Ensino de Software Pipelining e Escalonamento em GPUs com Python no Google Colab

Ricardo Ferreira , José Augusto M. Nacif
Universidade Federal de Viçosa, Brasil
ricardo@ufv.br

***Resumo*—O ensino de escalonamento estático e dinâmico de instruções em processadores com pipeline é amplamente documentado em livros didáticos. No entanto, há poucas informações disponíveis sobre técnicas mais avançadas de escalonamento, como o Software Pipelining e tópicos mais recentes, como o escalonamento de Warp em GPUs. Este artigo apresenta uma experiência de ensino que utiliza o Google Colab e Python para gerar exemplos e documentação sobre o ensino de Software Pipelining e escalonamento em GPUs. O trabalho foi desenvolvido em conjunto com os alunos da disciplina de Arquitetura de Computadores da UFV, incluindo a resolução de problemas e a criação de ferramentas de visualização como resultados.**

***Index Terms*—Google Colab, Software Pipelining, Escalonamento, GPU**

I. Introdução

O desenvolvimento dos processadores com pipeline foi fundamental na evolução das arquiteturas de computadores. A técnica do pipeline permite a execução simultânea de múltiplas instruções, otimizando significativamente o desempenho e a eficiência dos processadores. Nesse contexto, o escalonamento de instruções desempenha um papel importante na mitigação dos atrasos no pipeline, reduzindo o tempo de espera devido a dependências de dados e controle entre as instruções.

O escalonamento de instruções pode ser classificado em duas principais categorias: escalonamento estático e escalonamento dinâmico. O primeiro envolve a reorganização das instruções em tempo de compilação ou pelo programador. Em geral é aplicado a trechos de laços que dominam o tempo de execução dos programas. O escalonamento dinâmico, não requer que o programador ou o compilador tenham conhecimento da arquitetura, pois o próprio pipeline incorpora recursos em hardware para reordenar as instruções em tempo de execução e resolver as dependências de dados. Ambas as técnicas são amplamente estudadas e documentadas em livros didáticos [18], [20], formando a base do conhecimento para a maioria dos cursos de arquitetura de computadores.

No entanto, existe pouca literatura didática para o ensino de técnicas de escalonamento estático mais avançadas, como o *software pipelining*, e tópicos emergentes, como o escalonamento de Warp em GPUs (Graphics Processing Units). O *software pipelining* é uma técnica que visa explorar ainda mais o paralelismo a nível do laço, permitindo a execução de múltiplos laços em paralelo, resultando em maior utilização do pipeline e ganhos significativos de desempenho. Entretanto, os alunos tem muita dificuldade em visualizar a execução sobreposta das iterações diferentes. Não existem simuladores didáticos para o ensino de *software pipelining*. Além disso, modificar um laço para com escalonamento com *software pipelining* é uma tarefa com um certo grau de dificuldade para a maioria dos estudantes.

Além disso, temas mais recentes como o escalonamento de Warp em GPUs não possuem literatura didática disponível. Apesar de recentes na história da computação, as GPUs já podem ser consideradas uma arquitetura clássica. O tema começou a fazer partes dos livros didáticos [18]. Mais recentemente, a demanda foi impulsionada pelo processamento paralelo em aplicações de alto desempenho, como inteligência artificial, processamento de imagens e simulações complexas.

A complexidade dessas abordagens e a rápida evolução das tecnologias de processamento muitas vezes resultam em uma escassez de recursos educacionais detalhados e acessíveis. Essa lacuna no ensino pode dificultar o desenvolvimento de competências essenciais em arquitetura de computadores.

Neste contexto, este artigo propõe uma abordagem inovadora para o ensino de *software pipelining* e escalonamento de *warp* em GPUs. Utilizando a plataforma Google Colab e a linguagem de programação Python para modelar os códigos em assembly, buscamos oferecer exemplos práticos, documentação detalhada e ferramentas de visualização que tornam esses tópicos mais acessíveis e compreensíveis para estudantes e profissionais interessados em aprofundar seus conhecimentos em arquitetura de computadores e técnicas de otimização. Além disso, o material foi produzido com a colaboração dos estudantes ao realizar trabalhos práticos na disciplina de arquitetura de computadores da Universidade Federal de Viçosa, UFV.

Os resultados desta pesquisa têm como objetivo ampliar a disponibilidade de material educacional sobre esses temas. Além de incentivar os estudantes a participarem como colaboradores, capacitando-os para enfrentar os novos desafios da computação de alto desempenho. Ao compartilhar essa experiência de ensino e oferecer recursos educacionais abertos, esperamos contribuir para o avanço do conhecimento em arquitetura de computadores e promover a disseminação do aprendizado na área.

Ao longo das próximas seções, apresentaremos em detalhes a abordagem utilizada para o ensino de *software pipelining* e escalonamento de *warp* em GPUs, discutindo os desafios enfrentados, os resultados obtidos e as perspectivas para futuras pesquisas e aprimoramentos nessa área.

II. Trabalhos Relacionados

Apesar da disponibilidade de muitos simuladores e muitas ferramentas *online* em navegadores de internet para ensino de arquitetura de computadores [10], [16], não existem ferramentas que dão suporte para o ensino de *software pipelining* e escalonamento de instruções em GPUs.

Dentre os trabalhos mais próximos, podemos destacar técnicas para o ensino de escalonamento dinâmico como o Tomasulo com simuladores na web [21] ou versões em aplicativos para celulares [15]. Para o ensino de processadores MIPS ou RISC-V, abordando temas genéricos como o ensino de assembly e pipeline básico ou temas mais específicos como predição de desvio e extensões vetoriais, podemos destacar:

1) Simulador MARS para Assembly MIPS [23], desenvolvido em Java para instalação local;
2) Projeto de um preditor de Desvio para o processador MIPS [3], implementado em Java com instalação local;
3) Ensino da Extensão Vetorial do MIPS com o simulador MARS [1];
4) Ensino de projeto em Verilog do processador MIPS com pipeline 5 estágios [17], executa no navegador;
5) Simulador Web Risc-V [12] para ensino de Assembly e Pipeline 5 estágios, executa no navegador;
6) Simulador Venus para Assembly Risc-V [22], executa no navegador;
7) Extensões Vetoriais com simulação no Gem5 do processador Risc-V [19], instalação local;

Para o ensino de GPU existem poucos trabalhos. O trabalho [14] apresenta uma avaliação com *microbenchmarks* de diversos aspectos das arquiteturas de GPUs como memórias, escalonamento, latências das instruções dentre outros pontos em detalhes. Sugestões para o ensino e pesquisa das arquiteturas de GPU com foco nos vários tipos de memórias, no assembly PTX e na latências de instruções foram propostas em [2], [8], [9].

