MINICURSOS

SSCAD 2025

Minicursos do XXVI Simpósio em Sistemas Computacionais de Alto Desempenho

28 a 31 de outubro, 2025 – Bonito, MS, Brasil

Organização

UFMS

UNICAMP

Promoção

SBC Sociedade Brasileira de Computação

IEEE COMPUTER SOCIETY

CE-ACPAD

Financiamento

CAPES

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico

Patrocínio

aws

DELL Technologies

NVIDIA

versatus HPC

NVIDIA

KAIAK

SCHERM HIGH PERFORMANCE COMPUTING

SUPERMICRO

HPE Partner

Parceria

H2O ecoturismo e eventos

XXVI SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO
DE 28 A 31 DE OUTUBRO DE 2025
BONITO – MS

MINICURSOS DO XXVI SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO

Organizadores
Daniel Cordeiro
Calebe P. Bianchini

Porto Alegre
Sociedade Brasileira de Computação — SBC
2025

Dados Internacionais de Catalogação na Publicação (CIP)

S612 Simpósio em Sistemas Computacionais de Alto Desempenho (26. : 28 – 31 outubro 2025 : Bonito)

Minicursos do SSCAD 2025 [recurso eletrônico]. Organização: Daniel Cordeiro, Calebe P. Bianchini – Dados eletrônicos. – Porto Alegre: Sociedade Brasileira de Computação, 2025.

82 p. : il. : PDF; 5 MB

Modo de acesso: World Wide Web.
Inclui bibliografia
ISBN 978-85-7669-660-5 (e-book)

1. Computação – Brasil – Evento. 2. Sistemas computacionais. 3. Alto desempenho. I. Cordeiro, Daniel. II. Bianchini, Calebe P.. III. Sociedade Brasileira de Computação. VI. Título.

CDU 004(063)

Ficha catalográfica elaborada por Annie Casali – CRB-10/2339

Biblioteca Digital da SBC – SBC OpenLib

Sociedade Brasileira de Computação
Av. Bento Gonçalves, 9500
Setor 4 | Prédio 43.412 | Sala 219 | Bairro Agronomia Caixa Postal 15012 | CEP 91501-970
Porto Alegre - RS
Fone: (51) 992526018
sbc@sbc.org.br

Índice

Prefácio

Esta edição do Livro de Minicursos do SSCAD reúne o material didático elaborado pelos autores dos três minicursos apresentados durante o XXVI Simpósio em Sistemas Computacionais de Alto Desempenho, realizado de 28 a 31 de outubro de 2025, em Bonito, MS.

O primeiro minicurso propõe uma abordagem prática para o uso de Modelos de Linguagem de Grande Escala (*Large Language Models*, LLMs) como ferramenta de apoio à produção de material didático e à pesquisa nas áreas de Arquitetura de Computadores e Computação de Alto Desempenho. O capítulo apresenta conceitos fundamentais de engenharia de *prompt*, detalha as funcionalidades dos principais softwares de notebooks interativos — Jupyter Notebook e Google Colab — utilizados para o desenvolvimento e execução de código, e traz exemplos interativos de sistemas construídos com LLMs. Os exemplos exploram temas como ciência de dados, aprendizado de máquina, computação de alto desempenho, além de abordar arquiteturas de computadores avançadas, dedicadas e específicas, métodos de avaliação, medição e predição de desempenho.

O segundo minicurso oferece um tutorial detalhado sobre a adição de novas instruções ao *backend* RISC-V da infraestrutura do compilador LLVM. O tutorial descreve o passo a passo de um projeto de uma pequena extensão chamada “xmatrix”, que introduz 32 registradores matriciais dedicados de 512 bits (matrizes de 4×4 elementos de 32 bits) e um conjunto restrito de operações aritméticas e de carga/armazenamento.

Por fim, o terceiro minicurso aborda os mecanismos do Open MPI 5.0 — uma das APIs mais empregadas no desenvolvimento de aplicações paralelas em ambientes de alto desempenho — que simplificam as atividades de depuração e ajustes finos (*tuning*). O conteúdo inicia-se com uma breve introdução à MPI (*Message Passing Interface*), seguida de uma explanação detalhada da *Modular Component Architecture* (MCA), a qual concede ao desenvolvedor um conjunto extensivo de opções para ajuste e depuração de programas MPI. O capítulo termina com a exposição dos aspectos práticos do processo de *tuning*, incluindo a aplicação das ferramentas Valgrind e Memchecker para a depuração de aplicações paralelas.

Acreditamos que este livro permitirá que estudantes, pesquisadores, profissionais e entusiastas das áreas de Arquitetura de Computadores e Processamento de Alto Desempenho tenham acesso consistente e aprofundado ao conhecimento apresentado no SSCAD 2025, oferecendo um recurso sólido, claro e duradouro para o aprofundamento dos estudos, mesmo àqueles que não puderam participar do evento presencialmente.

Desejamos a todos uma excelente leitura.

Daniel Cordeiro (USP) e Calebe P. Bianchini (Mackenzie)
Coordenadores dos Minicursos do SSCAD 2025

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.

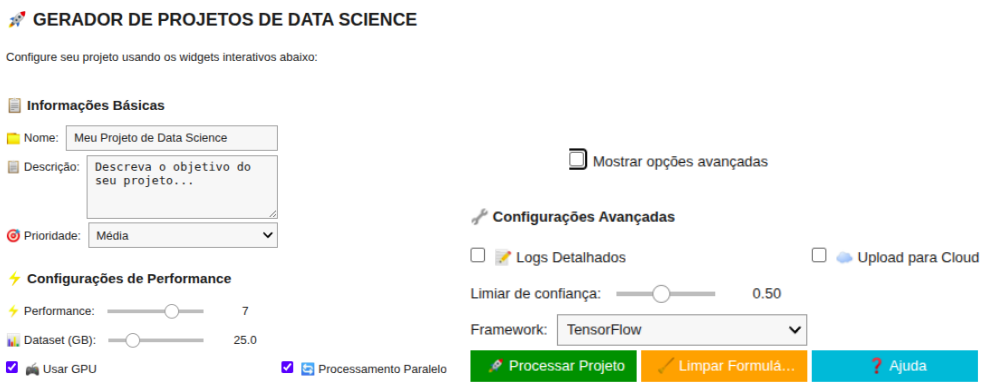

Figura 1.1. Exemplo de interface com vários botões e caixas de texto interativas da biblioteca ipywidgets. Para acesso ao exemplo clique aqui.

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')
```

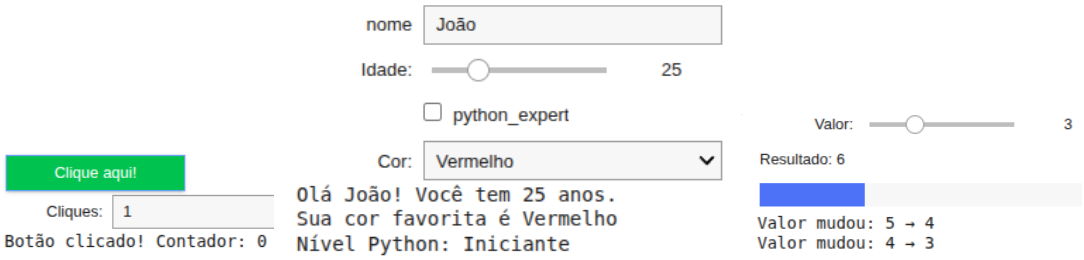

Figura 1.2. Interfaces: *on_click*, *interact*, *observe*. **Para acesso ao exemplo clique aqui.**

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).

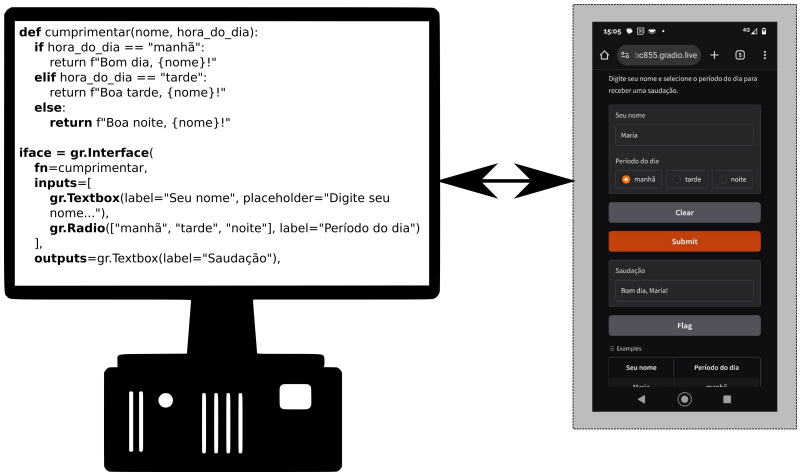

Figura 1.3. Acesso remoto com dispositivo móvel ao Google Colab via Gradio e Hugging Face. Para acesso ao exemplo clique aqui.

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.

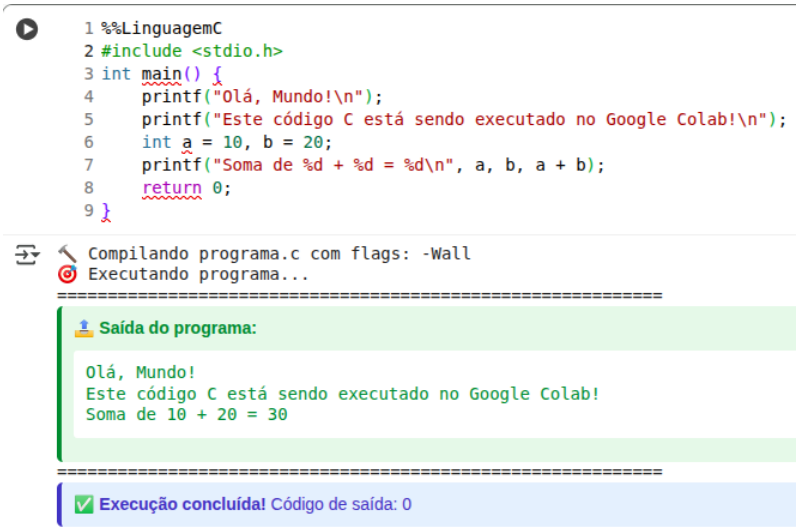

Figura 1.4. Criação de um comando mágico para compilar e executar um código C. Para acesso ao exemplo clique aqui.

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.

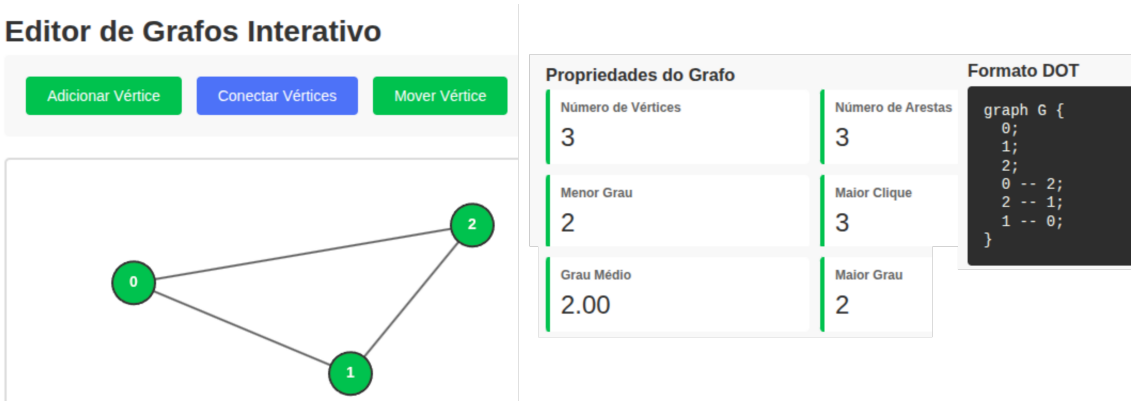

Figura 1.5. Editor de Grafos para exemplos com JavaScript usando a LLM Claude. Para acesso ao exemplo clique aqui.

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).

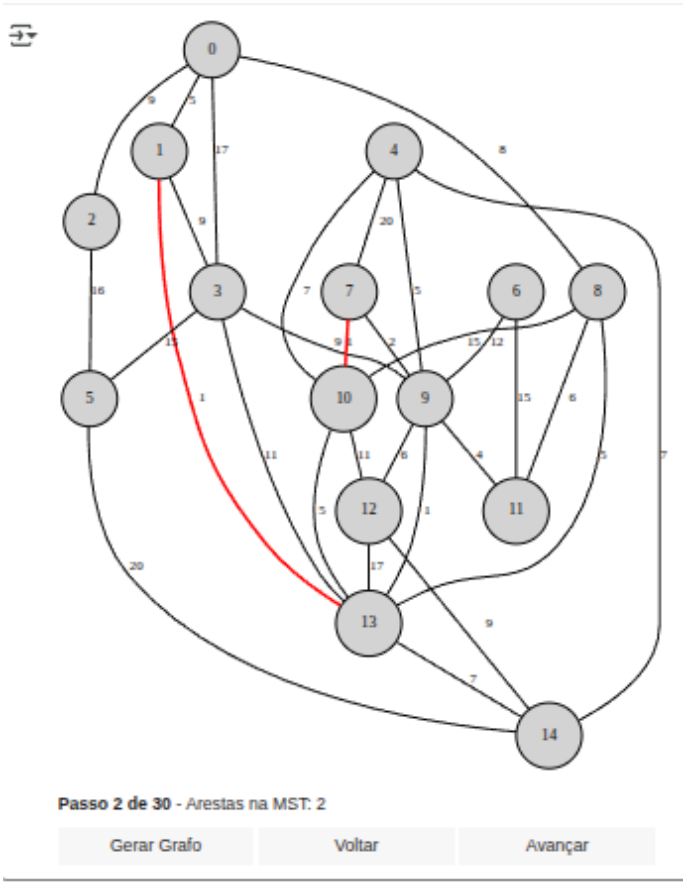

Figura 1.6. Árvore geradora mínima com Chatgpt e Deepseek usando graphviz e networkX. Para acesso ao exemplo Clique aqui

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.

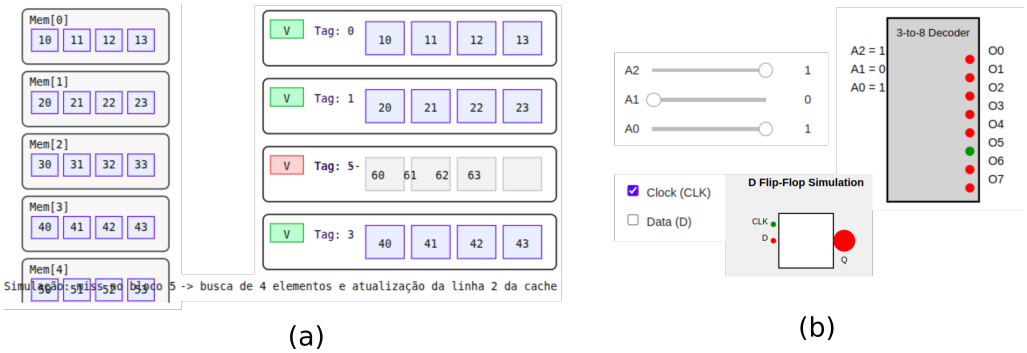

Figura 1.7. (a) Animação de Cache. (b) Decodificador e Flip-flop D extraídos de (Jamieson et al. 2025). Para acesso ao exemplo clique aqui.

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.

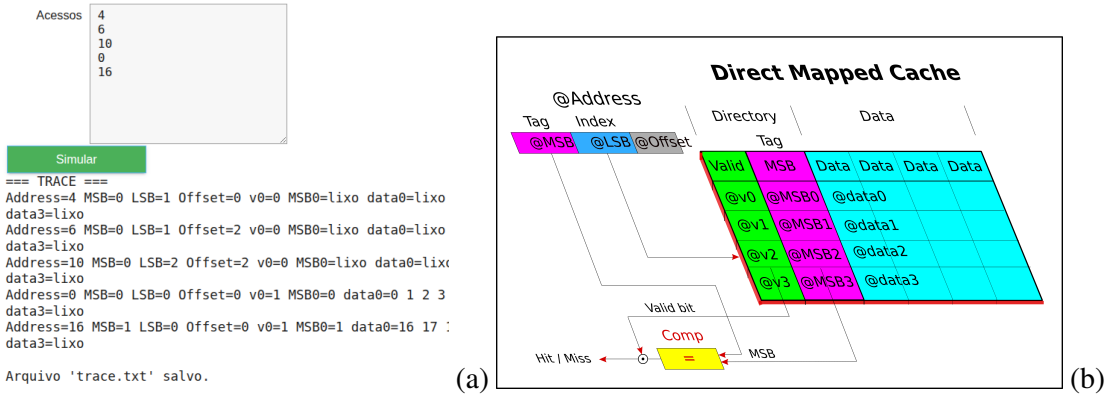

Figura 1.8. (a) Entrada da sequência de endereços para simulação e trecho do traçado produzido. (b) Figura SVG com prefixo "@" nos rótulos que serão substituídos pelos resultados da simulação. Para acesso ao exemplo clique aqui.

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.

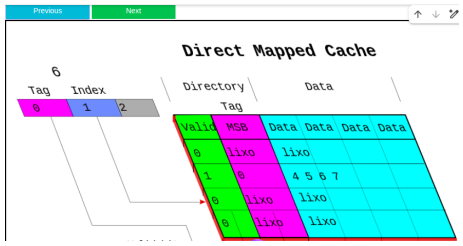

Figura 1.9. Resultado da animação da simulação do traçado da cache.

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

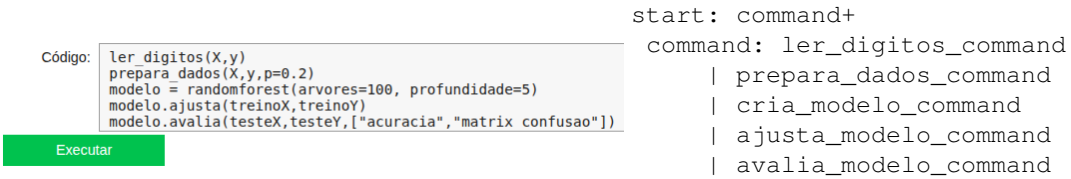

Figura 1.12. Exemplo de Linguagem de Domínio Específico com sua gramática correspondente com um trecho da gramática. Acesso ao exemplo clique aqui.

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).

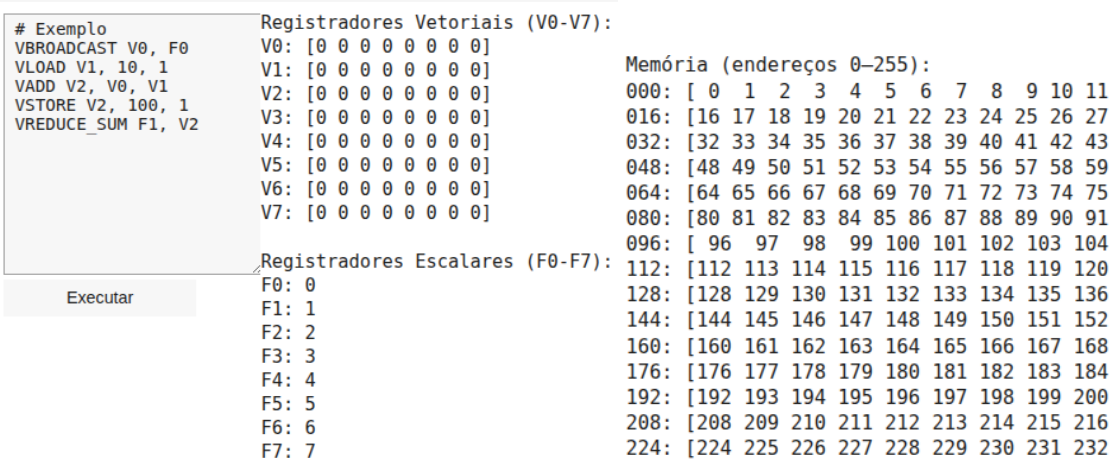

Figura 1.13. Simulador com uma linguagem vetorial, incluindo a janela de edição e botão para executar o código com a visualização dos registradores e da memória. Acesso ao exemplo clique aqui.

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

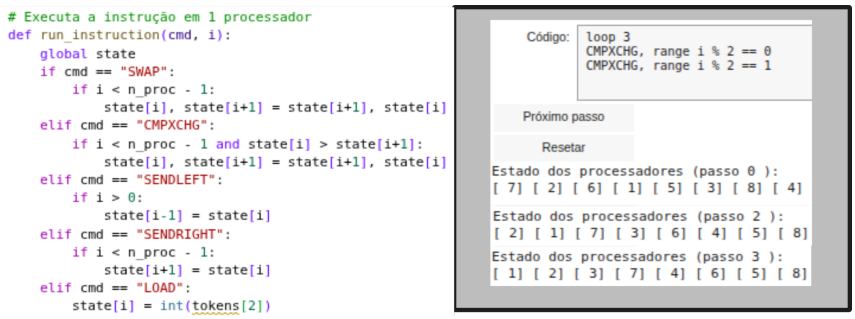

Figura 1.14. Simulador com uma linguagem Array Processor SIMD, incluindo a janela de edição e botão para executar o código com a visualização do array de processadores com o exemplo do algoritmo de ordenação par/ímpar. Clique aqui.

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.

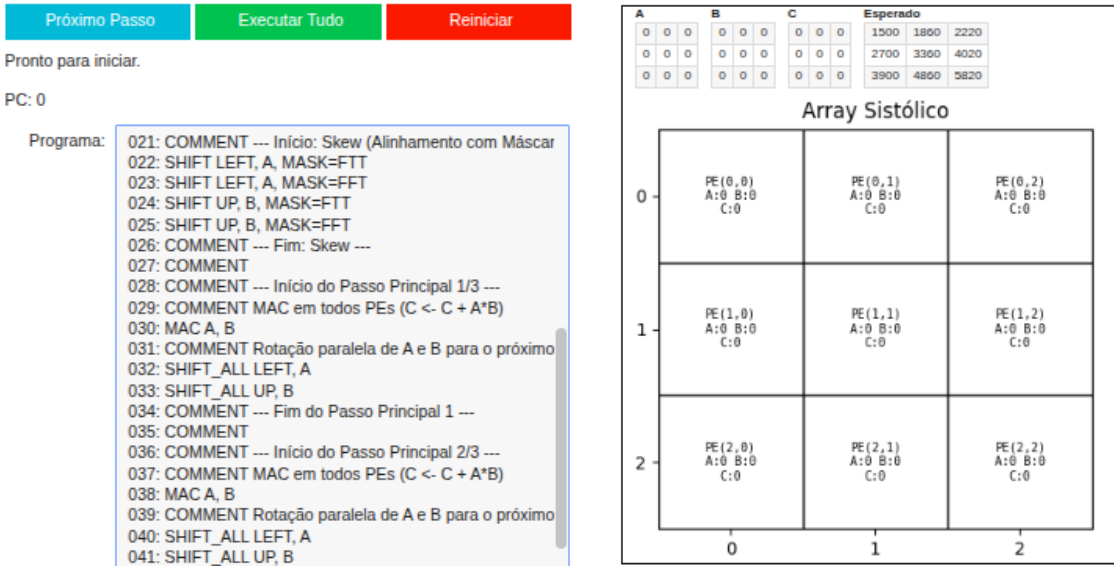

Figura 1.15. Simulador com uma linguagem Array Processor SIMD, incluindo a janela de edição e botão para executar o código com a visualização dos elementos de processamento (PE). Clique aqui.

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.

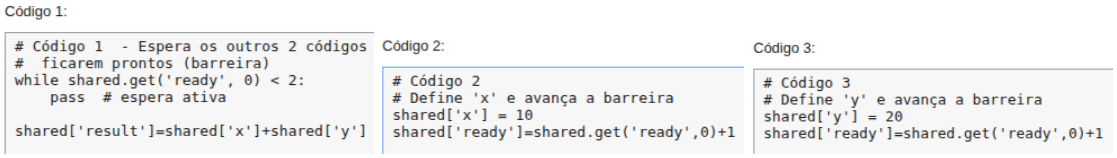

Figura 1.16. Simulador com três multiprocessadores e suas janelas de código, usando a linguagem Python e sincronismo com barreira através de memória compartilhada. Clique aqui.

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

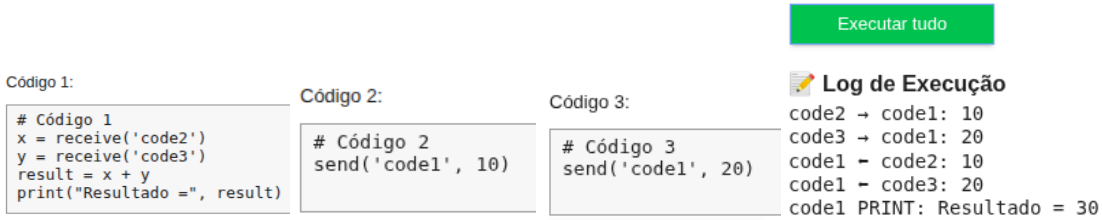

Figura 1.17. Simulador com três multiprocessadores por troca de mensagens e suas janelas de código, usando a linguagem Python. Clique aqui.

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.

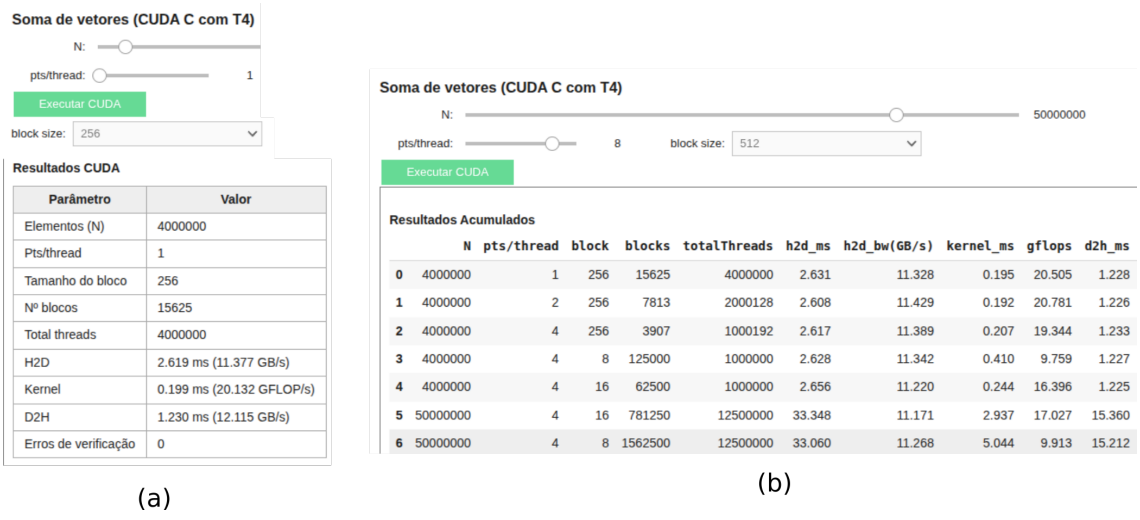

Figura 1.18. Execução em CUDA do exemplo Soma de Vetores variando bloco, tamanho do vetor, número de pontos por thread e fazendo medidas usando a linguagem Python: (a) Versão Inicial; (b) Versão com Rastreio. Clique aqui.

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.

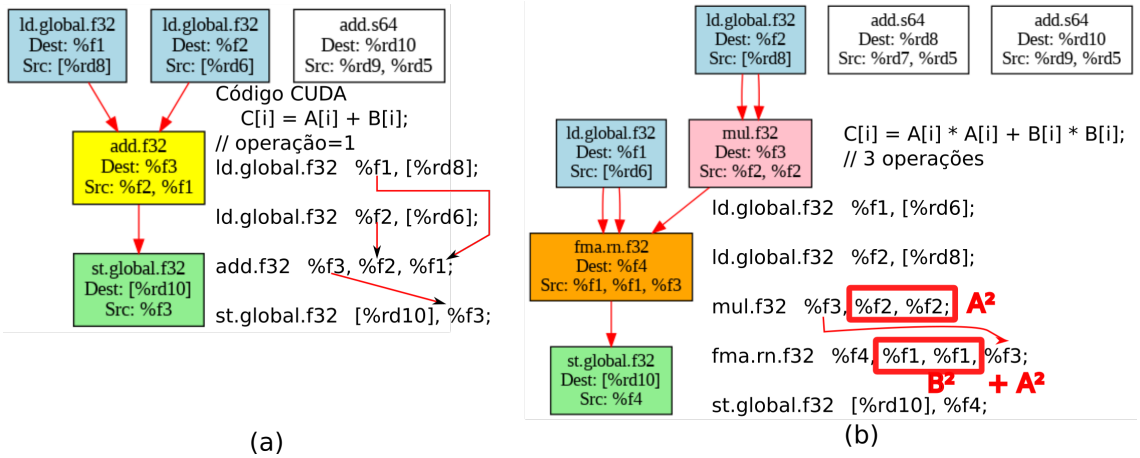

Figura 1.19. Grafo extraído do Assembly PTX para dois kernels simples em CUDA: (a) $C_i = A_i + B_i$; (b) $C_i = A_i^2 + B_i^2$. Clique aqui.

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;
```

