleatórios de 10 a 20 vértices e arestas com pesos, com um botão ipywidget para gerar. Um botão para desenhar passo a passo com uma animação a árvore geradora mínima com algoritmo de Kruskal, ter um botão para dar um passo à frente ou para trás na execução do algoritmo.

Um editor ou gerador de grafos é interessante como entrada para várias ferramentas e pode ser desenvolvido com a ajuda da LLM. Entretanto, neste exemplo, a LLM ChatGPT teve um pouco de dificuldade para enquadramento do grafo para visualização. Após uma evolução com quatro *prompts*, fizemos o exercício de usar a LLM DeepSeek, que foi bem-sucedida para revisar o código da ChatGPT. A Figura 1.6 ilustra o resultado. Uma avaliação mais profunda das LLMs com dois algoritmos de grafos e outros exemplos de estrutura de dados com visualização foi apresentada recentemente em (Lisboa et al. 2025).

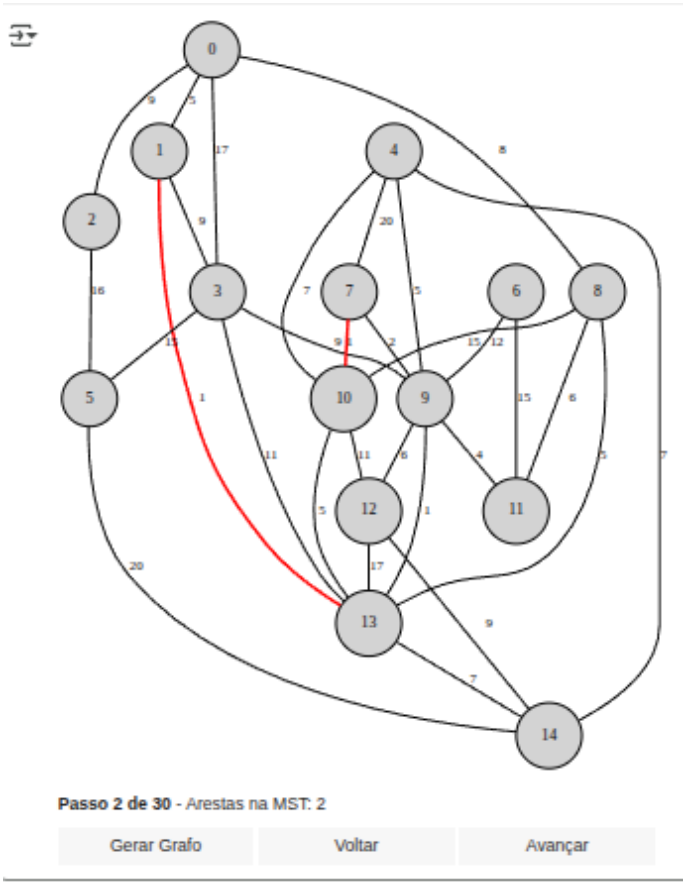

Figura 1.6. Árvore geradora mínima com Chatgpt e Deepseek usando graphviz e networkX. Para acesso ao exemplo Clique aqui

Outro recurso de visualização é o desenho de animações no formato vetorial SVG com auxílio da biblioteca svgwrite. A vantagem é que o código gerado pode ser editado e ajustado. Por exemplo, o prompt a seguir solicita uma geração de uma animação para um desenho de cache. Apenas 2 tentativas com o prompt foram realizadas nas LLMs DeepSeek e ChatGPT. O resultado parcial está ilustrado na Figura 1.7(a). Podemos observar que serão ainda necessá-

rios vários ajustes até convergir para um desenho sem sobreposições e com um posicionamento melhor. A grande vantagem é ter um código de partida com recursos programáveis, como o trecho abaixo:

```
# desenha 4 células de dados
cell_w = (cache_line_w - 120) / 4
for j in range(4):
    cx = lx + 112 + j * cell_w
    cy = ly + 12
```

Prompt de gerador de desenho com svgwrite

Usando a biblioteca svgwrite no Google Colab, fazer uma animação de uma cache com 4 linhas e blocos de 4 elementos em cada linha. Simular um miss e os 4 elementos do bloco são buscados na memória e atualizados na cache.

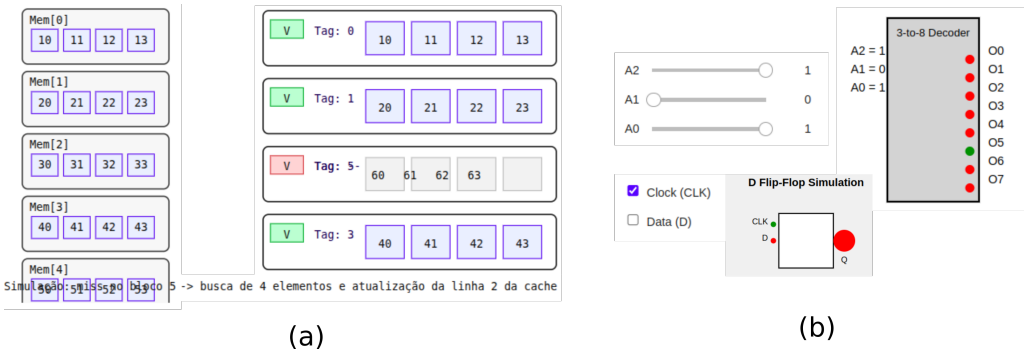

Figura 1.7. (a) Animação de Cache. (b) Decodificador e Flip-flop D extraídos de (Jamieson et al. 2025). Para acesso ao exemplo clique aqui.

Entretanto, no estágio atual das LLMs de uso geral, a qualidade das figuras ainda deixa a desejar. A Figura 1.7(a) mostra que temos um bom desenho, mas com sobreposições e desalinhamentos. Com uma sequência de *prompts*, é possível melhorar o desenho, mas ainda é limitado a exemplos de baixa complexidade. A Figura 1.7(b) mostra dois exemplos que foram criados usando esta técnica no ensino de lógica digital, apresentados juntamente com outras técnicas de LLM (Jamieson et al. 2025).

Outra alternativa é usar um editor para gerar uma ilustração ou figura como base. Esta ideia foi proposta em (de Figueiredo et al. 2024) para um simulador do caminho de dados do processador RISC-V. O desenho foi elaborado usando um editor vetorial para o formato SVG. Como o caractere "@" não é usado, os rótulos do desenho onde as variáveis têm valores (registradores, sinais de controle) foram rotulados com, por exemplo, "@registradorA" para o número do registrador A. O simulador gera um traçado da execução passo a passo com os valores das variáveis. De posse da sequência de valores, usando apenas substituição de strings, o desenho é atualizado com o valor passo a passo e gera uma imagem de cada passo. Depois, podemos navegar passo a passo ou gerar um vídeo.

Para exemplificar esta técnica, iremos elaborar um exemplo simples de um simulador de cache de mapeamento direto com 4 linhas. Primeiro, a LLM gera o simulador e o traçado, que é armazenado em um arquivo. Usamos o seguinte *prompt*:

Prompt de gerador de um simulador de cache com uma entrada editável e uma saída em arquivo

Simulador de cache no Colab para mapeamento direto com 4 linhas e blocos com 4 dados. Suponha uma memória de 1024 de tamanho, onde a posição i tem o conteúdo i. A cache inicial tem lixo no tag e nos dados, e os bits valid iguais a zero.
Receber uma sequência de acesso de uma janela de edição, como por exemplo: 4 6 10 0 16.
Para cada acesso, atualizar a cache e gerar um arquivo de trace com MSB, LSB, OFFSET do endereço.
Depois VALID0 MSB0 DATA0 (um para cada uma das 4 linhas da cache) e escrever o conteúdo MSB=0 LSB=1 OFFSET=0 VALID0=0 MSB0="lixo"DATA0="lixo"VALID1=1 MSB1=0 DATA1=4 5 6 7, VALID2= ... VALID3= ...
Neste exemplo, o primeiro acesso foi no endereço 4.

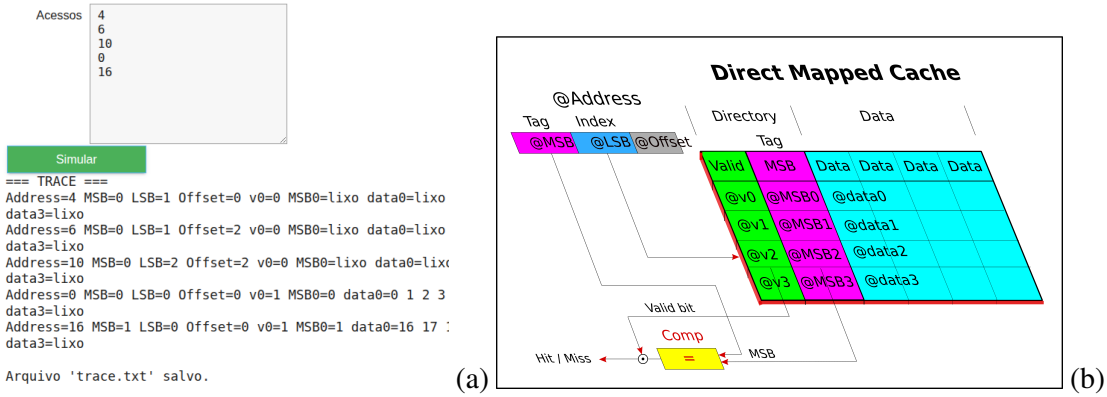

Figura 1.8. (a) Entrada da sequência de endereços para simulação e trecho do traçado produzido. (b) Figura SVG com prefixo "@" nos rótulos que serão substituídos pelos resultados da simulação. Para acesso ao exemplo clique aqui.