III. Software Pipelining

A técnica de *software pipelining* evita que o código do laço seja replicado, ao intercalar a execução de múltiplos laços. Entretanto, os alunos têm muita dificuldade em visualizar a sobreposição e entender o motivo para 'inverter o código'. O material produzido para o ensino do *software pipelining* envolveu inicialmente três problemas e dois exemplos como base. Toda a tarefa foi executada dentro do Google Colab [4], [5], [7], que, por ser um notebook computacional, permite a integração de texto, código, figuras, entre outros, em um único ambiente. O uso de Python também é outro facilitador importante para a modelagem da técnica e simulação, permitindo o desenvolvimento de exemplos simples e de fácil compreensão para verificar o escalonamento.

Primeiro, apresentaremos o exemplo de um laço simples, seguido da sua versão com laço desenrolado e, em seguida, a implementação com *software pipelining* na seção III-A. Depois, apresentamos os três problemas que foram resolvidos como trabalho em grupo pelos alunos na seção III-B. Finalmente, a seção III-C apresenta um visualizador de grafos desenvolvido por um grupo de alunos para auxiliar no ensino de escalonamento.

A. Exemplo de Software Pipelining

O primeiro exemplo é bastante simples e implementa um laço em assembly RISC para executar a operação $v_i = (v_i + v_i)^2$ sobre o vetor v. A Figura 1 apresenta o código em assembly.

```
LOOP:
  Ld f3,0(r1)    # Elemento Vi - Registrador f3
  add f3,f3,f3   #  vi + vi
  mult f3,f3,f3 # (vi+vi)^2
  sd f3,0(r1) # vi = (vi+vi)^2
  Addi r1,r1,4 # incrementa o indice i
  Beq r1,r2,LOOP  # retorna ao inicio do laco
```

Figura 1. Código em Assembly RISC para calcular $v_i = (v_i + v_i)^2$

Em seguida, demonstramos como simular este código em Python, conforme ilustrado na Figura 2. A memória é representada como uma lista ou array para leitura dos dados. O código é implementado com uma instrução por linha. Para facilitar os testes, alimentamos o vetor ou lista com valores facilmente verificáveis. Neste exemplo, utilizamos uma sequência de inteiros $0, 1, \ldots, i, i+1, \ldots$. Além disso, para simplificar ainda mais, o controle do laço é implicitamente tratado usando o comando *for*. Ao final deste primeiro exemplo, a lista ou vetor de saída é impressa para fins de visualização.

```
l = [0,1,2,...,29,30]
for i in range(20):
   f3 = l[i]      # load
   f3 = f3+f3     # add
   f3 = f3*f3     # mult
   l[i] = f3      # store

print(l)
# Saida esperada
# [0, 4, 16, 36, 64, 100,...
```

Figura 2. Código em Assembly RISC modelado em Python.

Este exemplo serve para ilustrar a sintaxe e não apresenta dificuldades para os alunos compreenderem o mapeamento entre o assembly e a linguagem Python.

Tabela I
Execução do Código para visualizar as dependências entre as instruções

Código	F	D	E	W	F	D	E	W
Ld f3,0(r1)	1	2	34	5	15	16	1718	19
add f3,f3,f3	2	34	5-7	8	16	1718	19-21	22
mult f3,f3,f3	34	567	8-12	13				
sd f3,0(r1)	567	8-12	1314					
Addi r1,r1,8	8-12	13	14	15				
Beq r1,r2	13	14						
LOOP								

Em seguida, usando a linguagem *markdown* [13], uma tabela mostra a execução do código considerando 2, 3 e 5 ciclos para execução do load/store, add, e mult, respectivamente. Para este rastreio de execução consideramos que o processador não possui escalonamento dinâmico e os bloqueios são realizados no estágio de decodificação do pipeline. A tabela I mostra o rastreio para nosso exemplo.

O próximo passo é mostrar o grafo de dependência do código. Usamos o *graphviz* [6] que já está integrado no Python e Colab. A Figura 3 ilustra o código *graphviz* e a Figura 4 mostra a visualização gerada. O grafo de dependência é um importante passo no ensino do *software pipelining* e o escalonamento estático de uma forma geral.

```
from graphviz import Digraph
codigo = Digraph()
codigo.graph_attr['rankdir'] = 'LR'
codigo.node('1','Ld F3')
codigo.node('2','add f3,f3,f3')
codigo.node('3','mult f3,f3,f3')
codigo.node('4','sd f3')
codigo.edge('1','2','f3')
codigo.edge('2','3','f3')
codigo.edge('3','4','f3')
codigo
```

Figura 3. Descrição Textual do Grafo de dependência em Graphviz/Python.

Figura 4. Grafo de Dependência de dados entre as instruções.

A definição e a técnica de *software pipelining* são apresentadas na aula teórica. A Figura 5 ilustra a sobreposição de várias iterações. Neste exemplo, o grafo possui 4 níveis, o que requer a sobreposição de 4 iterações do laço. Esse é o ponto mais importante no ensino da técnica, onde diversos aspectos devem ser reforçados. A simulação quantitativa em Python enfatiza a importância de cada passo nesse processo.

Primeiramente, a técnica de *software pipelining* sobrepõe as iterações. Notamos que o novo laço começa com a instrução de store da primeira iteração (em azul), seguida pela instrução de multiplicação da segunda iteração (em verde), depois a instrução de adição da terceira iteração (em roxo) e, finalmente, o load da quarta iteração (em negrito). Em outras palavras, o código é escrito de "trás para frente", começando pela última instrução da primeira iteração e terminando com a primeira instrução da quarta iteração.

Outros pontos importantes que devem ser destacados aqui são os seguintes: antes do laço do *software pipelining*, é necessário executar o código de preâmbulo que, neste exemplo, inclui 3 instruções da primeira iteração, duas instruções da segunda iteração e uma instrução da terceira iteração (destacadas com uma linha pontilhada na Figura 5). A simulação permite adicionar e remover o trecho do preâmbulo e verificar os erros introduzidos sem o preâmbulo. O mesmo ocorre para o epílogo que não está ilustrado no exemplo, mas deve ser adicionado para gerar a execução corretas das últimas iterações.