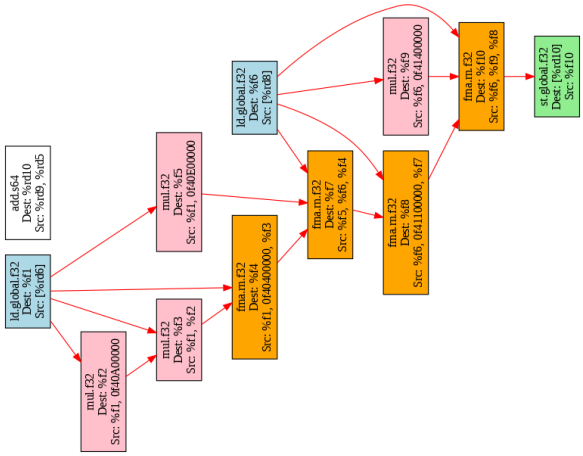

Figura 1.20. Grafo extraído do Assembly PTX para kernel simples em CUDA: $C[i] = 3a + 5a^2 + 7ab + 9b + 12b^2$**. Clique aqui.**

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

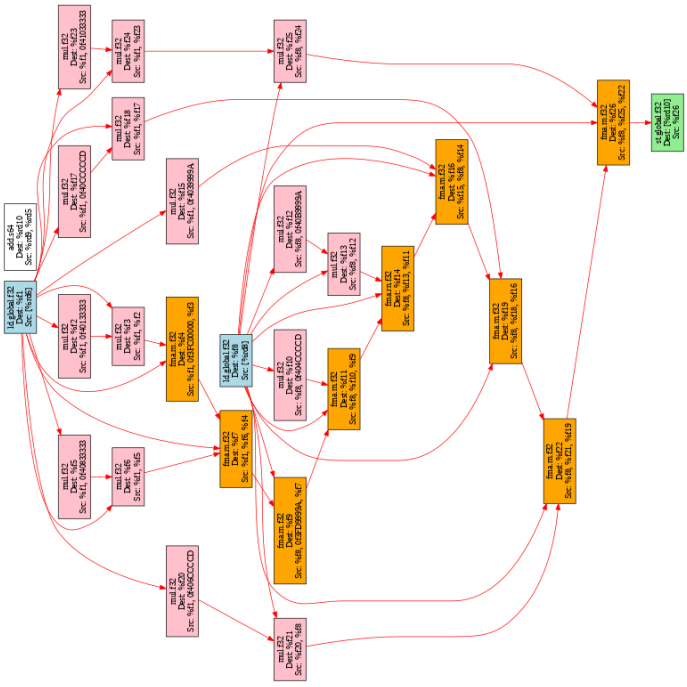

Figura 1.21. Grafo extraído do Assembly PTX para o kernel em CUDA do polinômio com 33 operações. Clique aqui.

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

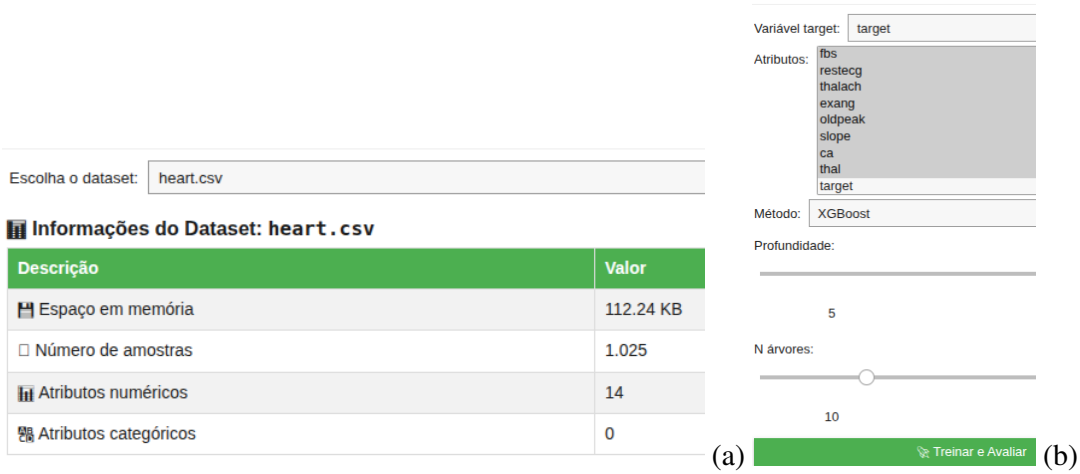

Figura 1.23. (a) Interface para seleção do dataset; (b) Interface para escolha do modelo de aprendizado de máquina. Clique aqui.

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.

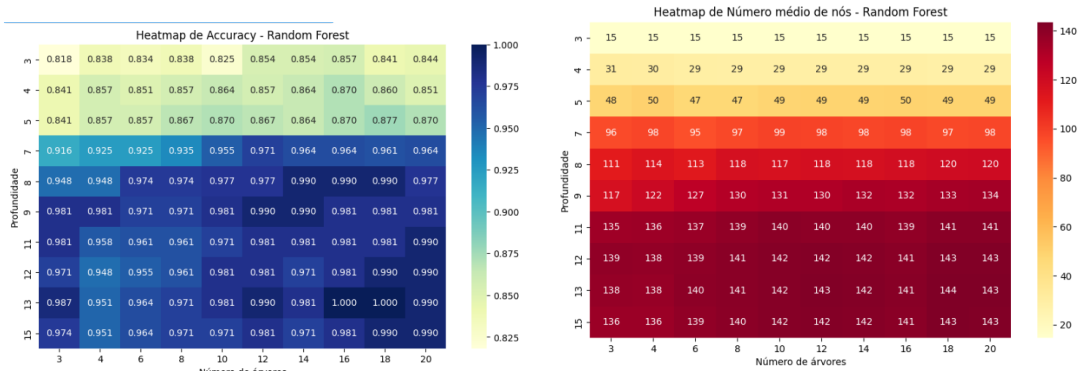

Figura 1.24. Exploração em Grade da Profundidade e número de árvores para XGBoost e Random Forest: (a) Mapa de Calor da Acurácia; (b) Mapa de Calor do Tamanho. Clique aqui.

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*.

Chapter 2

Implementing new RISC-V Instructions with the LLVM Compiler Infrastructure

Gustavo Leite, Instituto de Computação, Unicamp (Email) (Lattes)
Carlos E. C. Barbosa, Instituto de Computação, Unicamp (Email)
Hervé Yviquel, Instituto de Computação, Unicamp (Email) (Lattes)
Sandro Rigo, Instituto de Computação, Unicamp (Email) (Lattes)

Abstract

RISC-V is an open and modular architecture standard whose design makes it a fertile ground for innovation in computer architecture. Its extensible ISA enables the introduction of domain-specific instructions, tailored for applications ranging from embedded systems and IoT to high-performance computing (HPC) and artificial intelligence (AI). As new extensions appear, such as those for vector and matrix operations, compiler support becomes a key enabler for their practical adoption. In this context, understanding how to extend a compiler backend is an essential skill for researchers and developers working on custom accelerators.