Este *prompt* irá gerar o resultado da Figura 1.8(a), que mostra a tela do simulador de cache gerado com interface para digitar qual sequência de endereços para cache. O simulador gera o traçado. Na Figura 1.8(b), vemos o desenho da cache onde queremos exibir o traçado da simulação passo a passo. Solicitamos à LLM com o *prompt* a seguir para fazer a substituição e depois gerar as imagens.

A substituição simples de rótulos no SVG, por exemplo, `@VALID0`, `@MSB1`, `@DATA2`, pode ser feita por operações de texto (string replace). O formato do arquivo de traçado deve ser consistente e fácil de analisar. O formato sugerido é: um bloco por passo contendo cabeçalho com o endereço (MSB, LSB, OFFSET) seguido de quatro linhas com o estado de cada linha de cache (VALID, MSB, DATA). Exemplo de linha do trace (texto): `STEP 0: ADDR MSB=0 LSB=1 OFFSET=0`
`VALID0=0 MSB0="lixo"DATA0="lixo"`
`VALID1=1 MSB1=0 DATA1="4 5 6 7"...` Ter um formato regular permite escrever

um parser simples que gera, para cada passo, um dicionário de mapeamento `{"VALID0": 0, "MSB1": 0, "DATA1": "4 5 6 7", ...}`.

> Prompt de gerador de um simulador de cache com uma entrada editável e uma saída em arquivo
>
> Agora que temos o arquivo trace.txt, considere que você tem um arquivo cache.svg que tem o desenho. Ler este arquivo e substituir os valores. Ler o arquivo Trace, linha por linha, para cada linha. No arquivo cache.svg, terá @MSB que deverá ser substituído pelo valor da linha. Por exemplo, MSB=0, então trocar @MSB por 0. Fazer para todas as variáveis do trace. Cuidado que @MSB e @MSB1 devem primeiro fazer o match de MSB1 para depois fazer do MSB. Ao substituir por linha, do novo SVG gerar um PNG trace1.png, depois para linha 2 trace2.png, sempre reler o cache.svg inicial.
> Fazer também, usando os arquivos PNG na pasta cache_traces/traceX.png, onde X é o número, gerar dois botões next e previous para mostrar o conteúdo dos traces, começando do trace1.png no Google Colab.

O resultado pode ser visto na Figura 1.9, onde temos os botões de navegação *next* e *previous* para ir para frente e para trás. Podemos observar o conteúdo da cache sendo atualizado e o endereço corrente e sua decomposição nos três campos: tag, linha e bloco.

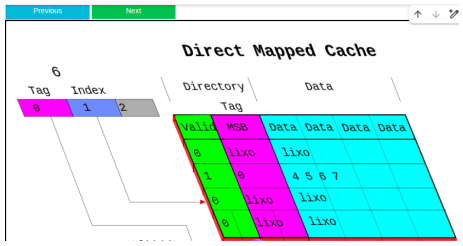

Figura 1.9. Resultado da animação da simulação do traçado da cache.

Outra alternativa eficaz é usar os recursos de JavaScript para exibição dos traçados. Neste caso, usaremos o mesmo simulador e a LLM Claude para gerar a interface visual com JavaScript. Solicitamos a interface com destaque para linha em uso e a opção para adicionar o traçado em uma janela de edição, uma vez que é mais complexo fazer o JavaScript acessar o sistema de arquivos do Google Colab.

A visualização do traçado com JavaScript pode fazer uso de interfaces animadas. Um exemplo é ilustrado na Figura 1.11, onde o traçado de um código Verilog para simulação de uma máquina de estados é visualizado. A máquina de estados possui duas chaves "A" e "B". Se "A" estiver ligada, o LED irá alternar entre aceso e apagado a cada ciclo, e se ambas as chaves estiverem ligadas, irá permanecer dois ciclos aceso e um ciclo apagado. A visualização permite a verificação sem a necessidade de analisar diagramas de forma de onda ou traçados em formato texto.

1.2.8. Interpretadores e Linguagens de Domínio Específico

As LLMs podem gerar interpretadores dedicados a uma determinada sintaxe expressa através de exemplos. Porém, é possível também definir uma linguagem e ter uma construção de um parser com uma gramática bem definida, o que facilita a extensão da ferramenta para adicionar novos comandos.

Figura 1.10. Resultado da animação da simulação do traçado da cache com JavaScript e LLM Claude.

Figura 1.11. Resultado da animação de um traçado de máquina de estados em Verilog com duas chaves e um LED usando JavaScript e LLM Claude. Acesso ao exemplo clique aqui.

A biblioteca *lark-parser* do Python pode ser usada como base. Ilustramos com um exemplo simples de uma linguagem em português para encapsular a biblioteca scikit-learn (Pedregosa et al. 2011) de aprendizado de máquina. A gramática será integrada ao código, facilitando a extensão e inclusão de novas funcionalidades. Nesta seção, iremos elaborar um exemplo e outros exemplos podem ser encontrados em (Coura et al. 2025). Nosso exemplo, inicialmente, criará uma linguagem com os comandos ilustrados na Figura 1.12.

1.3. Arquiteturas Paralelas

O ensino e pesquisa das arquiteturas paralelas foi teórico por várias décadas para a maioria dos estudantes. Nas últimas duas décadas, o acesso a máquinas paralelas se difundiu. Porém, muitas vezes a curva de aprendizado para uma máquina específica de pesquisa é demorada em

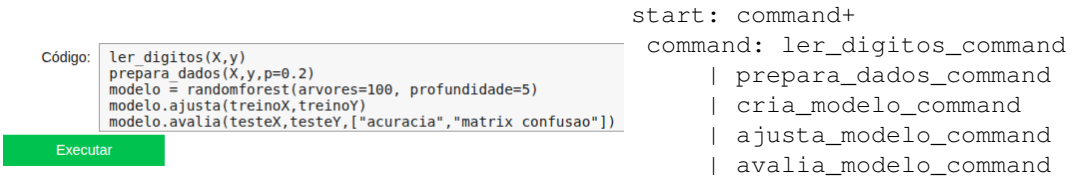

Figura 1.12. Exemplo de Linguagem de Domínio Específico com sua gramática correspondente com um trecho da gramática. Acesso ao exemplo clique aqui.

função da documentação escassa e configuração das máquinas. As LLMs permitem a criação de ferramentas de ensino e pesquisa de várias arquiteturas paralelas aliadas à construção de linguagens, interpretadores e simuladores. Portanto, os estudantes e pesquisadores podem elaborar exemplos para explorar mais rapidamente o espaço de ideias, conceitos e detalhar a implementação de forma interativa. Nesta seção, iremos ilustrar alguns exemplos de arquiteturas: arquiteturas vetoriais na Seção 1.3.1, arranjos de processadores ou *array processors* na Seção 1.3.2 e multiprocessadores na Seção 1.3.3.

1.3.1. Vetoriais

A programação em **assembly vetorial** permite manipular dados em blocos como vetores, reduzindo a sobrecarga de operações individuais, dos laços que geram testes e desvios condicionais e permitindo a exploração do paralelismo. Iremos ilustrar com um exemplo prático que inclui **load/store** de dados e operações vetoriais e escalares. O exemplo, apesar de simples, ilustra a criação de um subconjunto de instruções vetoriais para um assembly educacional. Suponha os exemplos de instruções do trecho de código a seguir, onde podemos ler e gravar vetores da/para memória e os registradores vetoriais, realizar operações com vetores e com escalares.

Assembly Vetorial

Exemplo de algumas instruções vetoriais

```
LOADV V1, A        ; Carrega vetor A em V1
LOADV V2, B        ; Carrega vetor B em V2
ADDV V3, V1, V2    ; Soma vetorial: V3 = V1 + V2
STOREV C, V3       ; Salva o resultado V3 no vetor C
LOAD S1, X         ; Carrega escalar X
MUL S2, S1, Y      ; Multiplica X por Y e armazena em S2
STORE Z, S2        ; Salva o resultado escalar em Z
```

As arquiteturas vetoriais foram pioneiras em vários avanços, como ILLIAC-IV no pós-guerra, um dos primeiros computadores vetoriais. Posteriormente, as várias máquinas da Cray nas décadas de 80 fizeram a evolução dos pipelines com várias unidades funcionais. Na década de 90, com as extensões MMX para os processadores Pentium, que introduziram a vetorização nos computadores pessoais para o processamento gráfico. Esta extensão evoluiu para as versões SSE, SSE2 e, mais recentemente, para as extensões AVX. Atualmente, com a demanda na área de inteligência artificial, as extensões vetoriais estão sendo propostas para várias versões dos processadores RISC-V.