Além disso, os registradores precisam ser renomeados, também destacados em vermelho na Figura 5. Embora nosso exemplo use apenas o registrador $f3$, para repassar as informações entre as iterações, serão necessários mais 2 registradores. Utilizaremos $f4$ para repassar o valor do add para o $mult$ e $f5$ para repassar o valor do $load$ para o add. O trecho de código pode ser executado com ou sem a renomeação de registradores, como forma de demonstrar a sua necessidade.

Finalmente, a Figura 6 apresenta o código do laço com o *software pipelining* e também o trecho do preâmbulo. É importante observar que o *load* da quarta iteração trabalha com o elemento $i+3$, pois está 3 iterações à frente no laço. A impressão da lista mostrará a execução correta.

Outro aspecto é usar a verificação automática com o código em alto nível comparado com a execução com o *software pipelining* ilustrado na Figura 7.

O segundo exemplo já envolve um aspecto de temporização entre a sobreposição das iterações que pode passar despercebido. Usamos a mesma ideia de um cálculo com vetor onde $v[i] = (v[i] * v[i] + v[i])$. A Figura 8 ilustra o código assembly inicial, sem aplicar a técnica de *software pipelining*.

Seguindo o fluxo, o próximo passo é construir o grafo de dependência, onde fica claro que existem duas arestas para o registrador $f3$ que carregam o valor inicial do vetor gerado pelo comando *load*. O detalhe é que uma das arestas atravessa dois níveis. Portanto ao sobrepor iterações para fazer o *software pipelining* irá gerar resultados inconsistentes se o grafo não for balanceado. A Figura 9 mostra em destaque com a cor vermelha que a aresta de $f3$ que liga o $load$ no add passa por dois "laços". Para resolver o problema, neste exemplo, além da renomeação de $f4$, precisamos de um registrador temporário para $f3$. Apesar de termos 4 iterações sobrepostas, 5 instruções serão necessárias com a adição de um "move"para preservar o valor de $f3$ de duas iterações anteriores.

A Figura 10 mostra o código final para o *software pipelining* do segundo exemplo que já inclui o balanceamento de registradores. Neste exemplo, usamos o registrador $f6$ para propagar o valor de $f3$ entre duas iterações.

B. Problemas

As tarefas atribuídas aos estudantes foram três trechos de código. Para cada código em alto nível, uma versão em assembly "python"deve ser gerada. Para depois, construir o grafo de dependência e posteriormente aplicar a técnica de *software pipelining*.

O primeiro problema apresenta um código em alto nível para ser mapeado em assembly. O código é bem simples pois envolve apenas uma atribuição de vetor $v[i] = ((2 * v[i] + v[i]) * v[i])$. Ao fazer o grafo, o primeiro desafio é o balanceamento dos registradores, para depois implementar a técnica de *software pipelining* como ilustrado na Figura 11(a). Podemos observar que o registrador $f3$, que também é usado para receber o $load$ do vetor, tem três arestas repassando o valor para o próximo nível, o nível seguinte e um terceiro

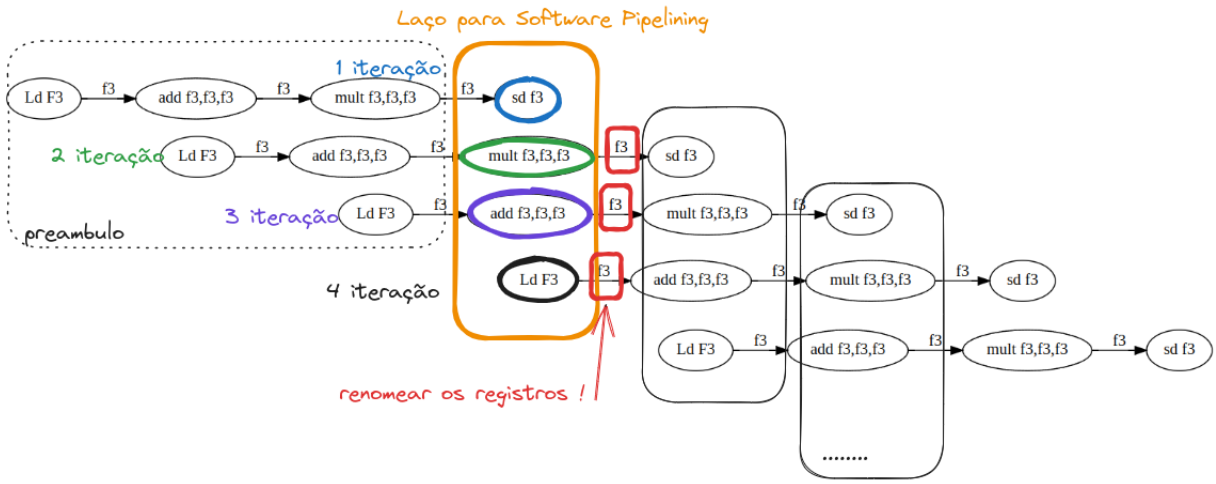

Figura 5. Técnica de Software Pipelining com a sobreposição das iterações.

```
l = [0,1,2,...]
# preambulo
f5= l[0]   # load 1 it
f4 = f5+f5 # add 1 it
f3= f4*f4  # mul 1 it
f5= l[1]   # load 2 it
f4 = f5+f5 # add 2 it
f5= l[2]   # load 3 it
for i in range(15): # laco
   l[i] = f3  # sd f3
   f3 = f4*f4 # mul f3,f4,f4
   f4 = f5+f5 # add f4,f5,f5
   f5 = l[i+3]  # ld f5,"12"(r1)

print(l)
```

Figura 6. Implementação em Python do laço com software pipelining que inclui o preâmbulo.

```
l = [0,1,2,...]
copia = l[:]
for i in range(15):  # alto nivel
  copia[i] = (copia[i]+copia[i])**2
# preambulo
f5= l[0]
....
for i in range(15): #laco
   l[i] = f3  # sd f3
    ...
   f5 = l[i+3]  # ld f5,"12"(r1)

#verificacao
for i in range(15):
  if copia[i] != l[i]:
    print("Erro na execução !")
```

Figura 7. Verifica a versão com Software Pipelining com o código em alto nível.

nível após. Serão necessários pelo menos dois registradores, pois um registrador pode ser reusado.