This minicourse presents a step-by-step introduction to adding new instructions to the RISC-V backend of the LLVM compiler infrastructure. Participants will learn the complete process of extending LLVM to support a custom matrix processing extension, including the definition of new instruction formats, encoding schemes, and register sets using the TableGen declarative language. The tutorial walks through the design of a small "xmatrix" extension that introduces 32 dedicated 512-bit matrix registers and a small set of load/store and arithmetic operations for 4×4 tiles. Through practical examples, attendees will see how to integrate these instructions into LLVM, assemble and disassemble them, and prepare the compiler for future integration with simulators, custom-chips and higher-level languages such as C/C++.

2.1. Introduction

Achieving maximum efficiency in high-performance computing requires a deep understanding of multiple layers of the system. Beyond the application itself, programmers must optimize for different levels of parallelism. For applications targeting machine clusters, this often involves understanding the cluster network's topology, link latency, and throughput. Within each node, engineers work with thread-level parallelism and must deal with thread synchronization problems. For each individual execution thread, there exists a third axis of instruction-level parallelism, where specialized extensions speed up specific operations. A prime example is Intel's Advanced Vector Extension (AVX), which introduces new vector registers and instructions for highly efficient vectorized code. Other vendors offer similar extensions, such as Arm's Scalable Vector Extension (SVE) and the RISC-V Vector Extension (RVV). More recently, hardware support for matrix operations has become common with extensions like Intel's Advanced Matrix Extension (AMX) and Arm's Scalable Matrix Extension (SME).

RISC-V is a relatively new architecture in the market, and its strength lies in the ISA standard being open-source and extensible. That means that any vendor building RISC-V hardware can implement their own instructions. RISC-V for high-performance computing is still in its early days, and currently, there is no ratified standard for matrix operations similar to Intel's AMX and ARM's SME mentioned before. Our research group has been working on building a prototype matrix accelerator for RISC-V. That involves proposing new ISA extensions, developing simulators, optimized software kernels that will explore those new instructions, and also retargeting a compiler to generate code using the new ISA extension. In this minicourse, we address this last step. The goal is to show how one can expand an existing RISC-V LLVM compiler backend, including new matrix specialized instructions.

This document is organized as follows: Section 2.2 provides background on the RISC-V instruction set and the LLVM compiler infrastructure. Section 2.3 presents the extension that will be implemented, while Section 2.4 provides the steps to write it using TableGen syntax. Section 2.5 demonstrates the new instructions in action. And finally, Section 2.6 provides concluding remarks and suggestions for next steps.

2.2. Background

This section covers the background necessary to carry out the implementation of the Matrix Extension for RISC-V. In Section 2.2.1 the basics of the RISC-V instruction set are covered, while Section 2.2.2 introduces the necessary tools to work with LLVM.

2.2.1. RISC-V

RISC-V is an open standard Instruction Set Architecture (ISA). It defines a set of registers and instructions that operate on those registers. Because RISC-V is free to use, hardware vendors can implement their processors according to the specification and automatically leverage the existing tools around it, such as compilers, linkers, operating systems, and more.

As the name suggests, RISC-V follows the Reduced Instruction Set Computer

(RISC) philosophy: it provides a simpler set of instructions that can be implemented efficiently in hardware. Consider Intel's x86 instruction set. Most instructions accept operands either from memory or from registers, sometimes with a variety of addressing modes. RISC-V, on the other hand, defines separate sets of instructions to access memory and to perform arithmetic computations. Thus, operands can only come from a register or be immediately encoded in the instruction itself. Besides that, RISC-V also reserves opcodes in the standard for custom, processor-specific instructions that are not tied to any particular ratified extension. The main use case for custom instructions is to add control instructions or optimized ALU instructions that have different semantics from the ones already defined in the standard.

RISC-V was created with extensibility in mind. The instruction set is organized into modular components called extensions, which hardware vendors can optionally implement. Every RISC-V processor must support a minimal core known as the Base Integer Instruction Set, which is uniquely identified by the letter "I". For example, the I instruction set does not even include multiplication instructions, which are instead provided by the "M" (for multiplication) extension. Similarly, the "F" and "D" extensions introduce single- and double-precision floating-point operations, respectively. Additional extensions cover vector processing, atomic operations, and bit manipulation, among others. Detailed descriptions of these modules can be found in the official RISC-V specifications.

Due to this extensibility and flexibility, RISC-V has rapidly gained traction across a wide range of computing domains, from deeply embedded IoT devices and secure hardware enclaves, to automotive controllers, high-performance computing (HPC) clusters, and datacenter-scale AI accelerators. As the ecosystem grows, so does the set of ISA profiles and custom instruction sets tailored for these use cases. The recent RISC-V profiles initiative (e.g., RVA22, RVA23, etc.) defines standardized combinations of extensions to facilitate software portability while still embracing specialization.

This flexibility at the ISA level, however, must be matched by equivalent flexibility in the software toolchain, particularly in the compiler backend. Robust compiler support is essential to make custom instructions usable in real-world applications and to ensure code generation fully exploits the capabilities of the underlying hardware.

Instruction Encoding

The RISC-V Base Instruction Set defines instructions to be 32-bits long, grouped in six instruction types, shown in Figure 2.1. It is possible to note that the opcode always occupies the least significant seven bits of the instruction. The remaining bits (7-31) are interpreted differently depending on the instruction type.

Following is a description for each kind of instruction:

R-type – Represents instructions that take operands from registers `rs1`/`rs2` and write the output to another register `rd`. Fields `funct3` and `funct7` control which operation is performed (i.e., add, sub, or, and, etc). Examples include arithmetic and logic instructions such as `add`, `xor`, and `mul`, etc. This is the most common instruction type.

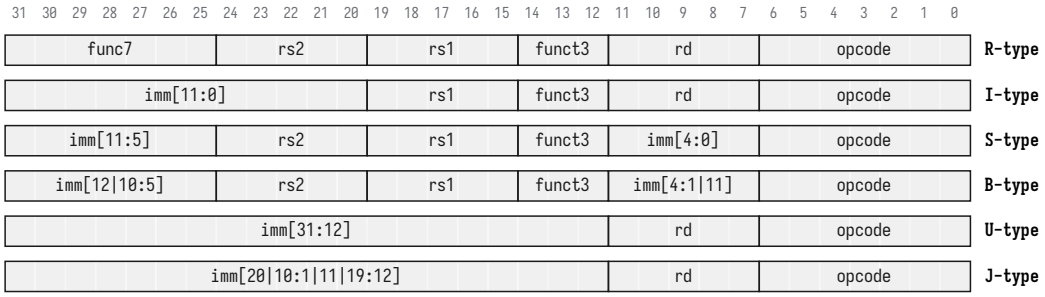

Figure 2.1. RISC-V base instruction formats.

I-type – Represents instructions that take one operand from a register `rs1` and another immediate operand. The output is a register `rd`. The funct3 `field` controls what the instruction computes. Examples include arithmetic instructions such as `addi`, `xori`, and `muli`, etc. This instruction format is also used to represent load instructions in which the immediate value is an offset from the address stored in the source register.

S-type – Represents store operations: the value to be stored is given by the second source register, the base address of the store is in the first source register, and there is an immediate offset from the base address.

B-type – Represents branch instructions. Compare the values in registers `rs1` and `rs2` , and add the immediate value to the program counter if they are equal. The field `funct3` encodes which comparison function to use (equal, not equal, less than, or greater than or equal).

U-type – Represents upper immediate instructions. Takes the immediate value and writes it to the upper part of register `rd`.

J-type – Represents the jump and link instruction. It saves the address of the next instruction (Program Counter + 4) into the destination register (rd), then updates the program counter by adding the immediate value, effectively performing a jump.

Opcodes

As mentioned earlier, RISC-V instructions use the lower seven bits to represent the opcode. Bits 0 and 1 of the opcode are always 1. Therefore, it is possible to represent 32 opcodes using the remaining five bits. The reason for this restriction is to differentiate 32-bit instructions from the compressed 16-bit instructions defined in the Compressed Instruction Set (“C” Extension). In this standard, the opcode occupies only two bits and can assume values 00, 01, or 10. In this manner, the hardware can unequivocally distinguish normal instructions from compressed instructions by just checking the first two bits. Table 2.1 shows how opcodes are encoded, taking the form `XXYYY11`, where `XX` is

mapped to the rows and YYY is mapped to the columns. The opcodes with YYY = 111 are currently reserved and have been omitted from the table.

	000	001	010	011	100	101	110
00	LOAD	LOAD-FP	*custom-0*	MEM	IMM	AUIPC	IMM32
01	STORE	STORE-FP	*custom-1*	AMO	REG	LUI	REG32
10	MADD	MSUB	NMSUB	NMADD	REG-FP	REG-V	*custom-2*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	REG-VE	*custom-3*

Table 2.1. RISC-V Opcodes.

Note the presence of four custom opcodes in the table. Such opcodes cannot be used by standard extensions and are reserved for vendor-specific instructions. The programmer can select any one of them to implement the custom matrix instructions.

2.2.2. LLVM

Compilers are generally implemented as a series of transformations on a source program. Common software engineering practice divides this pipeline into three stages:

The Frontend is responsible for parsing the input program, performing semantic analysis, and translating the code into an intermediate representation (IR).

The Middle-end takes the program in this intermediate form, runs a series of analyzes and transformation passes, and translates it to a lower-level representation that is very close to assembly language.

The Backend takes the output from the previous step, performs target-specific transformations (instruction selection, register allocation, etc) and generates executable or object code for the target architecture.

This layered structure, particularly the use of an intermediate representation, is crucial because it allows the frontend and backend to be decoupled. It becomes possible to compile many high-level languages (e.g., C/C++, Zig, Rust) into the same IR and then use a single backend to target multiple target architectures. Figure 2.2 visually represents this idea. In particular, LLVM is a framework for building compilers that implements a powerful intermediate representation called LLVM IR, along with several analysis and transformation passes, commonly referred to as “optimizations”.

In this context, LLVM has become the compiler infrastructure of choice for RISC-V development, offering a modular and retargetable backend that aligns well with RISC-V’s philosophy. Being able to integrate new instructions into LLVM allows hardware designers, researchers, and software developers to:

- Prototype and evaluate ISA changes rapidly;
- Generate optimized code for new instructions;

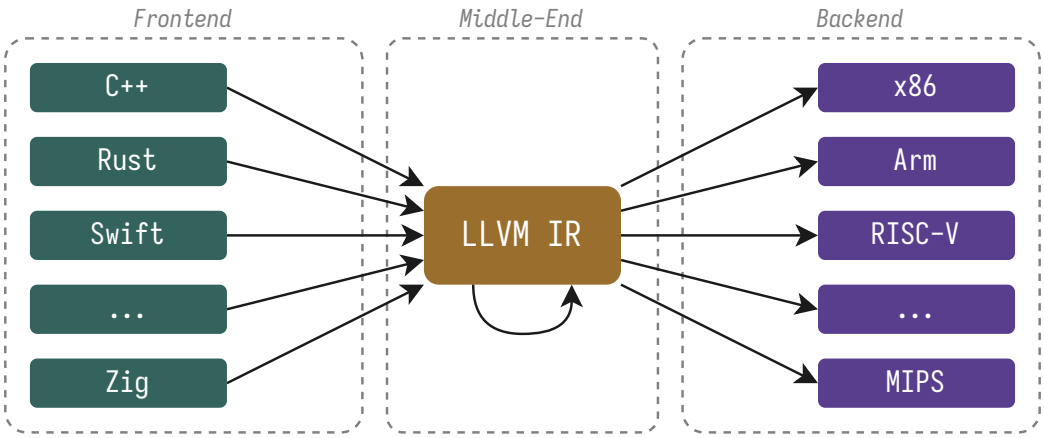

Figure 2.2. Overview of the compilation steps.

- Provide toolchain support for new RISC-V profiles or custom accelerators.

Focusing on how to extend the LLVM backend for RISC-V by adding new instructions, the modifications will occur in the backend, after the program is translated out of LLVM IR. The representation used in the LLVM backend is called Machine IR. However, it will not be necessary to interact with this representation since the backend can be extended by using a domain-specific language called TableGen.

2.2.3. TableGen

TableGen is a declarative language largely employed in the LLVM project. It allows programmers to define records of structured information that are used during the build process. For example, instead of writing an assembler by hand, this language shall be used to declare how the instructions are encoded (both in assembly and binary form) and then allow the C++ implementation to be generated automatically. This frees the programmer from writing boilerplate code and makes it much simpler to change the instructions later if the need arises. The syntax of TableGen is presented below.

TableGen Syntax

A record is a uniquely named list of typed key-value pairs, referred to as properties. Listing 2.1 provides an example record. The keyword `def` begins the declaration in line 1, followed by the record name `MyRecord` and a block delimited by curly braces. The record holds two properties: `Key1` of string type and value “Value 1” (line 2), and `Key2` of integer type with value 2 (line 3).

Nevertheless, records are rarely defined in this explicit manner. Instead, one can define a class that lists all properties a record should possess, much like a template, and then instantiate a record from it. Listing 2.2 shows a TableGen class in action. The `class` keyword is followed by its name `Car` and a list of typed parameters inside angle brackets

```
def MyRecord {
    string Key1 = "Value 1";
    int Key2 = 2;
}
```

Listing 2.1. TableGen record example.

(line 1). The body of the class is specified inside curly braces as a list of properties (lines 2-4). Now, instantiating a record from this "Car" class consists of using the `def` keyword as before, followed by the record name, a colon, and a list of classes from which the record is built (line 7). Note that classes themselves can also be derived from other classes.

```
class Car<string model, string color> {
    string Model = model;
    string Color = color;
}

def MyKombi : Car<"Kombi", "white">;
```

Listing 2.2. TableGen class example.

It is very common to define multiple records that share the values of some fields but not others. Consider the `Car` class from the previous example. Suppose that one would like to define a list of N car models in both black and white colors, thus $2N$ records are required. TableGen provides a pattern for efficiently encoding such variations with a multiclass, as shown in Listing 2.3.

It starts with the `multiclass` keyword, followed by a name and a list of parameters (line 1). Lines 2 and 3 define inner records for the multiclass. Note that both use the same model name but with different colors. Finally, three car models are defined in lines 6-8: Brasilia, Kombi, and Fusca from the `ColoredCar` multiclass. In order to distinguish between instantiating from a regular class, the keyword `defm` is used when instantiating from multi-classes. Writing this code results in 6 records being defined, where the name of the outer record is concatenated with the inner record, for example: `BrasiliaBlack`, `BrasiliaWhite`, `KombiBlack`, and so on.

```
multiclass ColoredCar<string model> {
    def Black : Car<model, "black">;
    def White : Car<model, "white">;
}

defm Brasilia : ColoredCar<"Brasilia">;
defm Kombi    : ColoredCar<"Kombi">;
defm Fusca    : ColoredCar<"Fusca">;
```

Listing 2.3. TableGen multiclass example.

Another common pattern in TableGen is to specify a property that is shared by the entire file or a large portion of it. Continuing with the previous example, suppose that now

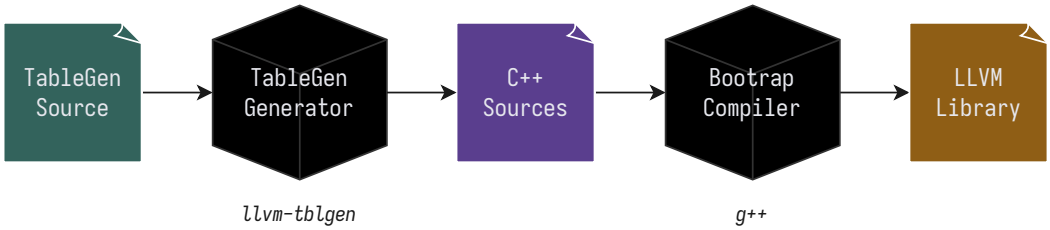

Figure 2.3. TableGen build process.

car records include a "Manufacturer" property. Even though multi-classes could be used for this purpose, an alternative syntax is shown in Listing 2.4. The `let` keyword introduces a comma-separated list of key-value pairs of properties, followed by the keyword `in`, and a block in curly braces (line 1). Inside the block, the records are defined normally, which effectively adds a "Manufacturer" property with the value "VW" to all of them at once (lines 2-4). While this feature might seem redundant for multi-class scenarios, both serve distinct purposes, as shown in the next section.

```
1 let Manufacturer = "VW" in {
2     defm Brasilia : ColoredCar<"Brasilia">;
3     defm Kombi    : ColoredCar<"Kombi">;
4     defm Fusca    : ColoredCar<"Fusca">;
5 }
```

Listing 2.4. TableGen let/in syntax example.

The last syntax relevant to this course is when we would like to create a sequence of records using an index in increasing (or decreasing) order. Listing 2.5 presents the `foreach` operator. It behaves like a for loop using the `Index` induction variable. The range of the loop goes from 0 to 3 (non-inclusive), increasing by 1 on each iteration. The symbol # is the concatenation operator. A record is defined in the body of the for-each loop with the name "Car" concatenated with the current index. The model name also uses the concatenation operator. In the end, the code shown in this example will create three records: `Car0`, `Car1`, and `Car2`.

```
1 foreach Index = !range(0, 3, 1) in {
2     def Car # Index : Car<"Model " # Index, "black">;
3 }
```

Listing 2.5. TableGen foreach example.

This concludes the brief introduction to TableGen. The language includes many additional constructs not covered in this tutorial. For a complete reference, check the TableGen Programmer's Reference[1] in the LLVM documentation.