Inicialmente, para ensino, é importante adicionar alguns recursos como a capacidade de executar o código e ter uma janela de edição para modificá-lo. Além disso, a interface de execução irá mostrar a visualização de parte da memória, dos registradores escalares e vetoriais.

Suponha a definição de um subconjunto de instruções em assembly vetorial para criação de uma

ferramenta com o auxílio das LLMs. O modelo inicial proposto tem a seguinte especificação para a arquitetura:

- **Registradores vetoriais** `V0` a `V7`: vetores de 8 elementos, com suporte a *stride*.
- **Registradores escalares** `F0` a `F7`: valores escalares.
- **Memória**: acessível por janela separada.

Em relação às instruções, podemos começar com o subconjunto mínimo:

`VLOAD Vx, addr, stride` Carrega 8 elementos da memória para `Vx`.

`VSTORE Vx, addr, stride` Armazena o conteúdo de `Vx` na memória.

`VADD Vx, Vy, Vz` Soma vetorial: `Vx` recebe o resultado de `Vy + Vz`.

`VMUL Vx, Vy, Vz` Multiplicação vetorial: `Vx` recebe o produto de `Vy` e `Vz`.

`VBROADCAST Vx, Fk` Preenche todos os elementos de `Vx` com o valor escalar `Fk`.

`VREDUCE_SUM Fk, Vx` Soma todos os elementos de `Vx` e armazena o resultado em `Fk`.

A Figura 1.13 ilustra o simulador com janela de edição. Um exemplo inicial é carregado. O estudante pode modificar, executar e visualizar os resultados nos registradores e memória. A primeira versão tem um subconjunto restrito e não pode executar laços. Uma segunda versão foi então gerada pelas LLMs para incorporar laços. Para demonstrar exemplos mais avançados, usaremos dois exemplos desenvolvidos no trabalho proposto em (Ferreira and Nacif 2025).

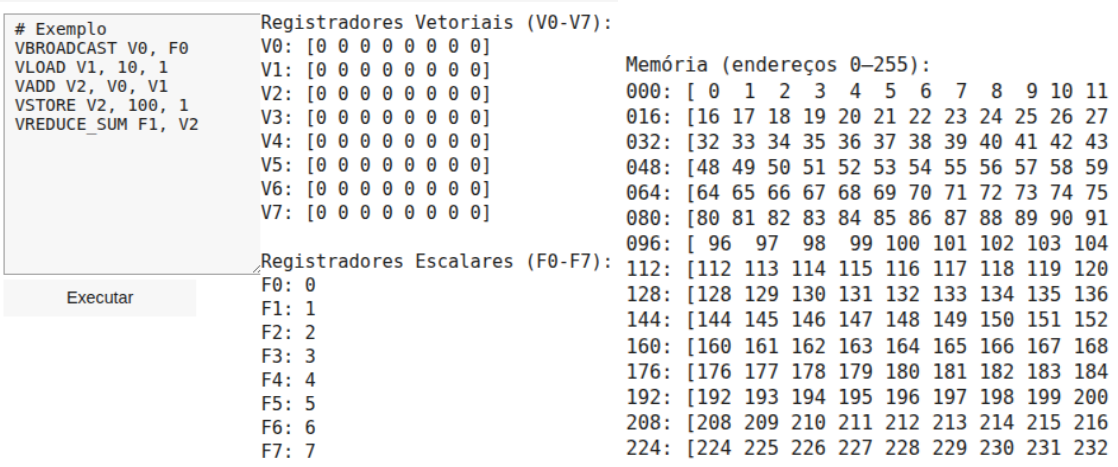

Figura 1.13. Simulador com uma linguagem vetorial, incluindo a janela de edição e botão para executar o código com a visualização dos registradores e da memória. Acesso ao exemplo clique aqui.

O primeiro exemplo é o código de multiplicação de matrizes. O simulador permite a edição dos dados, a edição do código para fazer outras versões, desde que use as mesmas instruções do assembly vetorial. Como a validação é um problema crítico em códigos gerados por LLM, que neste exemplo gerou, além do simulador, o código da multiplicação de matrizes, solicitamos à LLM para gerar um código Python para multiplicação de matrizes para fazer a contraprova da execução. Para fins didáticos, foram adicionados três modos de execução: tudo, uma instrução ou passo e a opção de avançar mais rapidamente com *n* instruções.

O segundo exemplo é o algoritmo TEA de criptografia com as etapas de codificação e decodificação. Ele trabalha com um par de elementos do vetor e 4 chaves secretas. Podemos vetorizar fazendo a execução de vários pares em paralelo com as instruções vetoriais. Para cada par, executamos uma sequência de mais de 500 operações de soma, XOR e deslocamento. O vetor é então criptografado. Para fazer a validação, implementamos a decodificação TEA, que também executa uma longa sequência de operações onde a vetorização explora a aplicação paralela sobre vários pares do vetor de dados. Ao final, podemos observar que o dado de entrada é recuperado, e podemos comparar a execução vetorial com uma execução sem vetorização.

1.3.2. Array Processor SIMD

O primeiro ponto aqui é deixar a distinção clara entre arquiteturas de vetores de processadores (*array processors*) das arquiteturas de processadores vetoriais (*vector processors*). O *array processor* é definido por um conjunto de elementos de processamento (PE ou *processing elements*) arranjados em uma determinada topologia de conexão. Pode ser um vetor de n PEs, um anel de n PEs, uma matriz de $n \times n$ PEs, etc. Todos os processadores (PEs) irão executar a mesma operação. Portanto, é uma arquitetura SIMD (*single instruction multiple data*) seguindo a taxonomia introduzida por Flynn.

Nos livros didáticos de arquiteturas paralelas existem várias ilustrações e topologias de *array processors*, porém poucos *array processors* foram prototipados em hardware e são raros os casos de máquinas comerciais. Com o apoio das LLMs, é possível criar um simulador de uma determinada arquitetura e ao mesmo tempo definir uma linguagem de programação.

Primeiro, temos que definir um modelo de memória. Podemos começar com um modelo simples, onde cada PE tem uma ou mais variáveis locais. Em seguida, definimos um conjunto de instruções simples aliado a uma máscara de execução. Como o programa é único e será disparado em todos os PEs, cada PE individualmente pode executar ou não a instrução corrente.

Como primeiro exemplo, iremos ilustrar uma implementação do algoritmo de ordenação paralela com trocas dos elementos vizinhos do vetor. O *array processor* tem 8 PEs ligados em linha, onde o PE_i é conectado aos PE_{i-1} e PE_{i+1}. Cada PE tem uma variável de estado que armazena o valor do elemento. O conjunto de instruções do simulador suporta a operação *SWAP*, que troca as variáveis de estados dos PE_i e PE_{i+1}. A segunda instrução é *CMPXCHG*, que só efetua a troca se o valor da variável i for maior que o valor da variável $i+1$. Outras duas instruções são *SENDLEFT* e *SENDRIGHT*, que podem enviar o valor da variável i para seu vizinho da esquerda ou da direita. Note que todas as instruções estão vinculadas ao número do PE. Uma máscara irá dizer quais os PE que irão operar. A Figura 1.14 mostra um trecho do código do interpretador/simulador com o processamento das instruções, ilustrando que novas instruções podem ser facilmente adicionadas.

O algoritmo de ordenação paralela par-ímpar de n elementos com n processadores tem complexidade $O(n)$. A Figura 1.14 ilustra o simulador e interpretador *array processor* com o código da ordenação. Como usamos apenas 8 elementos para ilustrar, o algoritmo executa em três iterações do laço com duas instruções *CMPXCHG* com as máscaras par e ímpar, respectivamente. Ao final do terceiro passo, o vetor está ordenado.