O segundo problema envolve a manipulação de dois elementos do vetor por vez, como ilustrado pelo código em alto nível:

```
For i=0; i < N; i+=2
  v[i] = v[i]+2
  v[i+1] = v[i+1]*2 + v[i+1]
```

As atribuições dos elementos pares e ímpares são diferentes e independentes. O ponto a ser trabalhado aqui é qual será o número de iterações que serão sobrepostas. Pois para os elementos pares, apenas uma adição é executada $v[i] = v[i]+2$. Já para os elementos ímpares, são executadas duas operações, uma adição e uma multiplicação. Figura 11(b) mostra que o grafo de dependência é composto por dois subgrafos desconexos. Um subgrafo com três níveis para os elementos pares e outro com quatro níveis para os elementos ímpares. O *software pipelining* tem que ser realizado com a sobreposição de 4

```
l = [0,1,2,...]
for i in range(20):
   f3 = l[i]
   f4 = f3*f3
   f4 = f4+f3
   l[i] = f4
```

Figura 8. Segundo Exemplo: Código Assembly sem software pipelining

iterações.

O terceiro problema também trabalha com 2 elementos do vetor por vez, mas existe uma dependência entre eles. O código em alto nível é:

```
For i=0; i < N; i+=2
  v[i] = (v[i]+2)*v[i+1]
  v[i+1] = v[i+1]*2 + v[i]
```

A Figura 12 mostra o grafo de dependência de uma implementação em assembly. Podemos observar que existem

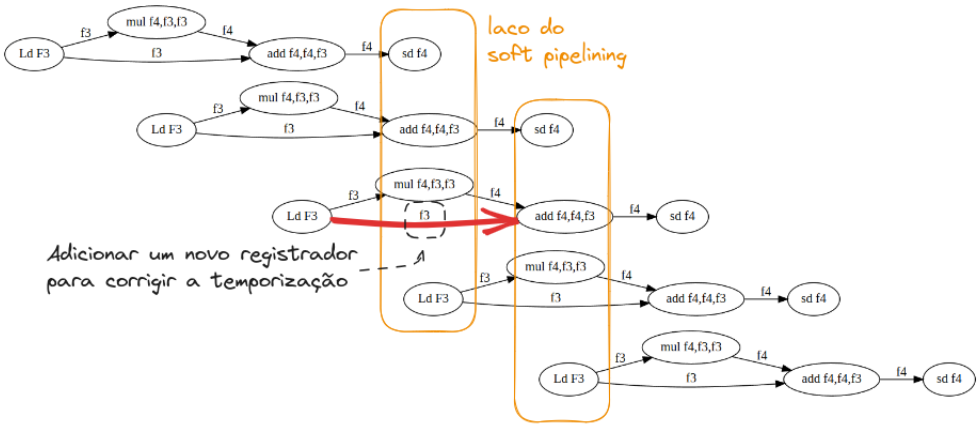

Figura 9. Técnica de Software Pipelining com a sobreposição das iterações.e balanceamento do registrador $f3$ para o segundo exemplo de código.

```
l = [0,1,2,...]
# preambulo
f3 = l[0]
f5 = f3*f3
f4 = f5+f3 # 1
f3 = l[1]
f5 = f3*f3
f6 =  f3 #2
f3 = l[2] # 3
for i in range(20): # laco
   l[i] = f4  # SD
   f4 = f5+f6 # add f4,f5,f3
   f5 = f3*f3 # mul f5,f3,f3
   f6 = f3  # balanceamento
   f3 = l[i+3]  # ld f3
```

Figura 10. Segundo Exemplo: Código Assembly com software pipelining e balanceamento

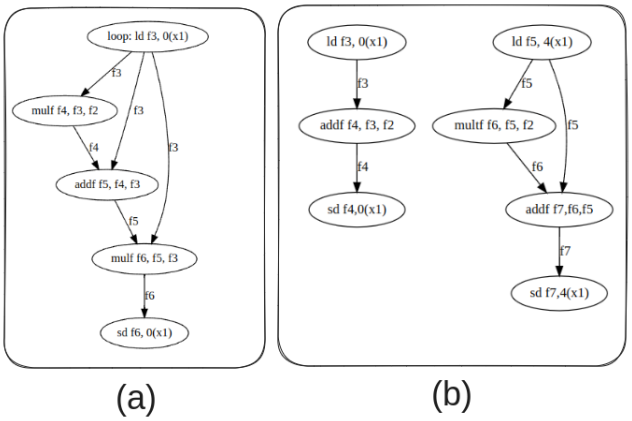

Figura 11. (a) Grafo de Dependência para problema 1; (b) Grafo de Dependência para o problema 2.

dois fluxos que estão interligados. A sobreposição deve considerar 5 iterações consecutivas observando a profundidade do grafo.

Figura 12. Grafo de Dependência para problema 3

C. Ferramenta de Geração do Grafo

Além da resolução dos problemas, os alunos foram incentivados a criar outras ferramentas para auxiliar no ensino de *software pipelining* dentro do ambiente Google Colab. Um dos grupos implementou um *parser* de um pseudo assembly para geração do grafo de dependência. A entrada deve respeitar a seguinte sintaxe:

```
instructions = (
    'ld f1, 0(x1)',
    'addf f1, f1, f2',
    'ld f3, 4(x1)',
    'mul f1, f1, f3',
    'sd f1, 0(x1)',
    'mul f3, f3, f2',
    'addf f3, f3, f1',
    'sd f3, 4(x1)'
)
```

A entrada é uma definição de um tupla em Python. Cada elemento da tupla é uma instrução no formato de um *string*. O código do *parser* foi feito em Python. A versão proposta inclui as instruções de load, store, multiplicação, adição com e sem imediato e desvios. Mais especificamente os mnemônicos: ld, sd, adi, beq, bne. Qualquer outro mnemônico que não esteja incluído nesta lista como add, mul, dentre outros, é considerado um operador binário com um registrador de destino seguido de dois registradores fonte, como add f1,f2,f3, por exemplo.

D. Considerações Finais

As soluções dos estudantes estão disponíveis para consulta no Colab Ìndice (clique aqui) de acesso público. O desempenho dos estudantes não foi aferido na prova em relação aos anos anteriores de forma quantitativa, mas de forma qualitativa, o desempenho foi bem superior e auxiliou a compreensão de temas correlatos como outras técnicas de escalonamento. O uso da linguagem Python foi unânime em uma consulta informal aos estudantes, nenhum deles sugeriu outra linguagem para solução dos problemas.