[1] https://llvm.org/docs/TableGen/ProgRef.html

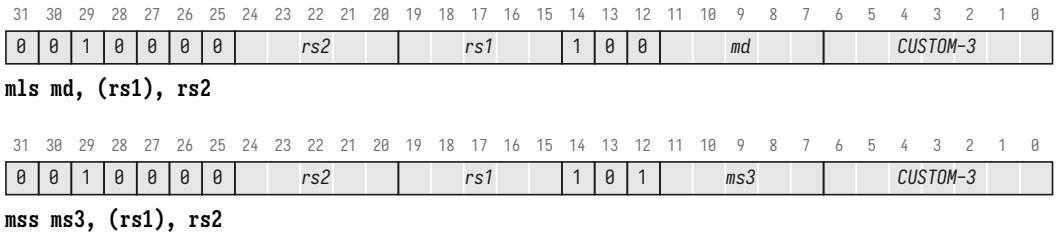

Figure 2.4. Memory (load/store) instruction encoding for the "xmatrix" extension.

As mentioned before, TableGen records do nothing on their own. They are instead used during the build process of LLVM itself to generate C++ code. This process is depicted in Figure 2.3. The TableGen source files are fed as input to a tool called `llvm-tblgen`, which expands all of the class instantiations into records and then filters these records through a "generator". EAlthough the internal workings of generators are beyond the scope of this course, we can assume that they produce a collection of C++ source files containing the implementation of the assembler, disassembler, and other tools. Still during the build process, those C++ source files are compiled using the bootstrap[2] compiler (typically `g++`) and the code end up in a statically linked library for LLVM.

2.3. RISC-V Matrix Extension

The proposed extension, called "xmatrix", adds 32 new matrix registers `m0-m31` with length of 512 bits, interpreted as a 4x4 matrix of 32-bit elements. The extension also includes memory, arithmetic, and move operations that utilize the matrix registers. The assembler must produce matrix instructions only when the "xmatrix" feature is enabled; otherwise, the code should be rejected with an error. Instructions from this extension shall use the *custom-3* opcode (`1111011`) from Table 2.1, chosen arbitrarily from the custom opcodes.

2.3.1. Memory Instructions

First of all, there must be a way to read matrix registers to and from main memory. Therefore, the instructions "matrix load" and "matrix store" are defined according to the encoding in Figure 2.4.

The **Matrix Load with Row-Stride** (`mls`) instruction reads a matrix register from memory, starting at a specified base address. The destination matrix register is encoded using five bits in the field `md` (`m0-m31`). The base address comes from the source operand in `rs1` , which encodes a scalar register. Also, there is an additional source operand called the leading dimension of the matrix (or equivalently, "row-stride") in the scalar register `rs2`. Figure 2.5 demonstrates how this instruction works.

The diagram shows the memory as a sequence of elements, each with 32-bits. The base address points to the first element that should be read. The load operation will read a single row of four elements and store it in the first row of the matrix register. Then,

[2]Bootstrapping is the process of using one compiler to build the source of another compiler.

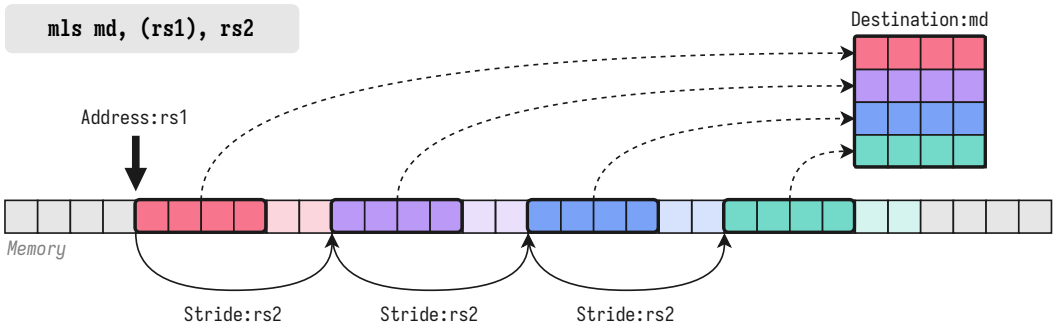

Figure 2.5. Matrix Load with Row-Stride (`mls`) instruction.

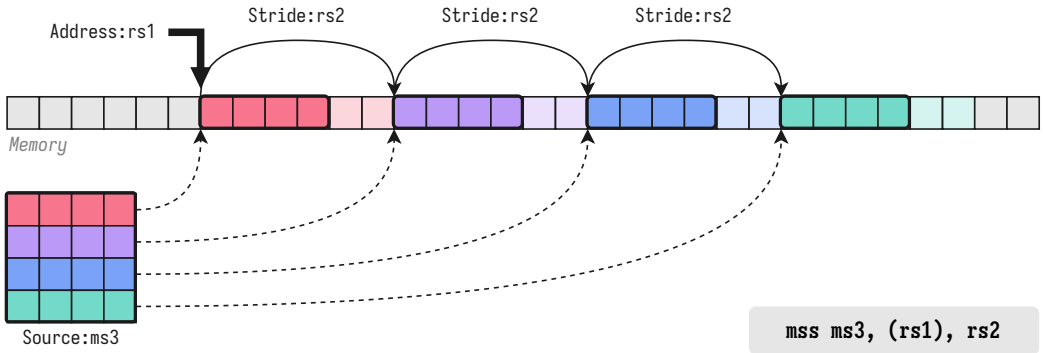

Figure 2.6. Matrix Store with Row-Stride (`mss`) instruction.

instead of reading the second row at an offset of four elements from the base address, it will actually jump "row-stride" elements forward before reading the next row. The process repeats until all rows of the destination register are filled.

The row-stride allows programmers to read a small 4×4 tile that is inside a larger $N \times M$ matrix. Without the stride, it would be necessary to first pack the corresponding elements of the tile sequentially in a separate region of memory using scalar operations and then execute a contiguous load. Notably, this strategy is highly inefficient in practice.

A contiguous load can be simulated when the row-stride matches the length of a row in the matrix register (i.e., `rs2 = 4`). Additionally, the row-stride can also be zero (the same four elements will be replicated to each row) or even negative (the rows are read in reverse order).

Similar to the load instruction, the **Matrix Store with Row-Stride** (`mss`) instruction writes a matrix register to memory, starting at a base address. It takes three source operands: the matrix register to be stored in `ms2`; the base address in memory from register `rs1`; and the row-stride in `rs2`. Its behavior is illustrated by Figure 2.6.

The first row of the matrix register is written to memory starting from the base address. Then, the pointer is incremented by the leading dimension, arriving at the address

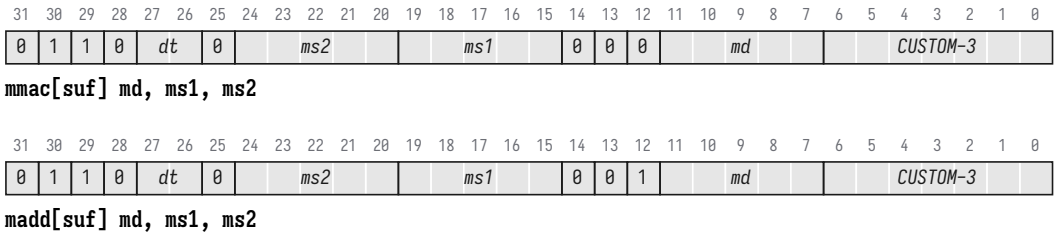

Figure 2.7. Arithmetic instructions encoding in the "xmatrix" extension.

where the second row will be stored. This process is repeated until the entire register has been written to memory. Similarly, the store instruction, like the load instruction, also accepts a null or negative stride.

2.3.2. Arithmetic Instructions

With the ability to read and write to matrix registers from memory, attention is focused on arithmetic instructions. Two groups of instructions are shown in Figure 2.7. The first group is the multiply-and-accumulate instruction, `mmac`. It is a group because there are three different data-types that this instruction can operate on: signed, unsigned, and floating point. Therefore we add a suffix `s`, `u`, or `f` respectively to each instruction. Likewise, another instruction that adds two matrices `madd` is defined for each data-type.

Bits 26 and 27 of the instruction encode the data-type being `00` for unsigned integer elements, `01` for signed integer elements, and `10` for single-precision floating point elements.

2.3.3. Movement Instructions

Finally, the instruction `mmv.xm` writes the value of a scalar register to all positions of a matrix register as shown in Figure 2.8. This operation is particularly useful for initialization, for example clearing a matrix register by moving the contents of `x0` (always zero). This pattern is so common, in fact, that a pseudo-instruction "`mzero md`" is defined to represent "`mmv.xm md, x0`".

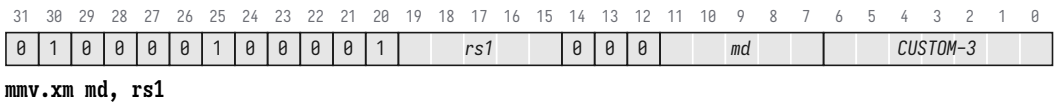

Figure 2.8. Movement instruction enconding in the "xmatrix" extension.

2.4. Implementation

In this section, it will be demonstrated how to implement the extension defined in the previous section. In Section 2.4.1, a new extension is declared to contain the new registers and instructions. Section 2.4.2 shows how to declare the new matrix registers. Finally,

the instructions themselves are defined in Sections 2.4.3, 2.4.4, and 2.4.5.

2.4.1. Feature

In order to define a new RISC-V extension, we must create two records in the file `RISCVFeatures.td`. This file is shown in Listing 2.6. The first record (lines 3-4) declares a new `RISCVExtension` that takes the major and minor versions of the specification as integers and a string with the extension name. In this particular case, the record name matters and must begin with the prefix `FeatureVendorX`. It will automatically add a new architecture flag "xmatrix" on the command-line.

```
1 // File: llvm/lib/Target/RISCV/RISCVFeatures.td
2 def FeatureVendorXMatrix
3     : RISCVExtension<0, 1, "Matrix Extension">;
4
5 def HasVendorXMatrix
6     : Predicate<"Subtarget->hasVendorXMatrix()">,
7       AssemblerPredicate<(all_of FeatureVendorXMatrix),
8                          "'XMatrix' (Matrix Extension)">;
```

Listing 2.6. RISC-V Matrix Extension declaration.

The second record (lines 6-9) defines predicates that indicate whether the feature is enabled or not. It is created from two classes: `Predicate` , which takes a C++ expression that performs the test; and `AssemblerPredicate` , which takes a TableGen predicate followed by a name. Although redundant, both predicate classes must be instantiated since they are used in different parts of the generation.

2.4.2. Registers

After declaring the new feature, it is possible to define the registers in Listing 2.7. The operator `foreach` is employed here. For each iteration, define a register (`RISCVReg`) with an index and a name (line 2). The # symbol means concatenation in TableGen. Therefore, registers are named "m0", "m1", etc.

```
1 // File: llvm/lib/Target/RISCV/RISCVRegisterInfo.td
2 foreach Index = !range(0, 32, 1) in {
3     def M#Index : RISCVReg<Index, "m"#Index>;
4 }
5
6 def MR : Registerclass<[v16i32, v16f32], (sequence "M%u", 0, 31), 1>;
```

Listing 2.7. Matrix registers declaration.

A register class is also defined in line 6. It takes, as the first parameter, a list of types that can be stored in such a register. In this case, a vector of 16 (4x4) 32-bit integers or 16 single precision floating point elements. The second parameter is a list of strings containing register names. In particular, it expands to a list of all matrix registers.

2.4.3. Memory Instructions

A new instruction can be defined in TableGen by instantiating a record from the `RVInst` class, as shown in Listing 2.8. This class has the following type parameters:

- `outs` specifies which operands are output from the instruction, along with the register class they belong to;
- `ins` specifies which operands are input to the instruction, along with their register class;
- `opcodestr` is a string with the assembly mnemonic that identify this instruction.
- `argstr` is a format string for the operands, which placeholders starting with the character "$";
- `pattern` is a list of patterns (i.e., record created from class `Pattern`) that converts an LLVM IR operation to Machine IR. This parameter shall be left empty; and
- `format` specifies which format is used by this instruction (R-type, I-type, etc from Figure 2.1).

Starting at line 1, a new subclass `MatrixInstLoad` is created from the `RVInst` class (line 2). Our new class takes the same parameters mentioned above and "forwards" them to the base class. It defines new operands `md`, `rs1`, and `rs2`, all encoded with five lines (lines 4-6). Next, the instruction encoding is declared. For each bit range in the `Inst` property we write which value it should take (lines 8-13). For example, the first seven bits (6-0) correspond to the opcode, thus we assign the value of the `OPC_CUSTOM_3` (line 13). This record has been previously defined in LLVM and its value is exactly the same as described in Table 2.1. According to the instruction encoding presented earlier, bits 31-25 are constant, therefore we directly assign the value `0b0010000` to it (line 8).

With the class properly coded, a new record called `MLS` is created (lines 16-19). We first wrap the record definition in a `let`/`in` block that sets a few required properties (line 16). The predicate previously defined that identifies the matrix extension is used to indicate that this instruction is only available when the feature is enabled. Additionally, we indicate that this instruction has no side effects (i.e. does not change a register that is not explicitly identified in the format), it may load from memory but never stores.

The instruction itself uses the mnemonic "mls" and its arguments are formatted like "md, rs1, rs2" (line 17). Pay close attention to the outs/ins declarations (line 17). This syntax is known as a DAG type and encode a list of operands with their type attached to it. We say the output operand (`outs`) from this instruction is the `md` register that should be one of those in the `MR` register class defined in the previous section. Another DAG value is used to specity the inputs `ins`: register `rs1` from `GPRMemZeroOffset` and register `rs2` from the `GPR` register class. The `GPR` is a register class that includes all general purpose registers `x0-x31`, while the `GPRMemZeroOffset` class includes all general

```
// File: llvm/lib/Target/RISCV/RISCVInstrInfo.td
class MatrixInstLoad<string opcodestr, string argstr, dag outs, dag ins
    >
    : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
    bits<5> md;
    bits<5> rs1;
    bits<5> rs2;

    let Inst{31-25} = 0b0010000;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b100;
    let Inst{11-7}  = md;
    let Inst{6-0}   = OPC_CUSTOM_3.Value;
}

let Predicates = [HasVendorXMPE], hasSideEffects = 0, mayLoad = 1,
    mayStore = 0 in {
    def MLS : MatrixInstLoad<"mls", "$md, $rs1, $rs2",
            (outs MR:$md), (ins GPRMemZeroOffset:$rs1, GPR:$rs2)>;
}

class MatrixInstStore<string opcodestr, string argstr, dag outs, dag
    ins>
    : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
    bits<5> ms3;
    bits<5> rs1;
    bits<5> rs2;

    let Inst{31-25} = 0b0010000;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b101;
    let Inst{11-7}  = ms3;
    let Inst{6-0}   = OPC_CUSTOM_3.Value;
}

let Predicates = [HasVendorXMPE], hasSideEffects = 0, mayLoad = 0,
    mayStore = 1 in {
    def MSS : MatrixInstStore<"mss", "$ms3, $rs1, $rs2",
            (outs), (ins MR:$md, GPRMemZeroOffset:$rs1, GPR:$rs2)>;
}
```

Listing 2.8. TableGen classes for memory instructions.

purpose registers but are used as a memory offset operand. This causes this operand to be wrapped in parenthesis, like the scalar load instruction `ld x3,(x4)`.

The store follows a similar pattern, a subclass of `RVInst` called `MatrixInstStore` is created to contain the encoding of all 32 bits (lines 21-33). The record `MSS` is instantiated with the corresponding mnemonic, argument format string and the outs/ins DAG patterns (lines 36-37). Note however that store operations do not produce values, therefore all operands are inputs and the outputs are empty. Another observation is that we use the same predicate as the load and also identify that this instruction may store to memory.

2.4.4. Arithmetic Instructions

In order to specify arithmetic instructions with TableGen, it is useful to first define constants for the "funct3" and "dt" fields (Listing 2.9). A class that holds a string of 3-bits is defined (lines 2-4) and then the constants for `mmac` and `madd` are defined according to the encoding (lines 5-6). A class that takes a 2-bit string as a parameter is declared to encode the data-type (lines 8-10). Records for each data type are declared according to the specification (lines 11-13).

```
1  // File: llvm/lib/Target/RISCV/RISCVInstrFormats.td
2  class MatrixAluFunct3<bits<3> value> {
3      bits<3> Value = value;
4  }
5  def MMacFunct3 : MatrixAluFunct3<0b000>;
6  def MAddFunct3 : MatrixAluFunct3<0b001>;
7
8  class MatrixDataType<bits<2> value> {
9      bits<2> Value = value;
10 }
11 def MUnsigned : MatrixDataType<0b00>;
12 def MSigned   : MatrixDataType<0b01>;
13 def MFloat    : MatrixDataType<0b10>;
```

Listing 2.9. Matrix instruction fields.

Similar to the memory instructions, a new class is created in Listing 2.10 from the `RVInst` base class (line 2). It takes the same parameters are the classes for memory instruction but also the bits corresponding to funct3 and the ones corresponding to the data-type field. The bits of the instruction are declared according to the specification (lines 8-15). Since all arithmetic instructions are very similar, another class is created in line 18. It indicates how the arguments of the instruction are formatted in assembly, that is "$md, $ms1, $ms2" in the argument format string. It also encodes that register "md" is an output and registers "ms1" and "ms2" are inputs, all of them from the class of matrix registers (line 19).