Um segundo exemplo foi construído para executar uma multiplicação de matrizes. Neste caso, a arquitetura *array processor* tem $n \times n$ processadores. Fizemos uma demonstração com a LLM gerando código para o exemplo 3×3. Como topologia, temos uma grade ou malha em duas dimensões dos PE. Temos três variáveis por PE: a, b e c. Como instruções, podemos distribuir

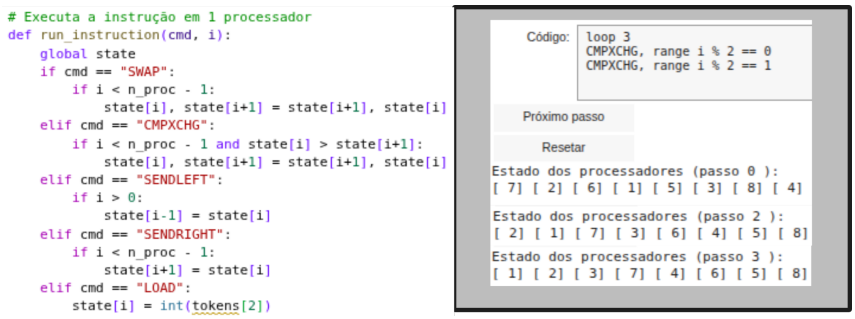

Figura 1.14. Simulador com uma linguagem Array Processor SIMD, incluindo a janela de edição e botão para executar o código com a visualização do array de processadores com o exemplo do algoritmo de ordenação par/ímpar. Clique aqui.

os dados enviando de uma linha para outra ou de uma coluna para outra em paralelo. Por exemplo, um *shift up A* irá mover o valor de A para as linhas acima (pode ou não executar com incremento em módulo, para primeira linha enviar para última). Além disso, pode ter máscara para que apenas algumas linhas ou colunas executem.

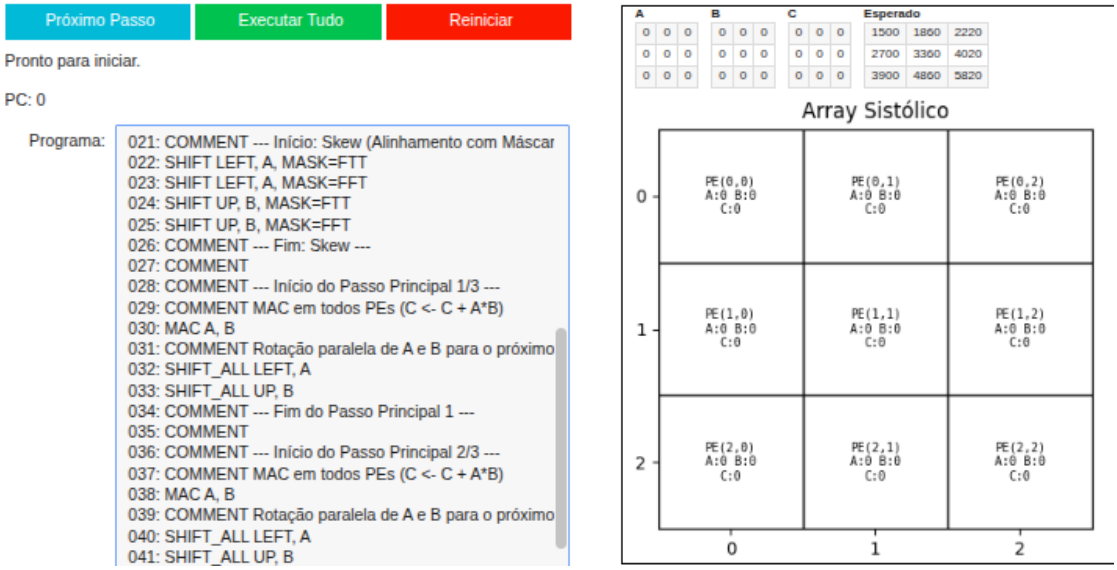

Figura 1.15. Simulador com uma linguagem Array Processor SIMD, incluindo a janela de edição e botão para executar o código com a visualização dos elementos de processamento (PE). Clique aqui.

A Figura 1.15 ilustra o interpretador/simulador do *array processor* 2D. Do lado esquerdo, temos a janela de código para executar o programa. A maior parte do algoritmo está concentrada no reposicionamento dos dados. No lado direito, temos os valores das variáveis a, b e c para cada *PE*, além do desenho da arquitetura em uma grade com duas dimensões. A linguagem do interpretador tem *shift* nas quatro direções *up*, *down*, *left* e *right*, além de uma máscara com *True* ou *False* para ativar ou não o movimento de cada linha ou coluna. No exemplo da Figura 1.15, vemos MASK=FTT, que irá executar o movimento apenas para linhas ou colunas 1 e 2 (True) e não executa para linha/coluna 0 (False). A instrução MAC (multiplica e acumula) irá executar a operação $C = C + A \times B$. Ao final, podemos ter a validação com o resultado comparando com uma implementação em Python para verificar se a programação paralela foi

implementada corretamente. Podemos observar no alto do simulador, acima da janela dos *PEs*, qual é o valor esperado para a multiplicação.

1.3.3. Multiprocessadores

Em relação às arquiteturas com multiprocessadores, diferente dos *array processors*, temos vários exemplos comerciais, uma vez que esta é a arquitetura predominante atualmente, onde praticamente todos os processadores de computadores pessoais e dispositivos móveis como telefones celulares usam um multiprocessador com múltiplos núcleos.

Em termos acadêmicos, temos dois modelos básicos: memória compartilhada e troca de mensagens. Os estudantes podem usar a programação com OpenMP ou MPI para exercitá-los nas máquinas comerciais. Com fins de ensino e pesquisa, nesta seção iremos ilustrar um exemplo simples para cada modelo.

Com apoio da LLM, criamos um ambiente inicial com um multiprocessador com três processadores. O programador tem uma janela de edição de código para cada processador, conforme ilustrado na Figura 1.16. Para ilustrar o funcionamento do modelo com memória compartilhada, iremos começar com um exemplo bem simples.

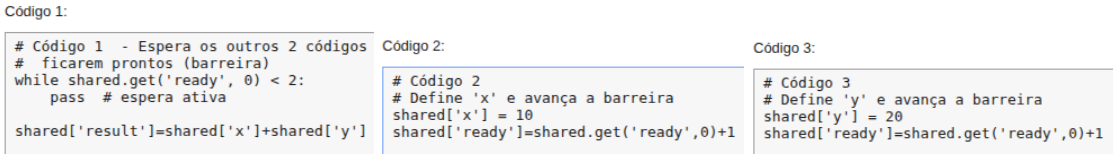

Figura 1.16. Simulador com três multiprocessadores e suas janelas de código, usando a linguagem Python e sincronismo com barreira através de memória compartilhada. Clique aqui.

O programador pode definir uma variável compartilhada do tipo *shared*. Associada a cada variável *shared*, pode executar o método *get* para saber o valor da variável. No exemplo ilustrativo da Figura 1.16, o processador 1 irá aguardar que a variável compartilhada *ready* tenha o valor 2 para prosseguir. Os processadores 2 e 3 irão definir um valor para as variáveis compartilhadas *x* e *y*, respectivamente. Depois, ambos os processadores irão incrementar o valor da variável *ready*, que irá disparar o processador 1 para sair do modo de espera, para então somar os valores das variáveis *x* e *y*. O interessante na implementação gerada pela LLM, além das três janelas de edição e de um mecanismo simples de sincronização, é que podemos programar código Python na janela de cada processador. O interpretador/simulador usa o mecanismo de thread do Python para gerenciar o sincronismo.

Com relação ao segundo modelo de multiprocessador com comunicação por troca de mensagens, implementamos, com auxílio das LLMs, um simulador com três processadores e três janelas de código, como ilustrado na Figura 1.17. Dois comandos fazem a comunicação: *receive* e *send*. O *receive(x)* aguarda a mensagem do processador *x* e o *send(x, valor)* irá transmitir o valor para o processador destino *x*.

No exemplo da Figura 1.17, os processadores 2 e 3 enviam os valores 10 e 20 para o processador 1. O processador 1 recebe os valores, soma e imprime.

Para ilustrar exemplos mais elaborados, a demonstração inclui dois códigos adicionais também gerados pelas LLMs para o modelo de troca de mensagens. O primeiro exemplo é um algoritmo distribuído de ordenação. O segundo exemplo é uma implementação paralela do algoritmo

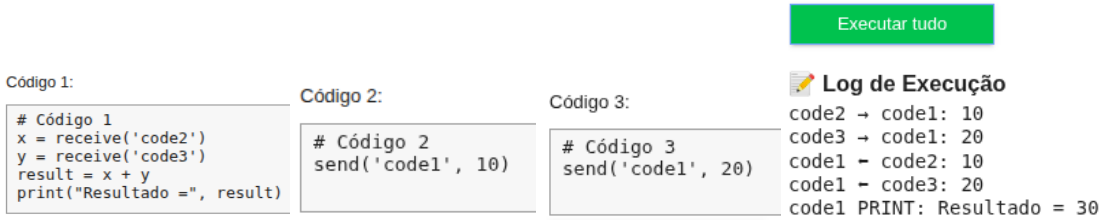

Figura 1.17. Simulador com três multiprocessadores por troca de mensagens e suas janelas de código, usando a linguagem Python. Clique aqui.