O tema *software pipelining* mesmo sem balanceamento e códigos mais complexos já é um desafio por si só. Porém com o auxílio da implementação e da verificação, foi possível avançar e ilustrar vários outros detalhes. Este pontos não são discutidos nos materiais didáticos disponíveis para o assunto.

Para o próximo oferecimento da disciplina de arquitetura de computadores na UFV, além dos 2 exemplos e dos 3 problemas já resolvidos, serão disponibilizados novos problemas e os alunos serão estimulados ao desenvolvimento de novas ferramentas para o ensino de *software pipelining*. Apesar dos resultados promissores, alguns alunos ainda tem dificuldades no entendimento da técnica.

IV. Escalonamento em GPU

O conhecimento da microarquitetura auxilia os desenvolvedores de GPU na otimização do desempenho de suas aplicações. Como o tempo de execução de cada núcleo influencia o desempenho global, o programador deve se preocupar com a eficiência de cada instrução ao escrever código de alto desempenho.

Para ensino das arquiteturas de GPU, poucos livros [18] abordam o assunto. O foco é centrado no modelo SIMT (Single Instruction, Multiple Threads) é uma arquitetura permite a execução de múltiplas threads simultaneamente, seguindo uma única instrução (mesmo código), mas em diferentes conjuntos de dados. Os conceitos básicos são:

SMX : (Streaming Multiprocessor) é uma unidade multiprocessada da arquitetura das GPUs da Nvidia. Também é conhecido como Streaming Processor ou Multiprocessador. Cada SMX pode executar 1 ou mais bloco de threads. O SMX é composto por um número específico de núcleos CUDA (Compute Unified Device Architecture), ou núcleos são equivalentes a unidade lógica aritmética sendo capazes de executar uma operação simples como uma soma, multiplicação ou leitura/escrita em memória. O SMX possui memória compartilhada, que é usada para armazenar dados intermediários, permitindo uma comunicação rápida entre os threads de um mesmo bloco. Além disso, o SMX possui um banco de registradores que chega a ser 4 vezes maior que a memória compartilhada e a cache L1.

Warps : O SMX executa instruções em sub-grupos chamadas ”Warps”. Um Warp é um grupo de threads (normalmente 32 threads) que são executadas ao mesmo tempo em um SMX. Todas as threads em um Warp executam a mesma instrução, mas podem operar em diferentes dados. Cada thread tem seu conjunto de registradores privados.

Thread: representa um fluxo independente de instruções, que é executado paralelamente em um núcleo de processamento da GPU;

Bloco : As threads são organizadas em blocos (ou grupos de threads). Existe um limite máximo para o número de threads em um bloco. Para GPU da Nvidia, pode-se ter no máximo 1024 threads por bloco. Os blocos são a unidade básica para associar as threads aos multiprocessadores.

Grid : Os blocos são organizados em um grid. Quando é realizada uma chamada de função ou *kernel* em uma GPU, o mesmo código é disparado para executar em múltiplos blocos com múltiplas threads. A chamada de *kernel* $<<< n, m >>>$ irá disparar n blocos com m threads cada. O valor de n pode ser bem alto. Nas GPU da Nvidia é possível disparar até 2 bilhões de blocos. Como cada bloco pode ter até 1024 threads, é possível disparar até 2 trilhões de threads.

O SMX é encarregado de gerenciar a execução das threads em uma unidade multiprocessada na GPU. Sua organização facilita o gerenciamento da execução e o compartilhamento de recursos. As GPUs possuem vários SMXs que trabalham em conjunto para processar tarefas em escala massivamente paralela. Por exemplo, a GPU T4, disponível gratuitamente no Google Colab, possui 40 SMXs, cada um contendo 64 núcleos CUDA, totalizando 2560 núcleos de processamento.

Ao contrário de um processador, o multiprocessador GPU opera com a execução e escalonamento de Warp, relacionado ao modelo SIMT. Se algumas threads em um Warp seguem caminhos diferentes (por exemplo, devido a um desvio condicional), ocorre a ”divergência de Warp”. Isso faz com que as threads em caminhos diferentes sejam executadas sequencialmente em vez de simultaneamente, o que pode reduzir o desempenho.

A latência das instruções em GPUs é pouco documentada. Alguns estudos recentes [2], [14] realizaram experimentos para coletar dados em diversas arquiteturas de GPU da Nvidia. Em média, uma operação de soma ou multiplicação de inteiros ou ponto flutuante de precisão simples tem uma latência de 4 a 10 ciclos. No entanto, diferentemente dos processadores, se houver dependência de dados entre duas instruções de um Warp (já que todas as suas threads executam juntas), o Warp será interrompido no pipeline. Em um processador, ele ficaria parado no estágio de decodificação. Uma GPU é mais parecida com uma arquitetura VLIW [11], onde o compilador é responsável pelo escalonamento e a instrução vai para execução somente quando suas dependências foram resolvidas.

O segredo da GPU é ter vários Warps no multiprocessador. Se algum deles tem uma dependência de dados, este warp irá sair do pipeline até que o resultado esteja disponível.

Enquanto isto, um ou mais Warps entram em execução, se estiverem prontos para executar. Se o código possuir muitas dependências e o número de Warp em execução for baixo, a GPU ficará inativa com perda de desempenho. Devido à grande variação nas latências das instruções e da memória, é difícil prever qual será o tempo de execução para um determinado código [2].

Outra característica pouco conhecida das GPUs é o formato das instruções. As GPU Turing e Volta utilizam 128 bits para codificar tanto uma instrução quanto suas informações de escalonamento associadas. As arquiteturas anteriores das GPUs da NVIDIA utilizavam uma palavra de 64 bits para instrução e outra separada para o escalonamento [14].

Codificar o escalonamento nas instruções apareceu pela primeira vez na arquitetura Kepler, que substituiu substancialmente o escalonamento dinâmico de hardware por escalonamento estático de software. As decisões de escalonamento de instruções são tomadas pelo compilador [24]. O escalonamento por software substituiu um escalonador de hardware complexo das gerações anteriores a GPU Kepler por um mais simples e eficiente, reduzindo a área de silício e consumindo menos energia. As GPUs mais novas melhoraram o escalonador com campos específicos para controle do hardware. Por exemplo, existe um campo de ciclos de espera. Este campo tem 4 bits e indica por quanto tempo o escalonador deve aguardar antes de emitir a próxima instrução, variando de 0 a 15 ciclos.