Now it is time to encode the variations regarding the data-type using a multiclass (line 21). It takes the opcode and funct3 fields as parameters and defines three inner records, one for each data-type, passing along the correct constants defined earlier (lines 22-24). Finally, instruction records themselves are declared by passing the corresponding opcode and funct3 bits (lines 27-30). Because of the multiclass, the result is a total of six

```
// File: llvm/lib/Target/RISCV/RISCVInstrInfo.td
class MatrixInstALUBase<string opcodestr, string argstr,
    MatrixAluFunct3 funct3, MatrixDataType dt, dag outs, dag ins>
        : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
    bits<5> md;
    bits<5> ms1;
    bits<5> ms2;

    let Inst{31-28} = 0b0110;
    let Inst{27-26} = dt.Value;
    let Inst{25}    = 0b0;
    let Inst{24-20} = ms2;
    let Inst{19-15} = ms1;
    let Inst{14-12} = funct3.Value;
    let Inst{11-7}  = md;
    let Inst{6-0}   = OPC_CUSTOM_3.Value;
}

class MatrixInstALU<string opcodestr, MatrixAluFunct3 funct3,
    MatrixDataType dt>
        : MatrixInstALUBase<opcodestr, "$md, $ms1, $ms2", funct3, dt, (
    outs MR:$md), (ins MR:$ms1, MR:$ms2);

multiclass MatrixInstALU_SUF<string opcodestr, MatrixAluFunct3 funct3>
    {
    def S : MatrixInstALU<opcodestr,     funct3, MSigned>;
    def U : MatrixInstALU<opcodestr#"u", funct3, MUnsigned>;
    def F : MatrixInstALU<opcodestr#"f", funct3, MFloat>;
}

let Predicates = [HasVendorXMPE], hasSideEffects = 0, mayLoad = 0,
    mayStore = 0 in {
    defm MADD : MatrixInstALU_SUF<"madd", MAddFunct3>;
    defm MMAC : MatrixInstALU_SUF<"mmac", MMacFunct3>;
}
```

Listing 2.10. Arithmetic matrix instruction declaration.

new records being defined: `MMACS`, `MMACU`, `MMACF`, `MADDS`, `MADDU`, and `MADDF`. No arithmetic instruction perform a load, store, or have any side-effects.

2.4.5. Movement Instructions

The final move instruction is given in Listing 2.11. The pattern closely resembles the other instructions: define a new class `MatrixInstMov` (line 2-3), declare the operands (lines 4-5), declare the encoding (lines 7-11), and finally instantiate the record (line 15). We also define a pseudo-instruction `mzero` as an anonymous record that inherits from `InstAlias` prividing the assembly format as a string and a DAG pattern to substitute for (line 16). Similar to arithmetic instructions, movement instructions neither load, nor store to memory and have no side effects.

```
1  // File: llvm/lib/Target/RISCV/RISCVInstrInfo.td
2  class MatrixInstMov<string opcodestr, string argstr, dag outs, dag ins>
3      : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
4      bits<5> md;
5      bits<5> rs1;
6
7      let Inst{31-20} = 0b010000100001;
8      let Inst{19-15} = rs1;
9      let Inst{14-12} = 0b000;
10     let Inst{11-7}  = md;
11     let Inst{6-0}   = OPC_CUSTOM_3.Value;
12 }
13
14 let Predicates = [HasVendorXMPE], hasSideEffects = 0, mayLoad = 0,
       mayStore = 0 in {
15     def MMV_XM : MatrixInstMov<"mmov.xm", "$md, $rs1", (outs MR:$md), (
       ins GPR:$rs1)>;
16     def : InstAlias<"mzero $md", (MMV_XM MR:$md, X0)>;
17 }
```

Listing 2.11. Movement instruction declaration.

2.5. Testing

With the implementation of the new instructions complete, we now proceed to compile the modified LLVM sources (Section 2.5.1) and test the assembler and disassembler using a dummy program (Section 2.5.2). This section concludes with a discussion on an optimized microkernel for general matrix multiplication (Section 2.5.3).

2.5.1. Compiling LLVM

The LLVM project uses CMake to generate the Makefiles required for building its libraries and tools, including Clang.

Listing 2.12 shows the command that should be executed in the root directory of the LLVM repository. The first command at line 1 generates the build system. By default, only the LLVM libraries (without Clang) are compiled, and the only backend enabled by default is the architecture of the host, most likely x86 or Arm. Thus, we explicitly enable Clang for the RISC-V backend. The second command on line 6 actually builds the

```
$ cmake -S . -B build \
    -DCMAKE_BUILD_TYPE=Debug \
    -DLLVM_ENABLE_PROJECTS="clang" \
    -DLLVM_TARGETS_TO_BUILD="RISCV"

$ cmake --build build -j
```

Listing 2.12. Compile LLVM with CMake.

libraries and the executable. Due to the size of LLVM, this process can take anywhere from 20 minutes to a few hours, depending on the host hardware specifications.

2.5.2. Assembling and Disassembling

To verify the correctness of the assembler and disassembler, consider a dummy function that simply multiplies register `m1` by `m2` and stores the result in `m0`, as shown in Listing 2.13 (lines 3-4). Clang can be used to assemble this function into object code using the command on lines 7-8. The flag "`-target riscv64`" specifies that RISC-V should be the target architecture, and "`-march=rv64g_xmatrix`" specifies that the target architecture is 64-bits, with the "G" extension enabled (short for "IMAFD" extensions) and also "xmatrix". Finally, we invoke `llvm-objdump` with the `-d` flag to disassemble the previously assembled object file and confirm that it reproduces the original assembly.

```
# Dummy assembly file
$ cat dummy.s
dummy:
    mmacf m0, m1, m2

# Assemble the code into and object file
$ build/bin/clang \
    -target riscv64 -march=rv64g_xmatrix -c -o dummy.o dummy.s

# Disassemble the object file
$ build/bin/llvm-objdump -d dummy.o
dummy.o:        file format elf64-littleriscv

Disassembly of section .text:

0000000000000000 <dummy>:
       0: 6a208077      mmacf   m0, m1, m2
```

Listing 2.13. Assemble and disassemble dummy code using Clang.

2.5.3. Matrix Multiplication Microkernel

OpenBLAS is a software package that implements highly optimized kernels to compute the most common numerical linear algebra routines. It is organized into three levels:

1. The first level includes operations on vectors, such as vector scaling (`SCAL`), dot product (`DOT`), and vector normalization (`NRM2`).

2. The second level gathers matrices through vector operations, such as general matrix-vector multiplication (`GEMV`) and general rank update (`GER`).

3. The third and last level includes matrix by matrix operations, with the most important being the general matrix-matrix multiplication (`GEMM`).

From all the routines available, matrix multiplication is arguably the most ubiquitous operation. It is used in applications across numerous domains, such as engineering, geophysics, and deep learning. In particular, the essence of Large Language Models (LLMs) is several matrix multiplication kernels interspersed with other operations, such as activation functions (ReLU, GeLU, etc.) and tensor normalization (LayerNorm, BatchNorm, etc). Therefore, being able to speed up this routine using hardware-level acceleration, such as the matrix extension that is being developed in this course, is of utmost importance to enable inference locally on custom built RISC-V hardware.

The GEMM kernel, however, not only multiplies two matrices but also accumulates the result into a third matrix. In practice, given matrix A with size $m \times k$, matrix B with size $k \times n$, and matrix C with size $m \times n$, it computes:

$$C = \alpha AB + \beta C \tag{1}$$

where α and β are real-valued scalars that control the sign and scaling of the terms being added. The BLAS package implements several versions of each routine for different data-types. Thus, we look at `SGEMM` which operates on matrices with single-precision floating-point elements.

To improve cache locality, the kernel computes the result matrix in tiles of fixed size, which is usually tied to the size and number of registers available. The tiles are first laid out contiguously in memory in a separate buffer, an operation called packing. For reasons that will become clear later, matrix A tiles are also transposed during packing. Then, tiles from A and B are multiplied together and accumulated into the corresponding tile in C. We shall call this operation on individual tiles a "micro-kernel". A micro-kernel with a tile size of 4x4 is presented below.

Listing 2.14 shows how this can be implemented in Assembly language, assuming $\alpha = 1$ and $\beta = 0$. First, the accumulator matrix register `m0` is cleared (line 2), and the index variable on k is initialized in scalar register `t0` (line 3). Register `t1` stores the size of a single row from a matrix register in bytes, which equals 16 (line 4). Inside the loop beginning at line 7, one matrix register is loaded from A (lines 8 and 9), and another from B (lines 10 and 11). Both registers are multiplied and accumulated into `m0` (line 13). The loop induction variable is incremented (line 14), and the loop exits if it is greater than or equal to k; otherwise, it iterates again (line 14). The micro-kernel ends by writing the accumulator to memory (line 17) and returning.

Even though the kernel shown above is correct, it is possible to hide memory latencies by having loads, stores, and MACs happen concurrently. The idea is to make better use of the processor pipeline by dispatching as many operations as possible so that all functional units are busy at the same time. This can be achieved by choosing

```
usgemm_naive:
    mzero m0
    li t0, 0            # i = 0
    li t1, 16           # Row size (bytes)
    sll a0, a0, 2       # K *= sizeof(float)

.loop:
    add t2, a3, t0      # A + i
    mls m20, (t2), a6
    add t3, a2, t0      # B + i
    mls m25, (t3), a5
    mmacf m0, m20, m25
    add t0, t0, t1      # i += 4
    blt t0, a0, .loop   # if i >= K, exit loop

.loop.end:
    mss m0, (a1), a4
    srl a0, a0, 2       # K /= sizeof(float)
    ret
```

Listing 2.14. Naive outer product micro-kernel.

a tile larger than the register size. Consider the two mutually exclusive strategies for implementing a highly-parallel GEMM micro-kernel. The first step computes the inner product between a row i of A and a column j of B, and accumulates it in element i, j of C. Note that one element of C is computed at a time. The second approach, however, computes the outer product between column i of A and row j of B, yielding a new matrix of size $m \times n$ that must be accumulated into C. No solution is better than the other in general. Choosing the best approach depends on the details of each architecture. For this RISC-V matrix extension in particular, we adopt an outer product micro-kernel.

Figure 2.9 introduces the layout of a single tile of each matrix in a GEMM operation. The idea is to choose the largest tile size of C such that all temporaries can be kept in registers rather than spilled to memory. That way, one can load 4 matrix registers from A along the m dimension, and 6 matrix registers from B along the n dimensions, resulting in $4 \cdot 6 = 24$ outer products, each stored in their own register. It sums up to a total of 34 registers, which is larger than the 32 available. However, it is possible to allocate only two registers that hold elements from A and reuse them accordingly. This brings the total to exactly 32 registers. Therefore, we choose a tile size of 16x24 elements, grouped in registers of 4x4 elements. This way, tile $C_{i,j}$ is the result of the operation $A_j B_i + C_{i,j}$.

The assembly source code for the improved micro-kernel is too large to be reproduced in full therefore we present the pseudo-code in Listing 2.15. The full code is available online[3]. We start by setting a constant `ROW_LEN` to be equal to the number of elements of a single row in a matrix register (4). Since this is an outer-product kernel, the outermost loop iterates over the k dimension (line 2). Next, iteration occurs inside the tile for matrix C, going over elements `acc[i,j]` that correspond to elements $C_{i,j}$ from Figure 2.9 (lines 3-4). The body of the loop nest loads a matrix register from A (`ma`), a matrix

[3] https://gist.github.com/leiteg/a5f4eb1fdd3fc7f2cca947b8c7e2be77

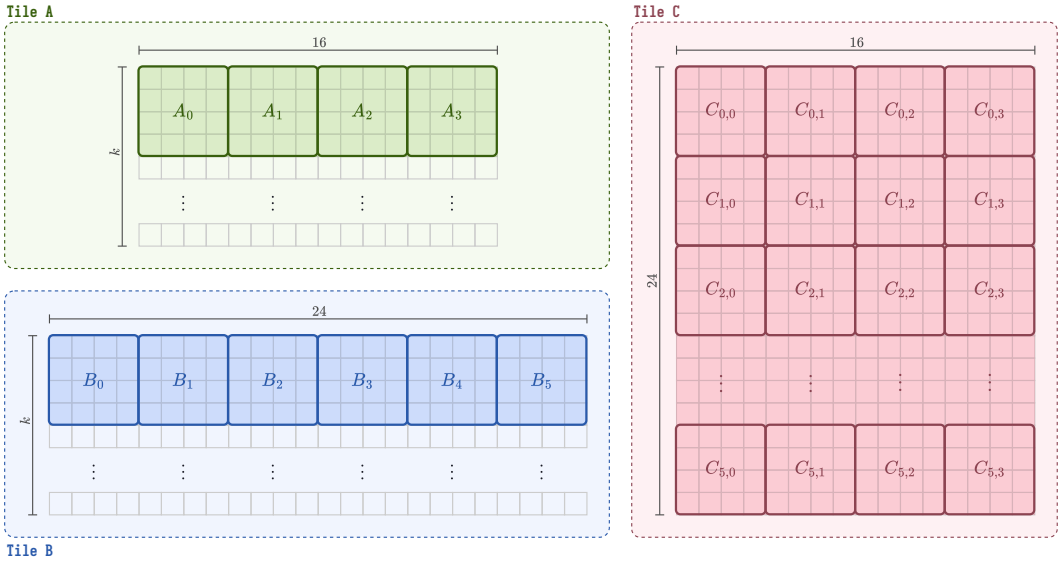

Figure 2.9. Outer product micro-kernel using the RISC-V Matrix extension (Tile A is green, tile B is blue, and tile C is red).

```
ROW_LEN = 4
for l in range(k, step=4):
    for j in range(6): # unroll
        for i in range(4): # unroll
            ma = mls(A[k * 16 + i * ROW_LEN], ROW_LEN)
            mb = mls(B[k * 24 + j * ROW_LEN], ROW_LEN)
            acc[i, j] = mmac(ma, mb, acc[i, j])
for j in range(6): # unroll
    for i in range(4): # unroll
        mss(acc[i, j], C[j * ldc + i * ROW_LEN], ldc)
```

Listing 2.15. Pseudo-code for the improved micro-kernel.

register from *B* (`mb`), multiply them together and accumulates on the corresponding accumulator register that is assumed to be initialized with zero (lines 5-7). As discussed above, there are enough registers to compute all iterations of the two innermost loops without the need to spill to memory. For this reason we unroll the loops on `i` and `j`. At this point, the outer product of matrices *A* and *B* is computed all that is left is storing the values back to memory (lines 8-12). This loop nest is also unrolled for improved performance.

2.6. Concluding Remarks

This tutorial has presented how to implement a matrix processing extension for RISC-V using the LLVM compiler infrastructure. The resulting toolchain allows programmers to write kernels in assembly that use matrix instructions and assemble them into object code. Programmers can also disassemble an object file back into instructions in textual format.

Compilers are often treated as opaque black boxes and many programmers assume that experimenting with them is too difficult and not worthwhile. It is true that compilers are generally large and complex to understand holistically. However, in practice, much software engineering is utilized to tame the complexity and keep the design modular. This modularity makes it possible for curious learners to peer into the individual stages of the compilation process and develop a deeper understanding incrementally. The objective of this course was precisely to show that extending a compiler is both feasible and conceptually approachable when done methodically.

2.6.1. Next Steps

Hardware development naturally progresses at a slower pace than software development. Before tape-out, processors must undergo extensive testing through simulators and FPGAs, as hardware bugs cannot be fixed once the circuit is fabricated. Software development, on the other hand, is incremental and iterative. Bugs can be patched, features refined, and new versions deployed with relative ease. Therefore, a good next step is to implement a simulator capable of executing the new matrix instructions. Free and open-source projects like QEMU[4] and GEM5[5] provide a good starting point for implementing emulators.

But the work inside the compiler itself rarely stops at the assembler. Programmers prefer to write their computational kernels in a high-level language like C/C++. The interested student can use the Clang C/C++ compiler shipped with the LLVM project and extend it with new built-in functions. In general, for each new instruction, there is a corresponding built-in function that, once called in C, becomes a single instruction in the executable. However, the process is not so straightforward. Remember that the code is first transformed into LLVM IR before being compiled into machine code. Thus, it is necessary to add new intrinsic operations to the IR to bridge the gap between built-ins and the final instructions. Both built-ins and intrinsics are specified using the already familiar TableGen syntax. In this case, however, the programmer is required to write some additional C++ code to define how these representations are translated within the compiler.

[4] https://www.qemu.org/
[5] https://gem5.org

Capítulo 3

Tuning e depuração de aplicações em Open MPI 5.0

Aleardo Manacero (UNESP)

Abstract

Open MPI is one of the most used APIs for programming parallel applications in high performance systems. The efficient application of Open MPI functionalities is rather fundamental to achieve lower costs and maximum efficiency while executing such programs. In this document we present some mechanisms offered by Open MPI 5.0 that make easier to debug and tune an application, improving the performance of our programs. The chapter starts presenting an overview of the main MPI functions to control and communicate, both collective and point-to-point, processes. We follow with the description of the Modular Component Architecture (MCA), which provides a set of options for tuning and debugging of MPI programs. In sequence several mechanisms for tuning programs, using MCA components, as well as debugging, including tools such as Valgrind, are presented. Finally, we conclude this chapter with a set of recommendations for tuning and debugging.

Resumo

Open MPI é uma das APIs mais usadas na programação de aplicações paralelas em sistemas de alto desempenho. Usar as funcionalidades do Open MPI de forma eficiente é importante para que os programas paralelos executem com o menor custo e máxima produtividade. Aqui apresentaremos mecanismos presentes no Open MPI 5.0 que facilitam os processos de depuração do código MPI e também de seu ajuste fino (tuning) de modo a aumentar a eficiência de nossos programas. O texto a seguir apresenta inicialmente um resumo das principais funções disponíveis no MPI, tanto de controle quanto de comunicação ponto-a-ponto e coletiva. Passa-se então à descrição da Modular Component Architecture (MCA), que fornece ao programador um conjunto de opções para tuning e depuração de programas MPI. Na sequência são apresentados mecanismos para tuning de programas, com o uso de componentes MCA, assim como para depuração, incluindo ferramentas como Valgrind. Por fim, fechamos o capítulo com um conjunto de recomendações finais para tuning e depuração.

MPI (Message Passing Interface) é uma API em uso para a paralelização de aplicações em uso há 30 anos, sendo ainda muito relevante para computação de alto desempenho. Seu projeto se iniciou com o objetivo de criação de um padrão para programação paralela, uma vez que naquele momento existiam várias bibliotecas trabalhando de maneiras bastante diferentes [1, 2, 3]. Como a ideia original era de se obter um padrão, o consórcio que criou o MPI permitiu que se criassem distribuições diferentes da API, como o mpich [4], LAN/MPI (descontinuado), Open MPI [5], e mesmo uma distribuição para Windows, MS-MPI [6]. Nas próximas páginas serão apresentados, em sequência, uma introdução aos comandos fundamentais do MPI, os conceitos da MCA (Modular Component Architecture) e os mecanismos presentes na distribuição openMPI para fazer o *tuning* e a depuração de aplicações MPI.