KNN de aprendizado de máquina, onde os processadores procuram os vizinhos mais próximos na sua partição de dados e enviam para o processador 1, que finaliza a execução.

1.4. Programação de Alto Desempenho com GPU

Nesta seção, iremos ilustrar como podemos usar as LLMs aliadas às facilidades do Google Colab para ensino e pesquisa usando GPUs. Inicialmente, iremos explorar o exemplo clássico de soma de vetores, onde $C_i = A_i + B_i$. Diferente dos *array processors*, em que o paralelismo está associado à distribuição das tarefas associadas ao número do processador, as GPUs usam um modelo mais abstrato, onde podemos associar o número de threads. Podemos ter milhares ou até bilhões de threads sendo disparadas. Cada thread pode executar uma tarefa simples ou complexa.

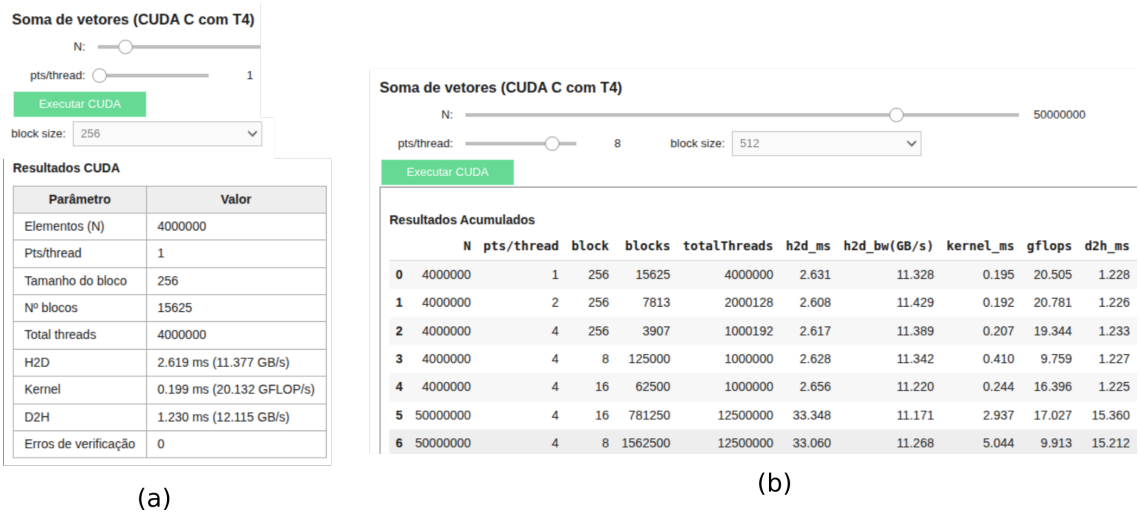

Figura 1.18. Execução em CUDA do exemplo Soma de Vetores variando bloco, tamanho do vetor, número de pontos por thread e fazendo medidas usando a linguagem Python: (a) Versão Inicial; (b) Versão com Rastreio. Clique aqui.

Nosso exemplo inicial usa a LLM ChatGPT para gerar um código CUDA com *ipywidgets* do Python para o exemplo de soma de vetores. A Figura 1.18(a) ilustra a interface, onde o usuário pode ajustar o tamanho do vetor N com um botão deslizante, o número de elementos que cada thread irá somar e o número de threads de cada bloco. Tendo estes dados, o código será gerado calculando quantos blocos e threads serão necessários para somar os vetores. A saída mostra em uma tabela formatada em HTML os parâmetros de entrada, os tempos de execução do kernel e das transferências de dados entre a CPU (Host) e a GPU (Device) com H2D (Host to Device) e D2H (Device to Host).

Neste exemplo, podemos observar que o desempenho é dominado pelo tempo de transferência de dados. O segundo exemplo, ilustrado na Figura 1.18(b), solicitamos à LLM para estender o código para armazenar em uma tabela o resumo de várias execuções. Assim, podemos explorar o tamanho do bloco e visualizar, por exemplo, que blocos com 64 threads ou menos irão ter um desempenho pior por subutilizar a GPU. Podemos aumentar e diminuir o tamanho do vetor e ver como escala o tempo de execução com o tamanho do vetor.

Apesar de simples, este *template* de código pode ser substituído por códigos mais elaborados, onde o programador pode facilmente explorar e documentar o espaço de soluções.

Para avaliar a geração de código para GPU da LLM DeepSeek, solicitamos o mesmo exemplo da soma de vetores. Foram gerados três tipos de código. O primeiro na linguagem C com CUDA. A célula de código inclui o exemplo completo do código com a parte da CPU e da GPU. No final da célula, foram adicionados os comandos para compilar e executar.

DeepSeek com três implementações em GPU

CUDA

```
cuda_code = '''
// Kernel CUDA para soma de vetores
__global__ void vectorAdd(
const float*A, const float*B,
float*C,int numElements) {
 int i = blockDim.x * blockIdx.x
    + threadIdx.x;
 if (i < numElements) {
   C[i] = A[i] + B[i];
  }
}
....
    cudaFree(d_C);
    return 0;
}
'''
with open('vector_add.cu',
     'w') as f:
  f.write(cuda_code)


!nvcc -arch=sm_75 -o vector_add
  vector_add.cu

!nvprof ./vector_add
```

PyCUDA e PyTorch

```
# PyCUDA
kernel_code = """
__global__ void vectorAdd
    (float *A, float *B,
       float *C, int n) {
    ...
}
"""
vector_add =
  mod.get_function("vectorAdd")
A_gpu = cuda.mem_alloc(A.nbytes)
cuda.memcpy_htod(A_gpu, A)
...
vector_add(A_gpu,B_gpu,C_gpu,
np.int32(n),block=(block_size,
1,1),grid=(grid_size,1))
expected = A + B
# PyTorch
...
A=torch.randn(n,device=device)
B=torch.randn(n,device=device)
...
C = A + B # soma na GPU
...
expected = A.cpu() + B.cpu()
```

Além do código em CUDA, a DeepSeek gerou a opção de usar *PyCUDA*, que mescla Python e CUDA. A primeira vantagem é que o trecho da CPU fica bem mais fácil para programar e introduzir a interface com a GPU, pois é um código Python. A segunda vantagem é que o código CUDA permanece em C, que é mais explícito que um código Python para usar com eficiência os recursos da GPU.

O código gerado ilustra que, usando *NumPy*, podemos inicializar facilmente o vetor, alocar memória com *cuda.mem_alloc*, transferir para GPU com *cuda.memcpy_htod*, depois chamar o

kernel CUDA. Para conferir, usamos a simples soma $A + B$ dos vetores *NumPy*. Para medir o tempo, usamos o método *time* da CPU. Podemos observar que a GPU é bem mais rápida se não consideramos o tempo de transferência entre a CPU e GPU.

Por fim, a DeepSeek também gerou um código com *PyTorch*, que já tem o método de soma de vetores e manipulação de vetores e matrizes na GPU. Primeiro são criados dois tensores para A e B com *torch.randn(n, device=device)* na GPU, depois é só somar com $C = A + B$. Para validar com a CPU, é necessário usar $A.cpu() + B.cpu()$.

O principal fator que pode gerar alto desempenho na GPU é a intensidade aritmética do código. A intensidade é calculada pela razão do número de operações aritméticas pelo número de operações com a memória. Um exemplo simples é medir a capacidade de cálculo de uma GPU. Suponha uma GPU hipotética com 10.000 núcleos e frequência de relógio de 1 GHz. Esta GPU tem o potencial de executar 10 Tera operações por segundo, pois 10.000 $\times$ 1 Giga/s = 10 Tera/s. Se os dados são floats de 32 bits, então para fazer $a + b$ precisamos ler 4+4 = 8 bytes. Ou seja, precisamos de uma vazão de memória de 80 Tera Bytes por segundo. Entretanto, a GPU só possui uma vazão próxima de 1 Tera Byte por segundo. Ou seja, a soma de vetores, que tem 2 operações de leitura e uma operação aritmética apenas, terá seu desempenho limitado pela vazão de leitura da memória global da GPU. Mesmo sendo mais rápida na leitura de memória que a CPU, a GPU estará sendo subutilizada.

Equação / Polinômio	**Tempo (ms)**	**Intensidade**	**GFLOPs**
$C[i] = A_i + B_i$	1,731	$1/2$	19,3
$C[i] = A_i^2 + B_i^2$	1,738	$3/2$	58,2
$C[i] = 3A_i + 5A_i^2 + 7A_iB_i + 9B_i + 12B_i^2$	1,762	$12/2$	228,8
Polinômio com 10 termos	1,816	$33/2$	615,2
Polinômio com 20 termos	1,899	$54/2$	954,2
Polinômio com 30 termos	2,595	$84/2$	1.088,3

Tabela 1.3. Comparação entre diferentes expressões e seus desempenhos computacionais. GFLOPs calculado como intensidade aritmética para um vetor com 32 Mega elementos.