A. GPU didática

Devido à alta complexidade e à escassez de informações disponíveis sobre a microarquitetura da GPU, somente dispomos de informações superficiais ou indiretas obtidas a partir de experimentos com microbenchmarks [14]. Portanto, apresentaremos um modelo de GPU bastante simplificado com fins didáticos, visando transmitir ao estudante o papel do compilador e possibilitar a compreensão dos problemas e desafios enfrentados pelo compilador de GPU.

A GPU proposta tem as seguintes especificações:

- Warps com 4 threads
- Banco de Registradores com 64 registradores
- Máximo de 32 threads por Multiprocessador
- Máximo de 4 blocos por Multiprocessador
- Máximo de 16 threads por bloco
- Instruções: Load e Store com 4 ciclos quando o dado estiver na cache e 20 ciclos memória, Add e Mul com 3 ciclos
- Ordem de swap dos Warps é aleatória

B. Exemplo de execução

A proposta dos exemplos e problemas é fazer tabelas de rastreio de execução considerando um escalonamento feito pelo ”compilador”e a alternância de Warp para esconder a latência de espera devido a dependência de dados.

Primeiro apresentamos um exemplo bem simples para definir uma tabela de execução e mostrar a ideia básica da troca de Warp para garantir a execução com alto desempenho. Suponha apenas 2 Warps para executar uma simples atribuição $a = b+c$. Duas instruções de $load$ irão ler os dados de b e c para os registradores. Depois a soma é realizada e finalmente o resultado é armazenado em a usando uma instrução de $store$. O código em assembly será:

```
ld r1; b
ld r2; c
add r1,r1,r2; b+c
sd r1; a = b+c
```

Como a linguagem $markdown$ não oferece muitos recursos para edição de tabelas, optamos por usar o formato csv de planilhas no formato texto com ”;”para ser o delimitador de colunas, uma vez que a ”,”faz parte da especificação das instruções. Usando uma célula de código onde é possível gravar arquivos no disco virtual do Google Colab como ilustrado na Figura 13.

```
PC; Inst; Warp; Fetch; Decode; Exec; W
0; Ld r1; w1; 1; 2; 3-6; 7;
1; Ld r2; w1; 2; 3; 4-7; 8;
0; Ld r1; w2; 3; 4; 5-8; 9;
1; Ld r2; w2; 4; 5; 6-9; 10;
x; nop; ; 5
x; nop; ; 6
2; add r1,r1,r2; w1; 7; 8; 9-11; 12
x; nop; ; 8
2; add r1,r1,r2; w1; 9; 10; 11-13; 14
x; nop; ; 10
3; sd r1; 11; 12; 13-17
x; nop; ; 12
3; sd r1; 13; 14; 15-19
```

Figura 13. Exemplo de trecho em CSV com o rastreio de execução do código $a = b + c$ com 2 Warps na GPU didática.

Este exemplo ilustrativo mostra que o mesmo código é executado por vários Warps. Cada Warp tem seu próprio apontador de instruções ou ”contador de programa”(PC). A primeira coluna da tabela de rastreio mostra o PC. O PC começa com o valor 0 que corresponde a primeira linha do código. A segunda coluna mostra a instrução que está sendo executada e a terceira coluna mostra o ID do Warp. A notação é w_i, onde i é o número do Warp. Para melhorar a visualização, o CSV é lido e usando a biblioteca $Pandas$, a tabela de rastreio é exibida com formatação como ilustrado na Figura 14.

A GPU didática tem 4 estágios de pipeline. O primeiro estágio é de busca ou $fetch$, seguido do estágio de $decode$ para decodificação. A instrução não irá parar no estágio de $decode$, portanto se a instrução entra no pipeline, ela será executada. A responsabilidade é do compilador que faz o escalonamento. No estágio de execução, o tempo depende do tipo de instrução. No exemplo adotamos 3 ciclos para instruções de aritmética e 4 ciclos para instrução de leitura e escrita em memória, supondo que o dado está na cache.

Podemos observar que o Warp w_1 executa as duas primeiras instruções para ler os valores de r_1 e r_2. Como a terceira instrução é um add que precisa dos valores dos registradores, o add do Warp w_1 só será escalonamento na linha 6. Será

	PC	Inst	Warp	Fetch	Decode	Exec	W
0	0	Ld r1	w1	1	2	3-6	7.0
1	1	Ld r2	w1	2	3	4-7	8.0
2	0	Ld r1	w2	3	4	5-8	9.0
3	1	Ld r2	w2	4	5	6-9	10.0
4	x	nop		5	-	-	-
5	x	nop		6	-	-	-
6	2	add r1,r1,r2	w1	7	8	9-11	12.0
7	x	nop		8	-	-	-
8	2	add r1,r1,r2	w1	9	10	11-13	14.0
9	x	nop		10	-	-	-
10	3	sd r1	11	12	13-17	-	-
11	x	nop		12	-	-	-
12	3	sd r1	13	14	15-19	-	-

Figura 14. Tabela de rastreio gerada com Pandas a partir do CSV

buscada no ciclo 7 e no ciclo 8 realiza a decodificação e leitura dos registradores r_1 e r_2. Portanto, após executar as duas primeiras instruções, o Warp w_1 é retirado pelo escalonamento e um outro Warp entra em execução. Como neste primeiro exemplo temos somente dois Warp, o w_2 irá entrar em execução no ciclo 3 e também só executa as duas instruções de *load*.

Com apenas 2 Warps, este exemplo também ilustra que a GPU ficará ociosa nos ciclos 5 e 6. Nestes ciclos, a GPU não busca nenhuma instrução, usamos a instrução *nop*, pois os warps w_1 e w_2 estão aguardando os dados da cache e não podem continuar. Depois observamos novamente que a instrução *add* de w_1 executa e volta a se retirar para aguardar o escalonamento da instrução *store*. Neste momento ocorre a inserção de mais um *nop*.

Feita a introdução, os alunos receberam alguns pequenos desafios de elaborar códigos modificando o número de Warps, modificando a posição do dado que passa da cache para memória global aumentando a latência de 4 ciclos para 20 ciclos, além de propor códigos com reuso dos dados. O objetivo é mostrar a importância manter os dados nos registradores, evitando a latência da memória. Além disso, alternar Warps e instruções com dependência irão manter a GPU ocupada e com alta vazão na execução de instruções.