3.1. Introdução ao MPI

Um programa MPI é composto por comandos da linguagem nativa e por chamadas para funções da biblioteca do MPI. A sua execução no sistema paralelo é iniciada por um comando específico, que dispara a execução do número projetado de processos paralelos nas máquinas do sistema. Esse comando é o *mpirun* ou *mpiexec*, com a seguinte sintaxe:

```
mpirun [ options ] program [ <args> ]
```

ou ainda na seguinte forma, para programas MPMD:

```
mpirun [ global options ]
       [ local options1 ] program1 [ <args1> ]
       ...
       [ local optionsN ] programN [ <argsN> ]
```

Existem muitas opções que podem ser usadas, especificando diversas configurações para a execução do programa. Nos limitaremos nesse momento a apresentar duas delas, que indicam o número *X* de processos paralelos a serem executados e o nome de um arquivo que lista as máquinas (uma por linha) em que a execução ocorrerá, que são:

```
mpirun [ -n X ] [ --hostfile <filename> ] <program>
```

O programa em MPI deve delimitar a região de código em que comandos MPI serão utilizados. Isso implica em definir o início dessa região, com `MPI_INIT`, e seu final, com `MPI_FINALIZE`.

A chamada de `MPI_INIT` faz com que o sistema inicialize suas estruturas de controle de comunicação, definindo a variável de ambiente `MPI_COMM_WORLD`. É essa variável que armazena as informações que permitem a comunicação entre os processos paralelos.

3.1.1. Funções de controle de execução

Após a inicialização de `MPI_COMM_WORLD` é possível aos processos paralelos obter informações sobre o ambiente completo, ou mesmo modificar sua estrutura topológica. Para isso existem funções específicas de controle, para determinar por exemplo quantos processos foram disparados, ou qual o identificador daquele processo dentro da hierarquia

criada. Existem ainda comandos que permitem modificar a topologia entre os processos ou conhecer processos vizinhos nessa topologia.

A Tabela 3.1 apresenta as principais funções que permitem que os processos paralelos tenham algum controle sobre sua execução. Destacamos aqui `MPI_Comm_size`, que retorna ao processo que a chamou o número de processos paralelos que foram criados, e `MPI_Comm_rank`, que diz ao processo qual a sua identificação dentro dos comunicadores paralelos.

É importante também destacar a função `MPI_Cart_create`, que vai organizar os processos em uma topologia cartesiana. Os parâmetros para essa função determinam o número de eixos do espaço cartesiano (*dims*), sendo que cada dimensão terá até *ndims* processos. Ainda temos um parâmetro, *reorder*, usado para habilitar ou não a reorganização dos processos de forma a otimizar a topologia. Um último parâmetro, *wrap*, que indica se processos numa dimensão terão vizinhança de forma circular.

Uma última função da Tabela 3.1 que merece ser descrita é `MPI_Cart_shift`, que permite identificar os vizinhos daquele processo dentro da topologia. O parâmetro *displ* indica a distância dos vizinhos a serem identificados nos parâmetros *src* e *dst*.

3.1.2. Funções de comunicação entre processos

Com os processos criados e estruturados topologicamente, a comunicação entre eles é que viabiliza a paralelização da aplicação. Essa comunicação envolve comandos de envio/recebimento de uma mensagem, envio de mensagens de *broadcast*, etc. Apresentamos na Tabela 3.2 apenas algumas das funções existentes (as mais simples), mas deixamos claro que no MPI a comunicação pode ocorrer de várias formas, envolvendo operações síncronas ou assíncronas, com temporização, etc.

`MPI_Send` e `MPI_recv` são as funções de comunicação mais simples. É importante observar que mensagens transmitidas devem conter sempre valores de mesmo tipo (*type*), sendo que os tipos em MPI são similares aos tipos comuns em C, embora recebam o prefixo `MPI_` antes dos nomes típicos de C. O manual do Open MPI [5] apresenta com detalhes os tipos definidos.

Essa função de envio de mensagens tem como característica o fato de não retornar de sua chamada antes do conteúdo a ser transmitido ser copiado para o dispositivo de rede. Assim, `MPI_Send` é uma função bloqueante, em que o processo emissor aguarda que os dados sejam transferidos de seu *buffer* para um segundo *buffer*, que pode ser tanto um local intermediário no próprio sistema ou o de recebimento no sistema remoto. Existe, entretanto, uma função de envio não bloqueante, que é `MPI_Isend`, em que o processo é liberado antes mesmo dos dados contidos em "msg" começarem a ser transferidos para um *buffer* intermediário.

A diferença entre as duas funções está na latência e na confiabilidade da transmissão. A latência é menor quando usamos envio não bloqueante, uma vez que o processo está apto a continuar executando assim que completar a chamada de envio. Infelizmente, essa redução de latência cria a possibilidade do processo modificar os dados supostamente transmitidos antes da transferência efetivamente acontecer.

Tabela 3.1. Principais funções de controle dos processos MPI

Função	Descrição
`MPI_Comm_size(MPI_COMM_WORLD, &size)`	retorna o número de processos disparados
`MPI_Comm_rank(MPI_COMM_WORLD, &myid)`	retorna a identidade (rank) do processo
`MPI_Cart_create(MPI_COMM_WORLD,dims,dsize,wrap,reorder,cart)`	cria uma topologia cartesiana (cart) de dims dimensões, em que wrap diz se é fechada ou não e reorder diz se é possível reordenar os processos
`MPI_Cart_coords(cart, myrank, dims, coords)`	retorna as coordenadas numa topologia de dims dimensões (coords é um vetor)
`MPI_Cart_shift(cart, dim, displ, &src, &dst)`	retorna ranks dos vizinhos (a uma distância displ) na topologia
`MPI_Cart_get(cart, ndims, dims, periods, coords)`	recupera a topologia para as variáveis dims, periods e coords

Tabela 3.2. Principais funções de comunicação entre processos MPI

Função	Descrição
`MPI_Send(msg, length, type, id, tag, comm)`	envia msg para processo id
`MPI_Recv(msg, length, type, id, tag, comm, status)`	recebe msg do processo id
`MPI_Sendrecv(smsg, slength, stype, dest, stag, rmsg, rlength, rtype, source, rtag, comm, status)`	envia smsg para processo dest e recebe rmsg do processo source
`MPI_Send(msg, length, type, id, tag, comm, *request)`	envia msg para processo id sem bloquear o emissor, sinalizando a operação na variável request
`MPI_Recv(msg, length, type, id, tag, comm, status)`	recebe msg do processo id

Figura 3.1. Movimentos de dados para funções de comunicação coletiva.

	Buffer de envio				Buffer de recebimento			
	Proc1	Proc2	Proc3	Proc4	Proc1	Proc2	Proc3	Proc4
MPI_Bcast	a				a	a	a	a
MPI_Gather	a	b	c	d	a,b,c,d			
MPI_Allgather	a	b	c	d	a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
MPI_Scatter	a,b,c,d				a	b	c	d
MPI_Alltoall	a,b,c,d	e,f,g,h	i,j,k,l	m,n,o,p	a,e,i,m	b,f,j,n	c,g,k,o	d,h,l,p
Quando aplicável, considera-se que o processo Proc1 é o root								

MPI apresenta também funções de comunicação coletiva, apresentadas na Tabela 3.3. A mais simples delas é a de *broadcast*, `MPI_Bcast`, em que um dado processo envia um conjunto de dados para os demais processos. Nela o parâmetro `root` indica qual dos processos fará o envio aos demais. Além dela temos `MPI_Scatter` e `MPI_Gather`, que implementam funções complementares entre si. Em particular, `MPI_Scatter` faz a distribuição da mensagem a ser transmitida, de tamanho `cnt`, para os demais processos, cada um recebendo um pedaço de tamanho `rcnt`. Já `MPI_Gather` faz exatamente o contrário, isto é, junta pedaços de um *buffer* em um único *buffer* pelo processo `root`.

As duas últimas funções (Gather e Scatter) possuem as variantes `MPI_Alltoall` e `MPI_Allgather`, em que as comunicações ocorrem sem a necessidade de se definir um processo raiz. A figura 3.1, a seguir, ilustra como essas comunicações coletivas ocorrem.

Também na Tabela 3.3 temos funções que, além de disseminarem dados, fazem algum processamento sobre eles. Funções como `MPI_Reduce` e `MPI_Allreduce` executam uma operação definida sobre os dados retornados pelos processos. Essa operação pode ser de um dos tipos pré-definidos apresentados no quadro a seguir:

Operação	Significado
MPI_MAX	Máximo dos valores apresentados pelos processos
MPI_MIN	Mínimo dos valores apresentados pelos processos
MPI_SUM	Soma dos valores
MPI_PROD	Produto dos valores
MPI_LAND	E lógico
MPI_BAND	E *bitwise*
MPI_LOR	OU lógico
MPI_BOR	OU *bitwise*
MPI_LXOR	OU exclusivo lógico
MPI_BXOR	OU exclusivo *bitwise*
MPI_MAXLOC	Máximo e localização do máximo
MPI_MINLOC	Mínimo e localização do mínimo

Tabela 3.3. Principais funções de comunicação coletiva em MPI

Função	Descrição
`MPI_Bcast(msg, count, type, root, comm)`	Comunicação por broadcast, disparado pelo processo com rank root, devendo existir em todos os processos do grupo comm
`MPI_Scatter(msg, cnt, type, rmsg, rcnt, rtype, root, comm)`	envia um segmento (de tamanho cnt) de msg para cada processo em comm, que o recebe em rmsg
`MPI_Alltoall(msg, cnt, type, rmsg, rcnt, rtype, comm)`	todos processos enviam dados para todos ou outros processos
`MPI_Gather(msg, cnt, type, rmsg, rcnt, rtype, root, comm)`	recebe um segmento de tamanho cnt (msg) de cada processo e o coloca em rmsg
`MPI_Allgather(msg, cnt, type, rmsg, rcnt, rtype, comm)`	todos os processos recebem os dados de todos os outros processos, sem um processo *root*
`MPI_Reduce(operand,result,count,type,op,root,comm)`	faz como MPI_Gather, porém realizando uma operação (op) entre todos os dados enviados
`MPI_Allreduce(operand,result,count,type,op,comm)`	Comunicação por *broadcast*, disparado pelo processo com *rank root*, devendo existir em todos os processos do grupo *comm*

3.1.3. Funções miscelâneas

Além das funções mais fundamentais para comunicação e controle da execução dos processos paralelos, MPI oferece um conjunto bastante grande de funções, a maioria das quais não trataremos aqui. Isso inclui funções para sincronismo de processos, criação de grupos diferenciados de processos comunicadores, medição de tempo de execução, verificação dos canais de comunicação e até mesmo para comunicação unilateral. Algumas delas são descritas a seguir:

Barreira de sincronismo

`MPI_Barrier(group)` - em que processos pertencentes a *group* se sincronizam na barreira criada por essa função.

Verificação de canais de comunicação

`MPI_Probe(src, tag, comm. &status)` - que verifica se existe mensagem no buffer de entrada em que processos pertencentes a *group* se sincronizam na barreira criada por essa função. Observar que `MPI_Iprobe` faz a mesma operação, mas sem bloquear o processo durante a verificação.

Tratamento de tempo

`MPI_Wtime( )` - retorna o tempo atual em segundos
`MPI_Wtick( )` - retorna a precisão adotada por `MPI_Wtime`, isto é, qual o intervalo entre dois tiques do relógio

Comunicação unilateral

A comunicação unilateral permite que acessos remotos à memória, sem a necessidade de bloqueio durante a transferência de dados. Para tanto se cria uma janela de memória local que será disponibilizada para acesso remoto, usando funções específicas para transferir dados entre duas janelas definidas para acesso remoto. Apresentamos aqui apenas algumas funções, sem especificar os tipos de cada parâmetro, que podem ser facilmente encontradas nos manuais de Open MPI.

`MPI_Win_allocate(size, disp_unit, info, comm, &base, &win)` - cria uma região de memória, *win*, remotamente acessível (RMA window)

`MPI_Win_create(&base, size, disp_unit, info, comm, &win)` - define que a região *win* está exposta externamente a partir do ponteiro *base*

`MPI_Win_free(&base)` - libera a região exposta a partir do ponteiro *base*

`MPI_Put(*org_addr, org_count, org_dtype, tgt_rank, tgt_disp, tgt_count, tgt_dtype, win)` - move dados a partir de `org_addr` para a janela `tgt_rank`

`MPI_Get(*org_addr, org_count, org_dtype, tgt_rank, tgt_disp, tgt_count, tgt_dtype, win)` - move dados de `tgt_rank` para `org_addr`

3.2. Arquitetura Modular de Componentes

A Arquitetura Modular de Componentes, MCA, forma um sistema de arquivos de configuração, comandos e variáveis de ambiente, que propicia um ambiente eficiente de execução paralela com o Open MPI. A configuração de um sistema Open MPI é quase que totalmente feita pelo MCA, que forma basicamente a espinha dorsal das funcionalidades do Open MPI. O MCA é composto por projetos, *frameworks*, componentes e módulos, como descritos a seguir.

3.2.1. Projetos

Dentro da arquitetura do MCA projetos são entendidos como sendo o nível mais alto de código dentro do Open MPI, sendo definidos três projetos nessa categoria, que são:

- Open Portability Access Layer (OPAL): compreende código em baixo nível, para portabilidade de arquitetura e sistemas operacionais, sendo a camada do Open MPI mais próxima do sistema;
- Open MPI (OMPI): projeto compreendendo a API do MPI e sua infraestrutura de suporte;
- OpenSHMEM (OSHMEM): compreende a API OpenSHMEM, que suporta a manipulação de memória compartilhada para comunicação em RMA (comunicação unilateral no MPI), e sua infraestrutura de suporte.

Esses projetos formam camadas de gerenciamento do sistema. Em versões anteriores do Open MPI existia um quarto projeto, o Open MPI Runtime Environment - ORTE. O ORTE deixou de ser projeto interno para ser uma contribuição externa, que é o PMIX Runtime Reference Environment - PRRTE. As funções presentes nesse projeto cuidam do gerenciamento de processos e entrada/saída.

3.2.2. Frameworks

Os frameworks permitem gerenciar componentes específicos do MPI. Cada framework lançado em tempo de execução gerencia um tipo de componente. A lista completa de tipos de componentes manipulados pode ser obtida com a execução do comando `ompi_info`, sendo que os componentes mais usados incluem:

- Point-to-point Messaging Layer (PML), manipula componentes para implementar troca de mensagens ponto-a-ponto.
- Byte Transport Layer (BTL), usados como unidade de transporte para componentes PML.
- MPI collective algorithms (coll), para comunicação coletiva.

Figura 3.2. Saída parcial do comando ompi_info --param btl tcp --level 9

```
          MCA btl: tcp (MCA v2.1.0, API v3.1.0, Component v4.1.4)
      MCA btl tcp: ---------------------------------------------------
      MCA btl tcp: parameter "btl_tcp_links" (current value: "1", data source: default, level: 4 tuner/basic, type: unsigned_int)
      MCA btl tcp: parameter "btl_tcp_if_include" (current value: "", data source: default, level: 1 user/basic, type: string)
                   Comma-delimited list of devices and/or CIDR notation of networks to use for MPI communication (e.g.,
"eth0,192.168.0.0/16").  Mutually exclusive with btl_tcp_if_exclude.
      MCA btl tcp: parameter "btl_tcp_if_exclude" (current value: "127.0.0.1/8,sppp", data source: default, level: 1 user/basic,
type: string)
                   Comma-delimited list of devices and/or CIDR notation of networks to NOT use for MPI communication -- all devices
not matching these specifications will be used (e.g., "eth0,192.168.0.0/16").  If set to a non-default value, it is mutually
exclusive with btl_tcp_if_include.
      MCA btl tcp: parameter "btl_tcp_free_list_num" (current value: "8", data source: default, level: 5 tuner/detail, type: int)
      MCA btl tcp: parameter "btl_tcp_free_list_max" (current value: "-1", data source: default, level: 5 tuner/detail, type: int)
      MCA btl tcp: parameter "btl_tcp_free_list_inc" (current value: "32", data source: default, level: 5 tuner/detail, type: int)
      MCA btl tcp: parameter "btl_tcp_sndbuf" (current value: "0", data source: default, level: 4 tuner/basic, type: int)
                   The size of the send buffer socket option for each connection.  Modern TCP stacks generally are smarter than a
fixed size and in some situations setting a buffer size explicitly can actually lower performance.  0 means the tcp btl will not tr
to set a send buffer size.
```

- MPI I/O (io), que trata de entrada/saída paralela de arquivos em sistemas de arquivos distribuídos.
- MPI Matching Transport Layer (MTL), usado exclusivamente como suporte para transporte de componentes PML.

O uso do comando `ompi_info` permite identificar quais parâmetros estão disponíveis no sistema e seus valores atuais. Na Figura 3.2 pode se ver parte da saída de uma execução desse comando, com dados para comunicação TCP, destacando por exemplo `btl_tcp_sndbuf` para determinar o tamanho buffer de envio mensagens, sendo que valor 0 permite ao protocolo otimizar o processo de envio.

3.2.3. Componentes

Componentes MCA formam uma coleção de códigos que implementam plugins para interfaces específicas. Esses plugins podem ser adicionados ao código a ser executado tanto em tempo de execução quanto na compilação.

3.2.4. Módulos

São instâncias de um componente, sendo que podem existir várias instâncias de um mesmo componente controladas por um único framework. Exemplos de módulos envolvem, por exemplo, módulos de TCP ponto-a-ponto.

3.2.5. Definindo parâmetros MCA

Os módulos e componentes MCA podem ter seus parâmetros definidos de várias formas. Isso permite que administradores de sistema façam ajustes finos na instalação do Open MPI para um hardware ou ambiente específico. Esses parâmetros podem ser definidos na linha de comando, variáveis de ambiente, arquivos de *tuning* de parâmetros MCA ou arquivos de configuração geral. Veremos alguns desses parâmetros nas próximas seções, mais especificamente aqueles voltados para depuração e ajuste de aplicações.