Para exemplificar como podemos aumentar o desempenho da GPU, iremos usar polinômios com os vetores A e B. Ao usar um polinômio com $A_i^2 + B_i^2$, teremos 2 operações de leitura e 3 operações aritméticas, o que aumenta a intensidade aritmética de 0,5 para 1,5. A Tabela 1.3 mostra 6 exemplos de polinômios com várias intensidades aritméticas.

O primeiro exemplo é a soma de vetor simples, que tem a intensidade menor que 1 e está limitada pela memória, tendo o desempenho de 19,3 GFLOPs na soma de um vetor de 32 Mega elementos. Ou seja, bem abaixo considerando que está usando uma GPU T4 do Google Colab, que tem o potencial de 8 Tera FLOPs. O segundo exemplo é a soma dos quadrados, que aumenta a intensidade para 1,5, mantém o mesmo tempo de execução, pois o gargalo era a memória, tendo um ganho de 3$\times$ no desempenho e executa 58,2 GFLOPs/s. O terceiro exemplo é um polinômio com 5 termos e 12 operações, resultando em uma intensidade de 6, não altera o tempo de execução e sobe o desempenho para 228,8 GFLOPs/s.

Para o quarto exemplo, aumentamos ainda mais o volume de cálculo com o polinômio com 10 termos $P = C[i] = 1,5a + 2,3a^2 + 4,1a^3 + 1,7b + 3,2b^2 + 5,8b^3 + 2,9ab + 6,4a^2b + 3,7ab^2 + 8,2a^2b^2$ e 33 operações, resultando em um desempenho de 615,2 GFLOPs/s sem praticamente alterar o tempo de execução.

O quinto exemplo é um polinômio com 20 termos $P = C[i] = 1,1a + 2,2a^2 + 3,3a^3 + 4,4a^4 + 1,5b + 2,6b^2 + 3,7b^3 + 4,8b^4 + 5,1ab + 6,2a^2b + 7,3a^3b + 5,4ab^2 + 6,5ab^3 + 7,6a^2b^2 + 8,7a^3b^2 + 9,8a^2b^3 + 10,9a^4b + 11,1a^4b^2 + 12,2a^3b^3$ e 54 operações, que também quase não altera o tempo de execução, mas aumenta o desempenho para o patamar de 954,2 GFLOPs/s.

Finalmente, o último exemplo já altera o tempo de execução, mostrando que já existe limitação no paralelismo do cálculo e uso dos núcleos da GPU, com um polinômio com 30 termos $C[i] = 1,1a + 2,2a^2 + 3,3a^3 + 4,4a^4 + 5,5a^5 + 1,6b + 2,7b^2 + 3,8b^3 + 4,9b^4 + 5,1b^5 + 6,2ab + 7,3a^2b + 8,4a^3b + 9,5a^4b + 6,6ab^2 + 7,7ab^3 + 8,8ab^4 + 9,9a^2b^2 + 10,1a^3b^2 + 11,2a^4b^2 + 12,3a^2b^3 + 13,4a^3b^3 + 14,5a^4b^3 + 15,6a^5b + 16,7a^5b^2 + 17,8a^5b^3 + 18,9ab^5 + 19,1a^2b^5 + 20,2a^3b^4$ e 84 operações, alcança mais de 1 Tera, mais precisamente 1,09 TFLOPs/s.

Existem vários detalhes que devem ser observados para otimizar ainda mais. Um deles é o código assembly PTX da GPU para visualizar como o compilador está atuando. O último exemplo desta seção, explorando as LLMs, é a visualização do código PTX. Iremos usar o mesmo exemplo dos polinômios. O compilador nvcc da NVIDIA exporta o assembly PTX. Apesar de ser um código intermediário, já fornece informações, porém é de difícil leitura para iniciantes. Nosso experimento faz a extração do código PTX, depois isolamos a parte do cálculo do polinômio que começa com uma instrução de *load.global* para os dois elementos de A e B.

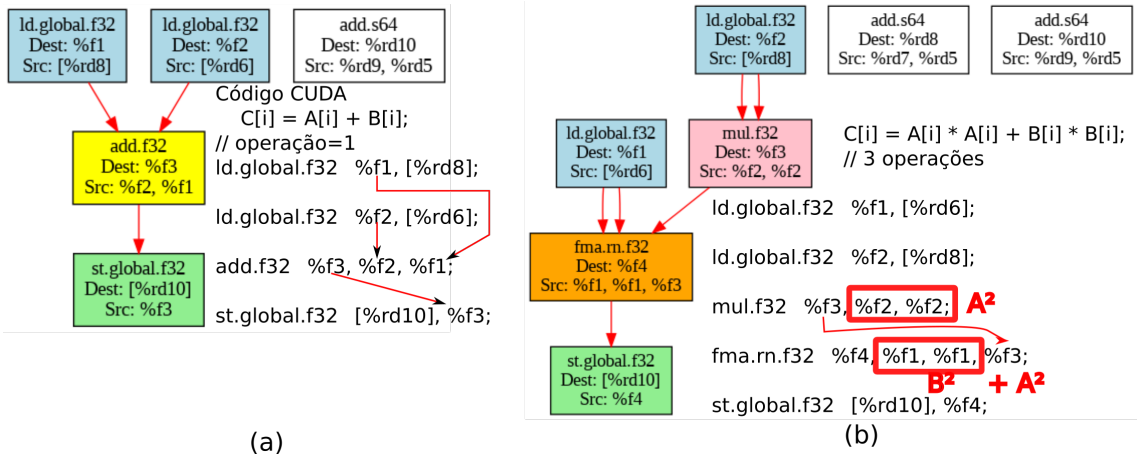

Figura 1.19. Grafo extraído do Assembly PTX para dois kernels simples em CUDA: (a) $C_i = A_i + B_i$; (b) $C_i = A_i^2 + B_i^2$. Clique aqui.

A Figura 1.19(a) mostra o trecho PTX para a soma simples de vetores com os registradores f_1 e f_2, que recebem os valores da memória global para os elementos dos vetores A e B, depois a instrução *add.f32* faz a soma e grava em f_3, que é gravado na memória pelo *store.global* no vetor C. Além do código PTX (apenas o trecho do cálculo), mostramos o grafo que foi gerado com auxílio da LLM, que processou o PTX, localizou o trecho, e a partir do código e da dependência entre as instruções gerou o grafo no formato DOT com a biblioteca Graphviz, que posteriormente foi transformado em uma figura no formato de imagem PNG.

A Figura 1.19(b) mostra o trecho PTX para a soma de quadrados dos vetores com os registradores f_2 e f_1, que recebem os valores da memória global para os elementos dos vetores A e B, depois a instrução *mul.f32* faz o quadrado de A, a instrução *fma*, que é multiplica e soma,

fará $b \times b + a^2 = f_1 \times f_1 + f_3$. Finalmente, o valor final em f_4 é gravado na memória pelo *store.global* no vetor C. São duas instruções, mas temos 3 operações, pois *fma* faz a multiplicação e soma.

Para o terceiro exemplo, onde temos 12 operações e o polinômio $C[i] = 3 \cdot a + 5 \cdot a^2 + 7 \cdot a \cdot b + 9 \cdot b + 12 \cdot b^2$, ilustramos o trecho de código PTX gerado e também o grafo extraído com auxílio da LLM DeepSeek, que gera o grafo de dependência de operações. Podemos observar os dois *loads* em azul e o *store* em verde, quatro operações de multiplicação em rosa e quatro operações *fma* em laranja, que realizam duas operações cada (multiplica e soma), totalizando 12 operações.

Assembly PTX

Polinômio com 5 termos

```
ld.global.f32    %f1, [%rd6];
mul.f32          %f2, %f1, 0f40A00000;
mul.f32          %f3, %f1, %f2;
fma.rn.f32       %f4, %f1, 0f40400000, %f3;
mul.f32          %f5, %f1, 0f40E00000;
ld.global.f32    %f6, [%rd8];
fma.rn.f32       %f7, %f5, %f6, %f4;
fma.rn.f32       %f8, %f6, 0f41100000, %f7;
mul.f32          %f9, %f6, 0f41400000;
fma.rn.f32       %f10, %f6, %f9, %f8;
st.global.f32    [%rd10], %f10;
```

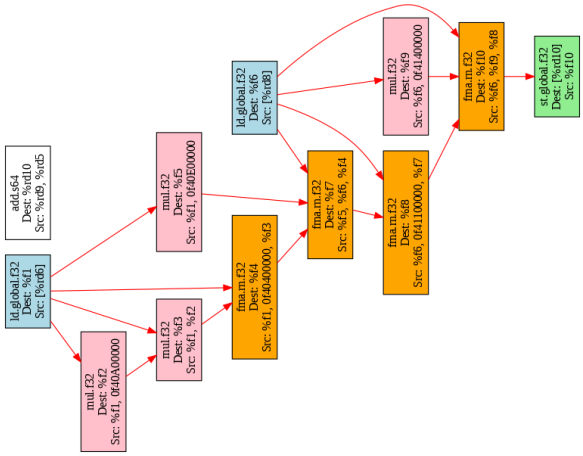

Figura 1.20. Grafo extraído do Assembly PTX para kernel simples em CUDA: $C[i] = 3a + 5a^2 + 7ab + 9b + 12b^2$**. Clique aqui.**

Para validar a extração do PTX e visualização do grafo, testamos também para o polinômio com 10 termos $P = C[i] = 1,5a + 2,3a^2 + 4,1a^3 + 1,7b + 3,2b^2 + 5,8b^3 + 2,9ab + 6,4a^2b + 3,7ab^2 + 8,2a^2b^2$, que tem 33 operações, onde teremos o grafo ilustrado na Figura 1.21 com 15 multiplicações, que são os vértices em rosa, e 9 vértices de multiplica/soma (fma), que geram 18 operações, totalizando $15 + 18 = 33$ operações.