C. Visualização e Exemplos gerados

Além de executar algumas variações de exemplos simples com poucas instruções, foi requisitado aos estudantes a elaboração de uma visualização do rastreio de execução com cores diferentes para cada Warps. Cada grupo explorou recursos diferentes das bibliotecas Python com *Pandas*, *Matplotlib* e outras. A seguir iremos mostrar alguns resultados gerados pelos estudantes.

Primeiro, uma visualização textual que mostra a evolução no tempo por colunas. Para o exemplo anterior teremos a seguinte entrada em *CSV*:

```
Warp; 1;2;3;4;5;6;7;8;9;0;1;2;3
w1; ld; ld; ; ; ; ; add; ; ; ; sd
w2;  ;  ; ld; ld; ; ; ; ; add; ; ; ; sd
```

que irá resultar na visualização abaixo usando a função *display* da biblioteca *pandas*:

```
  Warp    1    2    3    4  5  6    7  8    9  0
0   w1   ld   ld                    add
1   w2             ld   ld                   add
```

onde observamos o momento que cada instrução inicia a sua execução e quais ciclos que a GPU fica ociosa por não ter Warps ativos em função das dependências de dados.

Warp	1	2	3	4	5	6	7	8	9
w1	Ld	Ld							add
w2			Ld	Ld					
w3					Ld	Ld			
w4							Ld	Ld	

Figura 15. Tabela de rastreio gerada com tabulate a partir do CSV

Outra opção para melhorar a formatação para exibir a tabela de rastreio é o uso da biblioteca *tabulate* que pode formatar e exibir dados tabulares de forma visualmente agradável. Ela simplifica o processo de criar tabelas e apresentar dados em vários formatos, como texto simples, HTML, LaTeX e outros. O principal objetivo da biblioteca *tabulate* é converter uma lista de listas ou outras estruturas de dados em um formato de tabela com opções personalizáveis para cabeçalhos, alinhamento e outras configurações de formatação. A Figura 15 mostra a solução apresentada por um grupo de estudantes com o uso de uma tabela onde cada linha tem o rastreio da execução de um Warp. As colunas mostram a evolução no tempo. Neste exemplo temos 4 Warps executando o código inicial da atribuição $a = b + c$.

	0	1	2	3	4	5	6	7	8	9	10
0	warp	1	2	3	4	5	6	7	8	9	10
1	1	Ld r1	Ld r2					nop	add r1,r1,r2		
2	2			Ld r1	Ld r2			nop		add r1,r1,r2	
3	3					Ld r1	Ld r2	nop			add r1,r1,r2

Figura 16. Tabela de rastreio gerada com html a partir do CSV

Outra opção apresentada fez uso da biblioteca *pandas*, das funcionalidades da função *display* do *ipython* junto com facilidades para geração de *html*, uma vez que o Google Colab pode processá-la. A Figura 16 mostra a solução apresentada com *html* onde cada linha tem o rastreio da execução de um Warp. As colunas mostram a evolução no tempo. Neste exemplo temos 3 Warps executando o código inicial da atribuição $a = b + c$.

Outra opção é explorar as funcionalidades do *display* do *ipython*, usando cores. A Figura 17 mostra a solução apresentada com *display* onde cada linha tem o rastreio da execução de um Warp marcando que qual unidade a instrução se encontra e usando cores diferentes para Warps diferentes. As colunas mostram a evolução no tempo. Neste exemplo

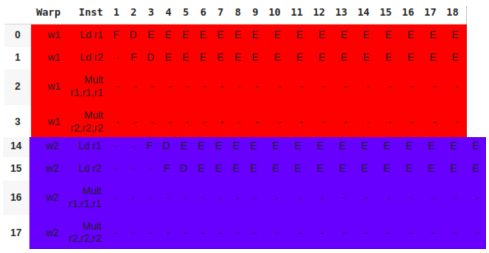

Figura 17. Tabela de rastreio gerada com display a partir do CSV

temos 2 Warps executando o código inicial da atribuição $a = b + c$ mas que tem uma longa latência pois os dados estão na memória global.

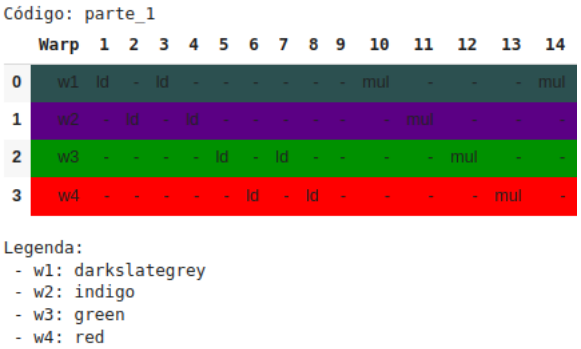

Figura 18. Tabela de rastreio gerada com display e pandas a partir do CSV

Outra opção com *pandas* e *display* é ilustrada na Figura 18 com 4 Warps e apenas o momento que a instrução é disparada.

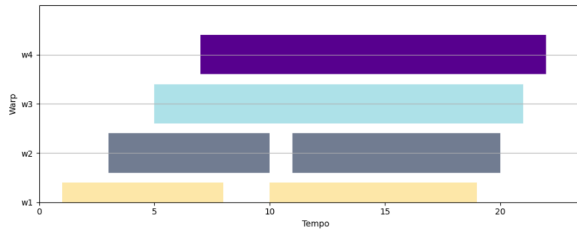

Figura 19. Tabela de rastreio gerada com matplotlib a partir do CSV

Uma opção com *matplotlib* é ilustrada na Figura 19 que apresenta a execução do mesmo código da atribuição $a = b+c$ com 4 Warps concorrentes. Os estudantes optaram por manter ativos os Warps com instruções em execução no pipeline. Podemos observar que os Warps 1 e 2 ficam fora do pipeline por alguns instantes, mas que o pipeline está sempre cheio com a alternância dos Warps.

V. Conclusão

Python com Google Colab é uma combinação atrativa para que os professores possam criar simuladores e material didático que pode envolver e engajar os estudantes em diversas áreas de aprendizado. Primeiro, o Google Colab é uma plataforma baseada na nuvem, que permite aos alunos e educadores acessar e trabalhar com o Python em qualquer dispositivo com acesso à internet, sem a necessidade de instalar nada localmente. Isso torna o ambiente de aprendizado mais acessível e reduz as barreiras para os estudantes começarem a explorar e experimentar com Python. Ao aliar estas facilidades com os desafios de arquitetura de computadores, podemos motivar os estudantes para estudar as técnicas de arquitetura de computadores como *software pipelining* e escalonamento de GPUs, fazendo ao mesmo tempo, ferramentas para validação do aprendizado.