Essa flexibilidade na definição de parâmetros permite que sejam modificados em mais de um momento. Por exemplo, o administrador do sistema pode configurar parâmetros otimizados mais genéricos com o uso de variáveis de ambiente, enquanto o usuário da aplicação pode definir parâmetros adaptados à sua aplicação com arquivos de *tuning*.

Outro aspecto a ser observado é que muitos dos parâmetros MCA não são de interesse de um usuário comum. Assim, foram definidos níveis de interesse, ou uso, de parâmetros. Foram criados nove níveis, separados em três categorias com três níveis cada. Essa categorização fica evidente para o comando *ompi_info*, que apresenta informações para parâmetros que estejam em níveis no máximo igual ao valor indicado em `--level` (valor padrão é 1). As categorias e níveis são as seguintes:

1. *End user* - envolvendo parâmetros que um usuário precisa definir para sua aplicação executar corretamente. São parâmetros dos níveis 1, 2 e 3.

2. *Application tuner* - tipicamente envolve parâmetros úteis no tuning da aplicação, ou seja, no controle de recursos como tamanho de buffers.

3. *Open MPI developer* - engloba parâmetros que não se encaixam nas categorias anteriores ou que sejam especificamente projetados para depuração ou desenvolvimento do Open MPI e não de uma aplicação.

Parâmetros de linha de comando

São parâmetros definidos no comando `mpirun`. A forma geral é dada por:

```
$mpirun --mca <param_name> <value> -np 4 a.out
```

Parâmetros definidos por variáveis de ambiente

Aqui os parâmetros podem ser definidos com comandos de exportação, para variáveis do tipo `OMPI_MCA_<param_name>`. A forma geral é dada por:

```
$export OMPI_MCA_<param_name>=<value>
```

Arquivos de *tuning* de parâmetros MCA

Arquivos para definição de parâmetros podem ser criados e lidos ao disparar uma aplicação MPI. Isso implica no uso da opção `--tune` para ler o arquivo de configuração. Por exemplo, se criarmos um arquivo denominado *foo.conf*, podemos disparar a aplicação com o comando:

```
$mpirun --tune foo.conf -np 4 a.out
```

O conteúdo de um arquivo de configuração consiste de uma ou mais linhas iniciadas por $-x$ ou $-mca$, como por exemplo:

```
-x envvar1=value1 -mca param1 value1 -x envvar2
-mca param2 value2
-x envvar3
```

Arquivos de configuração geral

Arquivos de configuração geral também pode ser usados para definir parâmetros, sem a necessidade de usar tags para essa definição. Esses arquivos podem ser encontrados em dois pontos:

```
$HOME/.openmpi/mca-params.conf
$prefix/etc/openmpi-mca-params.conf
```

3.2.6. Selecionando componentes para uso em tempo de execução

Cada um dos frameworks do MCA possuem um parâmetro que permite definir quais componentes serão usados em tempo de execução. Por exemplo, o framework BTL pode ter definidos quais protocolos serão usados durante a execução da aplicação MPI. Nesse caso é possível listar quais componentes serão usados ou quais não devem ser usados. Os exemplos a seguir mostram os casos de inclusão e exclusão respectivamente:

```
$mpirun --mca btl self,sm,usnic  ...
$mpirun --mca btl ^tcp,uct  ...
```

Deve ser observado que apenas uma lista de inclusão ou de exclusão pode ser usada, pois o significado dessa seleção é de que ou se especifica quais componentes serão usados, ou se especifica quais não serão usados (usando todos os não listados).

3.2.7. Alguns parâmetros relevantes

`rmaps_default_mapping_policy`, que define como aplicações serão executadas, que pode ter valores como *nolocal* (não executa no nó local), ou *hwtcpus* (usa threads de hardware como CPUs)

`pma`, que define qual módulo de comunicação ponto-a-ponto a ser usado, entre ob1, ucx, etc.

`use-hwthread-cpus`, que permite usar *hyperthreads* como recursos de binding.

3.3. Tuning de aplicações MPI

O tuning de uma aplicação Open MPI, isto é, ajustes em como ela usará a rede, memória, comunicação coletiva, escalonamento, etc., pode ser feito com a configuração de quais componentes devem ser adicionados à aplicação. Isso significa identificar os parâmetros dentro dos frameworks do MCA que devem ser incluídos e ajustados.

3.3.1. Ajustes de transporte

Como MPI executa em uma rede, uma parte importante no processo de tuning é ajustar a forma de transmissão dos dados entre os processos comunicantes. Isso envolve os componentes BTL, PML e parâmetros de rede/transporte.

Seleção de BTLs

O Open MPI suporta diferentes modos para mover dados entre processos, como:

- **tcp**, para comunicação baseada em TCP/IP em redes Ethernet ou IP.
- **openib, ofi, ucx** (dependendo da versão) para transportes de alto desempenho (InfiniBand, RoCE, etc.).
- **sm** para comunicação entre processos no mesmo nó (memória compartilhada).
- **self** para comunicação de um processo para ele mesmo (loopback interno).

Na maior parte das vezes o Open MPI já identifica quais recursos de rede estão disponíveis e usa os mais eficientes. Por exemplo, se existir uma interface Infiniband ou RDMA, o protocolo TCP será desativado automaticamente.

Para forçar explicitamente o uso de alguma forma de transporte, como tcp por exemplo, você pode fazer:

```
$mpirun --mca btl tcp,sm,self -np 8 a.out
```

Você pode ainda configurar o limite entre mensagens curtas, transmitidas de modo otimizado sem o *handshake*, e mensagens longas acrescentando, na linha de comando anterior, o parâmetro `--mca btl_tcp_eager_limit valor`.

Já quando falamos de redes Infiniband, o Open MPI fornece um framework mais moderno do que o `openib`, que é o Unified Communication X (UCX). A vantagem do UCX é que os vários padrões para comunicação unilateral (Infiniband, RoCE, NVlink, etc) podem ser tratados de forma unificada. Deve ser observado que quando incluímos o UCX devemos, em geral, excluir explicitamente o uso do TCP, da seguinte forma:

```
$mpirun --mca pml ucx --mca btl ^tcp -np 8 a.out
```

A simples especificação de uso do UCX não implica no ajuste desejado dos vários parâmetros de comunicação. Alguns dos parâmetros que podem ser ajustados são vistos na Tabela 3.4. Destaque-se, por exemplo, `UCX_RC_TX_QUEUE_LEN`, que permite ajustar filas de acordo com a carga de comunicação. A definição desses parâmetros pode ser feita em um dos modos indicados para componentes MCA como, por exemplo, na forma de variáveis de ambiente:

```
export UCX_TLS=rc,sm,self
export UCX_NET_DEVICES=mlx5_0:1
export UCX_IB_NUM_PATHS=2
```

Observe-se que para aplicações que usam mensagens grandes, como grandes blocos de dados, recomenda-se aumentar o número de WQEs e buffers RDMA.

Observação sobre memória e RDMA: o RDMA requer que as áreas de memória de envio/recepção sejam "registradas" no adaptador InfiniBand. O Linux, por padrão, impõe limites baixos ao registro de memória, MLock, o que causa erros como:

```
warning: RLIMIT_MEMLOCK is 64 KB, cannot register memory
```

Tabela 3.4. Parâmetros ajustáveis no framework UCX.

Parâmetro	Efeito	Observação
`UCX_TLS=rc,sm,self`	Seleciona "transport layers" (rc = reliable connected, sm = shared memory).	Evita uso desnecessário de TCP.
`UCX_RC_TX_QUEUE_LEN`	Define número de Work Queue Entries (WQEs) de envio/recepção.	Aumentar para workloads intensos.
`UCX_RC_VERBS_TX_MAX_WR`	Máximo de Work Requests pendentes.	Ajustar conforme hardware.
`UCX_NET_DEVICES=mlx5_0:1`	Seleciona a HCA (Host Channel Adapter) e porta.	Útil em nós com múltiplas interfaces.
`UCX_IB_NUM_PATHS`	Número de caminhos simultâneos (multirail).	Melhora throughput em topologias com várias HCAs.

Este limite pode ser ajustado no arquivo `/etc/security/limits.conf`, ajustando-se as variáveis *hard memlock* e *soft memlock*, permitindo então a criação e uso de buffers RDMA grandes. Podemos, por exemplo, remover totalmente esses limites fazendo:

```
* hard memlock unlimited
* soft memlock unlimited
```

Tuning de operações coletivas

Comunicações coletivas como broadcast, reduce, allreduce, scatter, gather, barrier, etc., são bastante usadas em aplicações paralelas, além de terem um impacto importante no desempenho. O Open MPI oferece um framework chamado `tuned`, que permite escolhas de algoritmo de roteamento, decisões dinâmicas e tuning fino para essas funções. A escolha começa com a definição do modo de decisão, que pode ser um desses três:

1. Decisão fixa (*Fixed decision*): é o modo padrão, em que o Open MPI usa uma "árvore de decisões" interna, baseando-se no tamanho da mensagem, número de processos, entre outros, para escolher um algoritmo de comunicação coletiva ótimo.
2. Algoritmo forçado (*Forced algorithm*): em que se força o uso de um algoritmo específico para aquela operação coletiva, ignorando a árvore de decisão. Essa escolha é útil se você já conhece qual algoritmo é melhor para seu caso.
3. Decisão dinâmica (*Dynamic decision*): em que se cria um arquivo de regras mapeando o tamanho de mensagem e comunicador para o algoritmo a usar. Isso permite decisões mais calibradas sob diferentes cenários de execução.

A ativação de um desses modos de decisão é feita usando os parâmetros MCA. Por exemplo, para ativar o modo dinâmico podemos inserir os seguintes parâmetros:

```
--mca coll_tuned_use_dynamic_rules 1
--mca coll_tuned_dynamic_rules_filename /path/arq.rules
```

Sendo que em `arq.rules` você especifica entradas para cada tipo de comunicação coletiva, como "`--mca coll_tuned_alltoall_algorithm 1`", em que 1 corresponde ao algoritmo linear. Isso faz com que a cada chamada de `MPI_Alltoall` se use o algoritmo linear, sem permitir a decisão por qualquer outro algoritmo.

Quando usamos o modo forçado, as definições MCA são da forma:

```
--mca coll_tuned_allreduce_algorithm 6
--mca coll_tuned_alltoall_algorithm 3
```

Os números indicados no final de cada linha correspondem a algoritmos específicos. A lista completa pode ser obtida com a execução de "*ompi_info –param coll tuned –level 9*". Por exemplo, `MPI_Allreduce` pode ter implementação linear (1), redução + broadcast(2), *recursive doubling* (3), *ring* (4), *segmented ring* (5), Rabenseifner (6). A identificação do melhor algoritmo depende de fatores como o número de processos, a distribuição de mensagens entre processos, tamanho das mensagens, largura de banda da rede, latência, congestionamento, topologia, etc. Por isso, os modos dinâmico e forçado são mais interessantes por permitirem escolhas mais otimizadas para cada aplicação.

Como as definições aqui podem ser feitas em vários momentos, ou até mesmo não feitas, definiu-se uma ordem de prioridade para qual algoritmo será escolhido. Assim, se a escolha é feita de modo forçado, esse algoritmo é adotado mesmo que existam definições dinâmicas para aquela função de comunicação. O modo de escolha usando árvores de decisão só é usado caso não existam definições forçadas e nem definições aplicáveis no modo dinâmico.

3.3.2. Afinidade e binding de processos / threads

Além dos ajustes na forma em que os processos se comunicarão, outro aspecto importante no tuning de uma aplicação é a definição de como os processos MPI e seus threads serão atribuídos aos diferentes nós e núcleos da rede. Isso permite minimizar latências e maximizar o uso de dados em cache e também a conectividade de rede.

Isso significa estabelecer critérios para os procedimentos de mapeamento (mapping) e binding entre processos e elementos de processamento. Esses procedimentos podem ser definidos da seguinte forma:

- Mapeamento define como os processos são distribuídos entre nós e sockets, o que pode ser feito por núcleo, por nó ou ainda por socket.
- Binding (ou affinity) define a qual núcleo ou conjunto de núcleos um processo MPI ficará associado, sem permitir sua migração entre núcleos.

Vale observar que em sistemas modernos, que possuem múltiplos sockets, núcleos e hierarquia mais ampla de memória, a migração de processos entre núcleos pode degradar o desempenho por quebrar a localidade de memória. Portanto, aplicar o binding de processos acaba sendo importante.

O Open MPI oferece opções de mapping/binding diretamente no mpirun. Por exemplo, se quisermos mapeamento pelo socket e binding para os núcleos, podemos executar:

```
mpirun --map-by socket --bind-to core ...
```

Mapeamento e binding em redes Infiniband

Em redes Infiniband as operações de mapeamento e binding são ainda mais sensíveis. Nelas se deve ter informações mais detalhadas sobre a topologia real da rede, a qual pode ser obtida com o comando `lstopo --of txt`. A partir dele se pode associar manualmente processos MPI a núcleos próximos ao adaptador IB (mlx5_0, mlx5_1 etc.) para evitar tráfego pela interconexão inter-socket.

Afinidade de threads com o uso de OpenMP ou similares

Se sua aplicação combina MPI e threads, com o uso de OpenMP, por exemplo, além de fixar os processos MPI a núcleos, é importante fixar onde threads vão executar. Nesse caso podemos, por exemplo, definir as seguintes variáveis de ambiente:

```
export OMP_PROC_BIND=true
export OMP_PLACES=cores
```

Essas definições ajudam a manter threads próximas ao processo MPI em um socket físico, evitando deslocamentos através da interconexão NUMA.

3.3.3. Metodologia para benchmarking e escolha de parâmetros

As escolhas descritas nas páginas anteriores apenas fazem sentido se conduzirem, de fato, a um melhor desempenho da aplicação. Fazer esses ajustes por tentativa e erro é, obviamente, um péssimo procedimento. Além das indicações mais particulares apresentadas a seguir, recomenda-se ao leitor o estudo de técnicas para avaliação e otimização de programas, que pode ser feito de modo resumido em [7].

1. Mensure uma linha de base: execute benchmarks simples (por exemplo, ping-pong entre pares, rodadas de MPI_Bcast, MPI_Allreduce, MPI_Alltoall) com configuração padrão.

2. Profiling/tracing: colete métricas de comunicação como latência, largura de banda, tempos em coletivas, congestionamento de rede, transferência de dados, etc.

3. Varie parâmetros MCA relevantes: altere sistematicamente parâmetros como limiares eager/rendezvous, tamanhos de buffer, número de conexões, parâmetros de coletivas (force/dynamic) etc.

4. Execute benchmarks de teste de carga real (sua aplicação ou similar) para ver impacto real.

5. Use o modo dinâmico / rules file para coletivas, gerando regras baseadas nos dados medidos. O repositório ompi-collectives-tuning provê scripts para coletar dados de coletivas e gerar arquivos de tuning.

6. Itere: faça ajustes de acordo com os resultados e repita as medições, observando melhorias ou pioras de desempenho. Parâmetros que otimizam para mensagens pequenas podem degradar para mensagens grandes, e vice-versa.

7. Registre e documente as melhores configurações para cada ambiente/aplicação para uso futuro.

3.3.4. Exemplos de linha de comando com tuning

Um exemplo de comando mpirun com vários ajustes, para uma rede convencional:

```
mpirun -np 64 \
  --map-by socket --bind-to core \
  --mca pml ob1 \
  --mca btl tcp,sm,self \
  --mca btl_tcp_eager_limit 65536 \
  --mca btl_tcp_max_send_size 131072 \
  --mca coll_tuned_use_dynamic_rules 1 \
  --mca coll_tuned_dynamic_rules_filename ~/tuned.rules \
  --mca coll_tuned_allreduce_algorithm 2 \
        ./meu_programa_mpi
```

Com a execução desse comando a aplicação aplica os seguintes ajustes:

- Fixação de binding / mapping (linha 1)
- Escolha explícita de transportes (linhas 2 e 3)
- Ajustes de buffer TCP (linhas 4 e 5)
- Ativação de modo dinâmico para coletivas, com regras externas (linhas 6 e 7)
- Forçar algoritmo específico para Allreduce (linha 8)

Já as definições de variáveis de ambiente e do comando *mpirun* a seguir, fazem ajustes considerando uma rede Infiniband:

```
export UCX_TLS=rc,sm,self
export UCX_NET_DEVICES=mlx5_0:1
export UCX_RC_TX_QUEUE_LEN=512
export UCX_RC_RX_QUEUE_LEN=512
export UCX_IB_NUM_PATHS=2
```

```
mpirun -np 64 \
  --map-by socket --bind-to core \
  --mca pml ucx \
  --mca coll_tuned_use_dynamic_rules 1 \
  --mca coll_tuned_dynamic_rules_filename ~/tuned.rules \
      ./simulacao_mpi
```

Com essa configuração garantimos:

- Uso dos pacotes UCX/InfiniBand para transporte RDMA (linhas de `EXPORT`)
- Afinidade de processos a sockets (linha 1 de mpirun)
- Aplicação de regras dinâmicas para comunicação coletiva (linhas 3 e 4 de mpirun)
- Aproveita múltiplos caminhos IB (último `EXPORT`)

3.3.5. Últimas observações sobre tuning

O Open MPI oferece um bom conjunto de mecanismos para tuning de desempenho. Esses mecacnismos podem ser aplicados tanto para otimizar os processos de comunicação como também a alocação entre núcleos e nós da rede a processos e threads da aplicação (mapeamento e afinidade). O uso correto dos mecanismos de tuning permite alcançar comunicações com latência mínima e throughput máximo. A correção envolve procedimentos de medição, análise e refinamento iterativos.

3.4. Depuração de aplicações MPI

Desenvolver aplicações MPI para clusters de alto desempenho frequentemente implica lidar com erros de difícil identificação, como o uso incorreto de handles MPI, vazamentos de memória, uso incorreto de datatypes, deadlocks, abortos silenciosos, problemas de comunicação etc. O processo de depuração da aplicação é feito, no Open MPI, de modos e com métodos diversos. Esses métodos envolvem o uso de parâmetros MCA, assim como ferramentas externas para perfilamento.