1.5. Aprendizado de Máquina

Atualmente, com a popularização da inteligência artificial, além da geração de texto e códigos com as LLMs, podemos explorar as técnicas clássicas de aprendizado de máquina na pesquisa

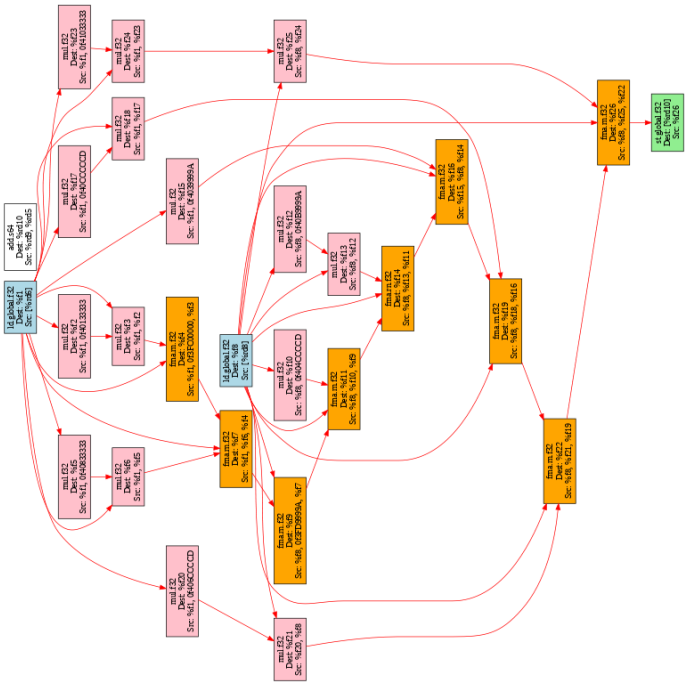

Figura 1.21. Grafo extraído do Assembly PTX para o kernel em CUDA do polinômio com 33 operações. Clique aqui.

e análise de dados.

Como as bibliotecas Python para processamento de dados tabulares (Pandas), visualização (Matplotlib) e algoritmos básicos de aprendizado de máquina (scikit-learn) estão bem maduras, são robustas e têm um grande volume de código disponível, as LLMs são extremamente eficientes na geração de código. Nesta seção, iremos usar conjuntos de dados (*datasets*) clássicos para exemplificar várias situações que podem ser adaptadas para uso em experimentos de pesquisa. Faremos dois exemplos básicos.

O primeiro conjunto de exemplos começa ilustrando vários mecanismos para leitura dos dados no Google Colab. Podemos usar o comando *wget* para buscar em uma URL da internet, podemos usar um arquivo na sua conta *Google Drive* ou podemos fazer upload de um arquivo do seu computador. Existe também um recurso bem interessante, que é um link dinâmico: ao gerar uma planilha em sua pasta do *Google Drive* com permissão de leitura, como o link nunca é alterado pela Google, podemos atualizar a planilha com novos dados sendo coletados e o processamento não precisa ser modificado.

Por exemplo, um formulário do Google pode gerar os dados e ir sendo atualizado do lado da coleta de dados e, do lado do processamento, o Google Colab pode ser executado novamente, sem modificações, para atualizar a análise. Este recurso é explorado com o comando *gdown*.

O primeiro conjunto de exemplos também ilustra a exploração básica da leitura de uma planilha genérica, estatísticas básicas do número de amostras (linhas) e de colunas (atributos), separação dos dados numéricos e categóricos, visualização dos dados com histogramas e gráficos de barra ou pizza. Mostramos também como usar os ipywidgets para selecionar os atributos, tipo de gráficos, além de também gerar combinações de 2 atributos para visualização dos dados para fazer uma análise exploratória. Mostramos também como ver a dispersão dos dados em duas

dimensões.

Figura 1.22. Visualização com JavaScript dos dados do dataset de carros americanos com consulta e seleção. Clique aqui.

O último experimento do primeiro conjunto de exemplos, diferente dos anteriores que exploram a interface em Python com *widgets*, usa a LLM Claude e JavaScript. Mostramos o exemplo do dataset de vários modelos de carros, onde o usuário pode selecionar as marcas de carros, a faixa de preços. Os carros serão filtrados, algumas informações são exibidas com destaque, como a média de quilometragem, preço médio e faixa de ano de fabricação, além do boxplot da quilometragem em função do ano de fabricação.

O segundo conjunto de experimentos alia a análise exploratória de dados com o uso de ferramentas de aprendizado de máquina. Além disso, usamos uma metodologia de gerar códigos gradativos com as LLMs e de forma desacoplada.

Primeiro, iremos ler um conjunto de dados e trazer para o Colab. No exemplo, podemos informar uma URL de um site onde estarão os dados a serem buscados. As planilhas são gravadas em uma pasta local do Google Colab com o nome *datasets*. Assim, podemos buscar dados de diferentes fontes e trazer para nosso conjunto de experimentos.

A segunda parte irá buscar quais arquivos estão na pasta *datasets* e criar um botão dropdown para a escolha de um deles. O usuário pode então visualizar as informações básicas com número de amostras e atributos, e espaço em memória alocado para ler o conjunto de dados. A Figura 1.23(a) ilustra a interface gerada em Python pela LLM DeepSeek.

A terceira parte também é independente das anteriores. A segunda parte irá carregar o conjunto de dados na estrutura de *dataframe* do Pandas na variável df. Todo o processamento da terceira parte assume que os dados já estão em df. Observe que todas as partes estão desacopladas e são genéricas para serem aplicadas em qualquer conjunto de dados.

O objetivo da terceira parte é selecionar e explorar os modelos de aprendizado de máquina para

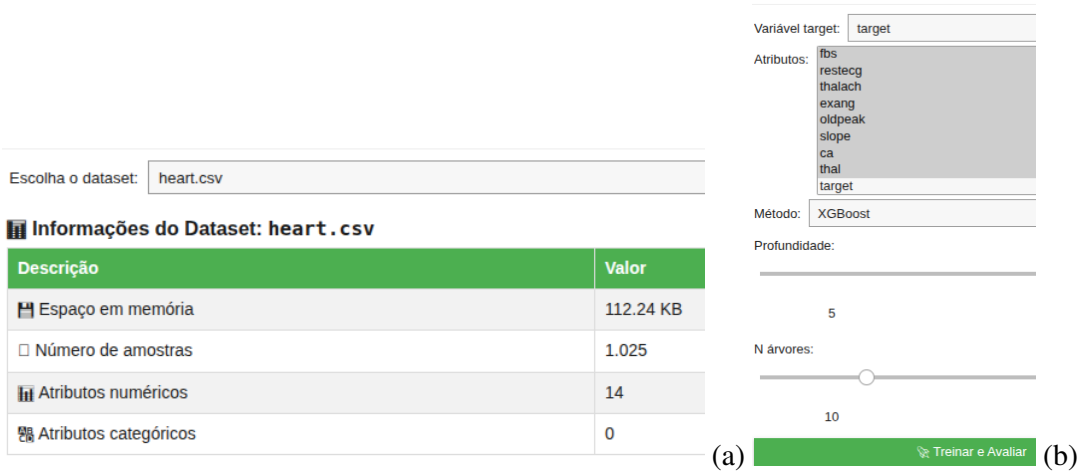

Figura 1.23. (a) Interface para seleção do dataset; (b) Interface para escolha do modelo de aprendizado de máquina. Clique aqui.

classificação. Portanto, o usuário deve escolher uma variável alvo. Os atributos são listados na interface, o usuário seleciona a variável alvo e pode também excluir alguns atributos. Depois, ele seleciona qual é a técnica de aprendizado de máquina que irá escolher. Dependendo da técnica, pode ajustar alguns parâmetros, como ilustrado na Figura 1.23(b). Uma vez selecionada, irá executar e mostrar a matriz de confusão, acurácia, precisão e outras métricas clássicas de aprendizado de máquina para classificação.