Além disso, o Google Colab suporta a execução de código em células, o que possibilita um ambiente interativo onde os estudantes podem executar código, visualizar resultados e receber feedback instantâneo sobre seus esforços. Essa interatividade ajuda os estudantes a aprenderem ao verem o impacto das alterações em tempo real. Por exemplo, o ensino de *software pipelining* tem detalhes como o balanceamento da propagação de valores de registradores entre iterações não consecutivas. Com a iteratividade, o estudante pode incluir e retirar o registrador para visualizar a inserção e remoção do erro na execução do código.

Python possui uma grande variedade de bibliotecas para criação de gráficos e visualizações. O Colab permite a integração perfeita com bibliotecas como Matplotlib, Seaborn e Plotly, que podem criar gráficos interativos e visualmente atraentes, além de tabelas. A atividade de escalonamento de Warps em GPUs, mostrou que cada grupo de estudantes explorou uma forma diferente de visualização. Isso torna a exploração de dados e a apresentação de informações muito mais envolventes para os estudantes. Com o Colab, é fácil instalar e importar essas bibliotecas, o que oferece a possibilidade de explorar áreas como ciência de dados, aprendizado de máquina, análise de texto, entre outras, ampliando as opções de criação de material didático e simuladores.

Além disso, o uso de recursos interativos usando Markdown e HTML permite que o material educacional seja mais completo combinando elementos multimídia, como imagens, vídeos e gráficos. Finalmente, o Colab facilita a colaboração entre estudantes e educadores, pois permite que múltiplas pessoas editem o mesmo documento ao mesmo tempo. Além disso, os notebooks do Colab podem ser facilmente compartilhados com outras pessoas por meio de um link, tornando-o um ótimo recurso para distribuir materiais educacionais.

O uso de python para ”simular”um assembly RISC foi bem atrativo para os estudantes e pode ser mais explorado. A geração de traços de execução com CSV permite também gerar vários exemplos que podem ser continuados e modificados para criar novos enunciados de problemas a serem resolvidos. Além disso, o uso de *pandas* ou *tabulate* gera uma visualização agradável dentro do ambiente do Colab.

VI. Agradecimentos

FAPEMIG (PIBIC, $APQ - 01577 - 22$), CNPq, NVIDIA, Funarbe. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior −− Brasil (CAPES)− Código de Financiamento 001.

REFERÊNCIAS

[1] FA Alves, Danilo Almeida, Lucas Bragança, André BM Gomes, Ricardo S Ferreira, and José Augusto M Nacif. Ensinando arquiteturas vetoriais utilizando um simulador de instruções mips. *International Journal of Computer Architecture Education (IJCAE)*, 4(1):9–12, 2015.

[2] Yehia Arafa, Abdel-Hameed A Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Low overhead instruction latency characterization for nvidia gpgpus. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2019.

[3] Hector Perez Baranda, Jeronimo Costa Penha, and Ricardo Ferreira. Implementação de um preditor de desvio no mips 5 estágios. *International Journal of Computer Architecture Education*, 6, 2018.

[4] Ekaba Bisong and Ekaba Bisong. Google colaboratory. *Building machine learning and deep learning models on google cloud platform: a comprehensive guide for beginners*, pages 59–64, 2019.

[5] Michael Canesche, Lucas Bragança, Omar Paranaiba Vilela Neto, Jose A Nacif, and Ricardo Ferreira. Google colab cad4u: Hands-on cloud laboratories for digital design. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[6] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9*, pages 483–484. Springer, 2002.

[7] Ricardo Ferreira, Michael Canesche, and Jerônimo Penha. Google colab para ensino de computação. In *Anais Estendidos do III Simpósio Brasileiro de Educação em Computação*, pages 46–47. SBC, 2023.

[8] Ricardo Ferreira, Salles Viana Gomes de Magalhães, and José AM Nacif. Métricas e números: Desmistificando a programação de alto desempenho em gpu. *Minicursos do WSCAD, Sociedade Brasileira de Computação, DOI: https://doi.org/10.5753/sbc.46486.1*, 2019.

[9] Ricardo Ferreira and Geraldo Fontes. Ensino de organizaçoes de memória em arquiteturas paralelas usando placas gráficas aceleradoras. *International Journal of Computer Architecture Education (IJCAE)*, 2, 2013.

[10] Ricardo Ferreira, Jose Nacif, Salles Magalhaes, Thales de Almeida, and Racyus Pacifico. Be a simulator developer and go beyond in computing engineering. In *2015 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2015.

[11] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[12] Roberto Giorgi and Gianfranco Mariotti. Webrisc-v: A web-based education-oriented risc-v pipeline simulation environment. In *Proceedings of the workshop on computer architecture education*, pages 1–6, 2019.

[13] John Gruber. Markdown: Syntax. *URL http://daringfireball. net/projects/markdown/syntax. Retrieved on June*, 24:640, 2012.

[14] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[15] Dimitris Kehagias and V Douskas-Bertlviser. Android-based simulator to support tomasulo algorithm teaching and learning. *International Journal of Computer Applications*, 975:8887.

[16] Matthias Koenig and Roin Rasch. Digital teaching an embedded systems course by using simulators. In *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*, pages 1–7. IEEE, 2021.

[17] Fernando Passe, Michael Canesche, Omar Paranaiba Vilela Neto, Jose A Nacif, and Ricardo Ferreira. Mind the gap: Bridging verilog and computer architecture. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[18] David A Patterson and John L Hennessy. Computer organization and design: the hardware/software interface (the morgan kaufmann series in computer architecture and design). *Paperback, Morgan Kaufmann Publishers*, 2013.

[19] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. A risc-v simulator and benchmark suite for designing and evaluating vector architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–30, 2020.

[20] William Stallings. Arquitetura e organização de computadores 8a edição, 2010.

[21] Nathan D. Typanski. Tomasulo: floating-point MIPS-like instruction pipeline. http://nathantypanski.github.io/tomasulo-simulator/.

[22] Keyhan Vakil. Venus, risc-v simulator. https://venus.kvakil.me/.

[23] Kenneth Vollmar and Pete Sanderson. Mars: an education-oriented mips assembly language simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 239–243, 2006.

[24] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–43, 2017.