A depuração de aplicações MPI não envolve, portanto, a identificação de problemas relativamente simples de execução, como ocorre com programas sequenciais. Veremos na sequência como tratar os seguintes aspectos:

- verificações em tempo de execução da API MPI (valores de argumentos, uso correto de handles, etc),
- relatórios de problemas como vazamentos de objetos MPI,
- controle sobre abortos, mensagens de erro, pilha de chamadas,
- visualização do estado interno de parâmetros de execução (MCA),
- suporte a depuradores paralelos e ferramentas externas de *profiling/tracing*

Tabela 3.5. Exemplos de parâmetros MCA para depuração

Parâmetro	Descrição / utilidade
`mpi_param_check`	Quando tem valor positivo, verifica, em tempo de execução, se os parâmetros passados para chamadas MPI são válidos. Essa verificação busca por erros como passar um NULL onde não poderia. Permite capturar erros de uso da API cedo, embora implique em perda de desempenho
`mpi_show_handle_leaks`	Quando ativado, lista handles MPI (como comunicadores, tipos de dados, requisições) que foram criados e não foram liberados após a chamada de MPI_Finalize
`mpi_no_free_handles`	Pode ser usado em conjunto com `mpi_show_handle_leaks`, permitindo identificar tentativas de uso de handles anteriormente liberados ou ainda quem fez a liberação, uma vez que apenas "marca" o handle como liberado
`mpi_show_mca_params`	Exibe (quando a inicialização do MPI ocorre) todos os parâmetros MCA e seus valores que irão afetar a execução. Isso inclui os que vieram via ambiente, linha de comando ou arquivos de configuração. Permite ter visibilidade do ambiente MPI real
`mpi_show_mca_params_file`	Se `mpi_show_mca_params` estiver definido, escreve a lista de parâmetros MCA no arquivo indicado em vez de stderr ou saída padrão
`mpi_abort_delay`	Permite a conexão de um depurador ao processo que tenha abortado, pois força que se espere o valor definido em segundos (ou até uma intervenção manual se o valor negativo) antes do processo efetivamente sair
`mpi_abort_print_stack`	Imprime um stack trace (se o sistema suportar) no momento de MPI_Abort, permitindo localizar onde ocorreu a falha, especialmente em chamadas MPI ou em código associado a MPI

3.4.1. Parâmetros MCA de depuração

É possível ativar a depuração de aplicações MPI usando parâmetros MCA, tanto por sua chamada em *mpirun*, como por variáveis de ambiente ou arquivos de configuração (como vimos na seção sobre o MCA). Na Tabela 3.5 são apresentados alguns dos parâmetros que podem ser usados para depurar códigos Open MPI.

Além desses, há variáveis para emitir mensagens de aviso quando componentes MCA não carregam, ou para exibir mais informações gerais de configuração.

3.4.2. Ferramentas de depuração externas

Além das variáveis internas de depuração, o Open MPI 5.0 suporta integração com ferramentas externas, permitindo uma inspeção mais profunda, quando for o caso. O utilitário *ompi_info*, embora não faça exatamente o trabalho de depuração, permite ver quais componentes estão instalados, que versões, caminhos e quais parâmetros MCA são su-

portados. Permite, por exemplo, listar todos os parâmetros MCA de um componente específico ao especificar um nível alto para o mesmo, como feito no comando a seguir, para parâmetros de comunicação coletiva.

```
ompi_info --param coll tuned --level 9
```

Depuradores paralelos (TotalView, DDT, outros)

Depuradores paralelos comerciais, obviamente, produzem resultados bem mais detalhados do que se pode obter com os parâmetros de depuração MCA. Uma dessas ferramentas é o TotalView [8], que é uma ferramenta de depuração largamente usada em computação de alto desempenho.

O uso de TotalView para depuração com o Open MPI demanda o uso do módulo MPIR shim para permitir que o depurador "enxergue" os processos MPI de modo consistente. TotalView pode exibir filas de mensagem, requisições pendentes do MPI, comunicadores abertos, etc, permitindo verificar o progresso das comunicações.

Outro depurador que pode ser utilizado é o DDT [9], que assim como o TotalView permite gerar informações sobre filas de requisições, filas de mensagens, deadlocks, etc.

O uso dessas ferramentas é, em geral, bastante simples. Elas possuem interfaces gráficas para que se configure a aplicação MPI. O problema com as mesmas é que, a menos de versões para experimentação, seu custo é usualmente elevado, com licenças anuais superando os 25 mil dólares.

Em uma linha semelhante, podemos usar ferramentas como HPCToolkit [10], Tau [11] ou mpiP [12], que fazem medição de desempenho (perfiladores) como auxílio na identificação de gargalos na execução. Aspectos como latência de mensagens, ocupação de banda, etc., podem ser bastante úteis no processo de depuração do código.

3.4.3. Depuração de memória em aplicações Open MPI com Memchecker e Valgrind

Em aplicações paralelas MPI, erros de memória podem ser difíceis de detectar, pois são mascarados por concorrência, podem produzir falhas não determinísticas ou ainda envolver buffers compartilhados e comunicação entre processos. Valgrind e Memchecker formam um par de ferramentos que podemos usar para depurar eventuais erros de memória.

O Valgrind é um conjunto de ferramentas de instrumentação de código que executa o binário em um ambiente simulado, verificando acessos à memória e chamadas de sistema. Já o Memchecker é uma infraestrutura interna do Open MPI que usa o Valgrind Memcheck para detectar erros de acesso à memória.

Com essas ferramentas é possível detectar vazamentos de memória, uso após liberação, validar regiões de memória em uso e verificar se buffers usados para comunicação foram devidamente inicializados.

Quando o Open MPI é compilado com suporte à depuração e valgrind, o ambiente de execução insere verificações adicionais durante chamadas MPI. Isso permite, por

Tabela 3.6. Opções úteis para aplicação do Valgrind

Opção	Descrição
`--leak-check=full`	Reporta todos os vazamentos de memória, com pilha de chamadas.
`--show-leak-kinds=all`	Mostra diferentes tipos de vazamento (definite, indirect, possible).
`--track-origins=yes`	Indica a origem de valores não inicializados.
`--gen-suppressions=all`	Gera regras automáticas de supressão para bibliotecas de sistema.
`--log-file=valgrind.%p.log`	Cria um arquivo de log separado para cada processo MPI (identificado por PID).

exemplo, validar que um buffer enviado via MPI_Send contém dados válidos, ou ainda detectar escrita fora de limites de arrays em mensagens MPI.

O suporte ao Memchecker não está habilitado por padrão. É necessário recompilar o Open MPI com as opções adequadas, como visto nos comandos a seguir, em que as flags `--enable-debug --enable-memchecker` fazem a habilitação da depuração com o uso do Memchecker.

```
$ ./configure --enable-debug --enable-memchecker
$ make -j
$ sudo make install
```

Para fazer uso dessas ferramentas é necessário instalar Valgrind previamente, com o uso de *apt*, por exemplo. Para usar Valgrind em conjunto com mpirun, basta prefixar o comando de execução:

```
$ mpirun -np 4 valgrind --leak-check=full ./meu_progr
```

Para ganhar eficiência na depuração recomenda-se algumas opções específicas para o Valgrind, vistas na Tabela 3.6. Na execução do programa compilado dessa forma, cada processo MPI_COMM_WORLD criará um arquivo valgrind denominado `<pid>.log`, com o resultado das verificações.

Uso integrado com o Memchecker

Quando o Open MPI detecta que está sendo executado sob o Valgrind, ele ativa automaticamente as verificações de memória adicionais. Essas verificações são implementadas no código-fonte da biblioteca, a partir de parâmetros MCA de comunicação coletiva e ponto-a-ponto (como ompi_coll_tuned_*, ompi_request, etc.). Assim, erros como o exemplo abaixo podem ser capturados:

```
int buf[10];
    MPI_Bcast(buf, 20, MPI_INT, 0, MPI_COMM_WORLD);
```

Sendo que Valgrind/Memchecker reportará:

```
==12345== Invalid read of size 4
==12345==    at 0x4C31C2A: memcpy (vg_replace_strmem.c:1017)
==12345==    by 0x5120B15: ompi_coll_tuned_bcast_intra_generic
==12345==  Address 0x7f... is 8 bytes after a block of size 40 alloc'd
```

Indicando que o buffer (buf) é pequeno demais para conter os 20 inteiros enviados.

Podemos ainda verificar a inicialização de buffers MPI, identificando erros como na seguinte implementação, em que se envia um vetor parcialmente inicializado. Nesse caso o Memchecker (via Valgrind) detecta valores indefinidos sendo lidos por alguma função do MPI, permitindo observar erros lógicos em inicializações parciais.

```
double A[5];
    A[0] = 1.0;  // outros elementos não inicializados
    MPI_Send(A, 5, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
```

Saída esperada:

```
==12450== Conditional jump or move depends on uninitialised value(s)
==12450==    at 0x52091B2: ompi_datatype_copy_content_same_ddt
```

3.4.4. Uso prático e exemplos de depuração com Open MPI 5.0

A depuração usando parâmetros MCA pode ser feita de modo bastante simples, como veremos nos exemplos a seguir:

Identificar uso incorreto de handles ou tipos de dados inválidos

Se existe a suspeita de manipulação de um comunicador após sua liberação com MPI_Comm_free, podemos executar o programa com a chamada a seguir e, poderemos constatar se existem handles não liberados no relatório gerado.

```
mpirun -np 4 --mca mpi_param_check 1 \
  --mca mpi_show_handle_leaks 1 \
  ./meu_programa
```

Entender exatamente quais parâmetros de runtime estão sendo aplicados

Podemos verificar se o ambiente de execução do MPI está como esperado executando o programa com o comando:

```
mpirun -np 4 --mca mpi_show_mca_params 1 \
  --mca mpi_show_mca_params_file /tmp/mca_params.log \
  ./meu_programa
```

Isso produzirá um log em `/tmp/mca_params.log` com todos os parâmetros MCA (de ambiente, arquivo ou linha de comando) tal como o processo de rank 0 os vê.

Diagnóstico de abortos

Se o programa aborta silenciosamente ou você quer investigar onde exatamente aborta, podemos usar:

```
mpirun -np 4 --mca mpi_abort_print_stack 1 \
  --mca mpi_abort_delay 30 \
  ./meu_programa
```

Com esse comando, quando o programa invocar MPI_Abort, ele imprimirá, se possível, o conteúdo do stack trace e esperará por 30 segundos antes de realmente terminar. Isso permite que se conecte um depurador ao processo abortado usando seu PID e hostname que aparecem na mensagem indicativa do aborto.

3.4.5. Limitações, custos e impactos no desempenho

O processo de depuração do código é, obviamente importante, mas precisa ser feito com bastante cuidado. A inserção de mecanismos de acompanhamento da execução do programa gera custos adicionais ao sistema, como:

1. **Sobrecarga de desempenho:** verificações em tempo de execução, como feito com mpi_param_check, implicam custos adicionais (checar argumentos, sobrecarga de controle), degradando o tempo de execução.
2. **Uso de memória extra:** é necessário manter estruturas extras para depuração e logging, para que se possa exibir informações sobre handles não liberados, etc.
3. **Dependência do sistema e da plataforma:** alguns mecanismos de depuração, como stack trace, dependem do sistema operacional, de suporte de símbolos de debug, se o binário foi compilado com símbolos (-g), se não foi otimizado demais.
4. **Limitação da instalação do Open MPI:** nem todos componentes, ou níveis de parâmetros, têm suporte completo à depuração, o que inviabiliza a identificação de problemas mais sutis relativos a alguns transportes de rede ou componentes MCA, que podem não entregar visibilidade de requisições pendentes, por exemplo.
5. **Interferência:** em programas MPI com threads, ou com muitos processos, ativar muita informação ou registros pode gerar congestionamento de I/O, atrasos ou até mascarar erros por mudanças no tempo de execução.

3.4.6. Últimas observações sobre depuração

O Open MPI 5.0 oferece um bom conjunto de funcionalidades para depuração. Essas funcionalidades permitem verificar o uso correto da API MPI em tempo de execução, incluindo a detecção de vazamentos de handles MPI e geração de traços para determinação de abortos. Open MPI permite ainda o uso de depuradores e perfiladores de programas paralelos, aumentando as possibilidades de visualização de problemas. O custo do processo de depuração, em termos de desempenho, é admissível em fases anteriores à fase de produção, quando se deve desativar os mecanismos de depuração.

3.5. Conclusões e recomendações finais sobre tuning e depuração

Para finalizar, apresentamos aqui uma consolidação de boas práticas e cuidados a serem tomados, tanto no processo de tuning de uma aplicação quanto de sua depuração.

3.5.1. Recomendações para tuning

Boas práticas

Quando descrevemos a metodologia para tuning apontamos alguns procedimentos importantes, que devem ser tomados como boas práticas. Além deles temos outros pontos que devem ser considerados numa lista não exaustiva, como:

1. Tenha benchmarks para diferentes classes de comunicação (pequenas, médias, grandes mensagens).
2. Evite depender exclusivamente da árvore fixa padrão para coletivas — use forced/dynamic quando pertinente.
3. Considere a topologia da rede e o agrupamento de nós no cluster.
4. Fixe processos e threads para minimizar migrações e maximizar localidade de memória.
5. Ajuste o registro de memória e configurações do sistema operacional (mlock, limites de lock de memória) para RDMA/OpenFabrics.
6. Habilitar UCX (–mca pml ucx) quando operar com Infiniband, pois é mais eficiente e suportado em clusters modernos.
7. Ainda em Infiniband use `UCX_TLS=rc,sm,self`, evitando o uso de TCP.
8. Sempre que possível teste diferentes algoritmos de comunicação coletiva, pois diferentes algoritmos são ótimos em diferentes regimes de mensagem.
9. Ajuste os tamanhos dos buffers de entrada e saída, procurando melhorar o throughput em cargas intensas.
10. Use múltiplos caminhos aproveitando HCAs múltiplas e topologias Fat-Tree.

Armadilhas

Do mesmo modo que existem boas práticas a serem seguidas, existem práticas que devem ser evitadas. Elas são:

1. Evite executar mais processos MPI do que núcleos físicos disponíveis (oversubscription). Em muitos casos, o Open MPI faz busy-waiting (“spinning”) para checar progresso de comunicação, o que pode saturar a CPU e provocar queda drástica de desempenho se houver oversubscription.

2. Não esqueça de habilitar os modos BTL *sm / self* BTLs quando customizar os modos de transporte de dados (btl).

3. Valores exagerados de buffer podem gerar uso excessivo de memória ou fragmentação.

4. Garanta que os limites de registro de memória, principalmente para RDMA, sejam adequados, pois limites baixos podem causar erros de "*memory registration*".

5. Tenha cuidado com a escolha de algoritmos, pois forçar algoritmos inadequados para um cenário específico pode piorar o desempenho.

3.5.2. Recomendações para depuração

Como já indicado, a depuração é um processo importante no desenvolvimento de aplicações MPI. Algumas recomendações finais possíveis incluem:

1. Compile com símbolos de depuração (-g) e sem remover informação de pilha, para que stack trace ou mensagens de erro sejam legíveis.

2. Testes locais pequenos primeiro, ou seja, use poucos nós/processos para reproduzir o erro, pois depurar localmente facilita a inspeção com ferramentas como gdb.

3. Habilite verificações de parâmetros MCA (`mpi_param_check`) para capturar erros de uso desde o início, nas fases iniciais de desenvolvimento.

4. Use `mpi_show_handle_leaks` para detectar vazamentos de objetos MPI — isso é importante em aplicações de longa execução ou que criam muitos comunicadores ou datatypes.

5. Documente quais parâmetros MCA foram usados — use `mpi_show_mca_params` e grave os *logs*. Isso facilita a comparação entre múltiplas execuções.

6. Combine a depuração com ferramentas externas — depuradores paralelos, tracers, profilers — para verificar também o comportamento de comunicação, não apenas erros.

7. Reduza a sobrecarga de trabalho com a aplicação em produção, desativando verificações e registros pesados quando tiver certeza que o código funciona, evitando penalidades de desempenho.

Referências

[1] FAUSEY, M. R. CPS and the Fermilab farms. In: FERMI NATIONAL ACCELERATOR LAB., BATAVIA, IL (UNITED STATES). 1992. Disponível em: <https://www.osti.gov/biblio/6946034>.

[2] CARRIERO, N.; GELERNTER, D. Linda in context. *Communications of the ACM*, ACM New York, NY, USA, v. 32, n. 4, p. 444–458, 1989.

[3] GEIST, A. et al. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. Cambridge, MA, USA: MIT Press, 1995. ISBN 0262571080.

[4] GROPP, W.; LUSK, E. Sowing mpich: a case study in the dissemination of a portable environment for parallel scientific computing. *The International Journal of Supercomputer Applications and High Performance Computing*, v. 11, n. 2, p. 103–114, 1997.

[5] The Open MPI Community. *Open MPI v5.0.x*. 2025. `https://docs.open-mpi.org/en/v5.0.x/`. Accessado em julho de 2025.

[6] MICROSOFT. *MS - MPI v10.1.3*. 2025. Disponível em: <https://learn.microsoft.com/pt-br/message-passing-interface/microsoft-mpi>.

[7] MANACERO, A. Técnicas para análise e otimização de programas. In: *Minicursos do SSCAD 2024*. [S.l.]: SBC, 2024. cap. 6, p. 23.

[8] Perforce. *TotalView*. 2025. https://www.perforce.com/products/totalview. Acessada em outubro de 2025.

[9] Linaro Limited. *DDT - Distributed Debugging Tool*. 2025. Disponível em: <https://www.linaroforge.com/linaro-ddt>.

[10] HPCToolkit Project. *HPCToolkit*. 2025. Acessada em outubro de 2005. Disponível em: <https://hpctoolkit.org/index.html>.

[11] PERFORMANCE RESEARCH LAB. *TAU - Tuning and Analysis Utilities*. 2025. `https://www.cs.uoregon.edu/research/tau/home.php`. Acessada em outubro de 2025.

[12] LLNL. *LLNL/mpiP: A lightweight MPI profiler*. 2025. https://github.com/LLNL/mpiP. Acessada em outubro de 2025.