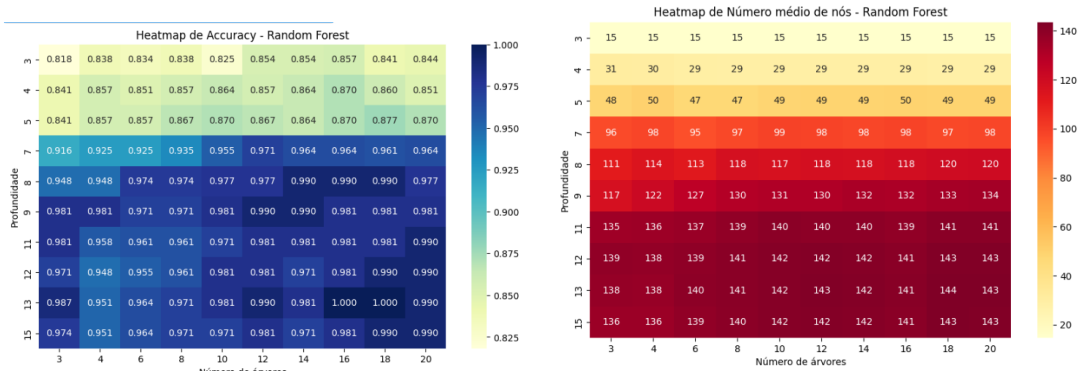

Figura 1.24. Exploração em Grade da Profundidade e número de árvores para XGBoost e Random Forest: (a) Mapa de Calor da Acurácia; (b) Mapa de Calor do Tamanho. Clique aqui.

Outro recurso interessante das LLMs é a exploração dos hiperparâmetros dos modelos. Podemos fazer a exploração e visualização com mapas de calor, por exemplo. A Figura 1.24 mostra a exploração para o conjunto de dados selecionado na terceira parte. O usuário pode escolher o modelo de Random Forest ou XGBoost. Depois, o usuário seleciona o número mínimo e máximo para a profundidade e para o número de árvores. No exemplo da Figura 1.24, podemos observar com as cores que, a partir de um certo valor de profundidade e/ou número de árvores, a acurácia satura e não vale a pena aumentar o gasto com mais vértices e árvores.

1.6. Considerações Finais

O uso de Modelos de Linguagem de Grande Escala (LLMs) como assistentes de desenvolvimento representa uma mudança de paradigma na forma como projetamos, implementamos e validamos ferramentas computacionais. Ao longo deste minicurso, evidenciamos que a eficácia desses modelos depende da formulação dos *prompts*. A capacidade de gerar código funcional, criar visualizações interativas e integrar múltiplas linguagens (Python, JavaScript, C++, CUDA) em um mesmo fluxo de trabalho amplia a produtividade de pesquisadores, docentes e estudantes.

Os exemplos apresentados demonstraram que as LLMs com *prompts* simples são capazes de auxiliar tanto na criação de protótipos rápidos quanto na elaboração de simuladores e interfaces complexas, como ilustrado nos casos de aprendizado de máquina, arquitetura de computadores e programação paralela.

Além do ganho em produtividade, o ambiente acessível do Google Colab aliado à expressividade das LLMs possibilita a criação de experiências de aprendizagem mais interativas, visuais e exploratórias.

1.7. Agradecimentos

Gostaríamos de agradecer a colaboração de todos os estudantes das disciplinas de Arquitetura de Computadores e Organização de Computadores da Universidade Federal de Viçosa. Apoio financeiro do Projeto FAPEMIG APQ-01577-22, CNPq e CAPES.

1.7.1. Disponibilidade de dados e materiais

As ferramentas desenvolvidas neste trabalho são de código aberto e estão disponíveis no link `https://colab.research.google.com/drive/10_rpBNXruJWlYrdLrPyflFmr6hdf1-RU?usp=sharing` ou Clique aqui.

1.7.2. Outras informações relevantes

O texto deste artigo é de responsabilidade dos autores, onde ferramentas de IA foram usadas apenas para revisão ortográfica e gramatical, além de algumas sugestões. O tema do trabalho é sobre o uso de IA, neste aspecto os modelos de IA foram avaliados para geração dos simuladores apresentados.

Referências

[Al-Shetairy et al. 2024] Al-Shetairy, M., Hindy, H., Khattab, D., and Aref, M. M. (2024). Transformers utilization in chart understanding: A review of recent advances & future trends. *arXiv preprint arXiv:2410.13883*.

[Almanasra and Suwais 2025] Almanasra, S. and Suwais, K. (2025). Analysis of chatgpt-generated codes across multiple programming languages. *IEEE Access*.

[Canesche et al. 2021] Canesche, M., Bragança, L., Neto, O. P. V., Nacif, J. A., and Ferreira, R. (2021). Google colab cad4u: Hands-on cloud laboratories for digital design. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

[Chen et al. 2023] Chen, B., Zhang, Z., Langrené, N., and Zhu, S. (2023). Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*.

[Coura et al. 2025] Coura, P., Freitas, I., Costa, H., Nacif, J., and Ferreira, R. (2025). Desmis-

tificando o ensino de inteligência artificial e aprendizado de máquina. In *Simpósio Brasileiro de Educação em Computação (EDUCOMP)*, pages 25–27. SBC.

[de Figueiredo et al. 2024] de Figueiredo, G. A., de Souza, E. S., Rodrigues, J. H., Nacif, J. A., and Ferreira, R. (2024). Desenvolvendo ferramentas para ensino de risc-v com python, verilog, matplotlib, svg e chatgpt. *International Journal of Computer Architecture Education*, 13(1):43–52.

[Elon University 2025] Elon University (2025). Survey: 52% of u.s. adults now use ai large language models like chatgpt. Elon University. Accessed: 28 de outubro de 2025.

[Ferreira et al. 2024] Ferreira, R., Canesche, M., Jamieson, P., Neto, O. P. V., and Nacif, J. A. (2024). Examples and tutorials on using google colab and gradio to create online interactive student-learning modules. *Computer Applications in Engineering Education*, page e22729.

[Ferreira and Nacif 2025] Ferreira, R. and Nacif, R. D. G. P. (2025). Desenvolvendo simuladores para arquitetura de computadores com auxílio de modelos generativos de linguagens. *International Journal of Computer Architecture Education*, 14.

[Godage et al. 2025] Godage, T., Nimishan, S., Vasanthapriyan, S., Palanisamy, V., Joseph, C., and Thuseethan, S. (2025). Evaluating the effectiveness of large language models in automated unit test generation. In *2025 5th International Conference on Advanced Research in Computing (ICARC)*, pages 1–6. IEEE.

[Jamieson et al. 2025] Jamieson, P., Ferreira, R., and Nacif, J. (2025). Board# 72: Leveraging large language models to create interactive online resources for digital systems and computer architecture education. In *2025 ASEE Annual Conference & Exposition*.

[Joel et al. 2024] Joel, S., Wu, J. J., and Fard, F. H. (2024). A survey on llm-based code generation for low-resource and domain-specific programming languages. *arXiv preprint arXiv:2410.03981*.

[Lisboa et al. 2025] Lisboa, M. O., Costa, H., Coura, P., Freitas, I., Villela, M. L. B., and Ferreira, R. (2025). Modelos generativos de linguagem na construção de ferramentas de ensino de computação com interface gráfica. In *Simpósio Brasileiro de Educação em Computação (EDUCOMP)*, pages 639–650. SBC.

[Pedregosa et al. 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830.

[Pérez and Granger 2007] Pérez, F. and Granger, B. E. (2007). Ipython: a system for interactive scientific computing. *Computing in science & engineering*, 9(3):21–29.

[Rule et al. 2019] Rule, A., Birmingham, A., Zuniga, C., Altintas, I., Huang, S.-C., Knight, R., Moshiri, N., Nguyen, M. H., Rosenthal, S. B., Pérez, F., and Rose, P. W. (2019). Ten simple rules for writing and sharing computational analyses in jupyter notebooks. *PLOS Computational Biology*, 15(7):1–8.

[Rule et al. 2018] Rule, A., Tabard, A., and Hollan, J. D. (2018). Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12.

[Ságodi et al. 2024] Ságodi, Z., Siket, I., and Ferenc, R. (2024). Methodology for code synthesis evaluation of llms presented by a case study of chatgpt and copilot. *Ieee Access*, 12:72303–72316.

[Vyas and BHARDWAJ 2025] Vyas, H. and BHARDWAJ, R. G. (2025). Chatgpt vs deepseek: A comparative evaluation on the international computer science benchmark–acm icpc.

[Zala et al. 2023] Zala, A., Lin, H., Cho, J., and Bansal, M. (2023). Diagrammergpt: Generating open-domain, open-platform diagrams via llm planning. *arXiv preprint arXiv:2310.12128*.