Um Ambiente integrado à Qiskit para Execução Remota de Algoritmos Quânticos em GPU

Maria Eduarda Mascarenhas da Silva[1], **Helena Carvalho Leal**[2], **Calebe Micael de Oliveira Conceição**[1], **Rodolfo Botto de Barros Garcia**[1]

[1]Departamento de Computação – Universidade Federal de Sergipe (UFS) – 49.100-000 – São Cristóvão-SE, Brasil

[2]Departamento de Física – Universidade Federal de Sergipe (UFS) – 49.100-000 – São Cristóvão-SE, Brasil

{dudxyz, helena_leal19}@academico.ufs.br,

{calebe,rodolfo.botto}@dcomp.ufs.br

Abstract. *This paper presents an environment integrated with Qiskit for the remote execution of quantum algorithms using GPUs, aiming to overcome the performance limitations of local hardware. The solution enables dynamic resource allocation and the efficient parallelization of quantum operations, thereby democratizing large-scale simulations. For validation, a system was developed combining a Qiskit-integrated API with GPUs for the computation of quantum matrices and vectors. Experimental results demonstrate significant performance gains over exclusively CPU-based approaches, which helps to advance research in quantum computing. Therefore, the environment facilitates the development, testing, and validation of algorithms for the scientific community.*

Resumo. *Este artigo apresenta um ambiente integrado ao Qiskit para execução remota de algoritmos quânticos utilizando GPUs, visando superar limitações de desempenho em hardware local. A solução permite alocação dinâmica de recursos e paralelização eficiente de operações quânticas, democratizando simulações de grande escala. Para validação, foi desenvolvido um sistema combinando API integrada ao Qiskit com GPUs para cálculo de matrizes e vetores quânticos. Resultados experimentais demonstram ganhos significativos de performance frente a abordagens baseadas exclusivamente em CPUs, o que impulsiona pesquisas em computação quântica. Portanto, o ambiente facilita o desenvolvimento, teste e validação de algoritmos para a comunidade científica.*

1. Introdução

A computação quântica tem sido apontada como a próxima revolução tecnológica, prometendo avanços significativos em áreas como otimização, inteligência artificial, simulações químicas e segurança cibernética [Preskill 2018]. Diferente dos computadores comuns, os computadores quânticos exploram princípios fundamentais da mecânica quântica, como a superposição e o emaranhamento, permitindo representar e processar informações e dados de maneira mais eficiente. Grandes empresas como IBM, Google e Microsoft, assim como governos ao redor do mundo, têm investido em massa nessa tecnologia emergente. Segundo a McKinsey (2024), a computação quântica deverá gerar cerca de US$1,3 trilhão em valor econômico até 2035 [Company 2024].

Apesar de seu potencial transformador, essa tecnologia ainda enfrenta desafios significativos. Entre eles, destacam-se as limitações físicas dos dispositivos, a necessidade de correção de erros e a dificuldade de acesso a hardware quântico em larga escala. Por esse motivo, o uso generalizado de computadores quânticos ainda não é uma realidade, e pesquisadores e desenvolvedores recorrem amplamente a simulações quânticas realizadas em computadores clássicos. Diversos grupos de pesquisa têm explorado soluções alternativas para otimizar simulações quânticas, como o uso de clusters de alto desempenho, arquiteturas híbridas CPU-GPU e plataformas de computação em nuvem especializadas [Sarode 2024]. Cada abordagem apresenta vantagens e limitações, relacionadas tanto ao desempenho quanto à escalabilidade e ao custo operacional.

Nesse contexto, ferramentas como Qiskit - um framework de código aberto desenvolvido pela IBM - têm desempenhado papel central ao permitir a simulação e desenvolvimento de algoritmos quânticos em ambientes clássicos. No entanto, a execução dessas simulações demanda grande poder computacional, especialmente para circuitos quânticos de escala maior, o que limita a experimentação e o avanço da pesquisa. Para contornar restrições computacionais em diversas áreas científicas, tem-se utilizado unidades de processamento gráfico (GPUs), conhecidas pela sua alta capacidade de paralelização [Schmidt and Hildebrandt 2024].

Diante desse cenário, este trabalho propõe o desenvolvimento e validação de um ambiente híbrido que integra o framework Qiskit com recursos de computação paralela baseados em GPU, para execução remota da simulação de algoritmos quânticos. O desenvolvimento inclui o desenvolvimento de uma API RESTful para gerenciar a criação, o armazenamento e a inspeção de objetos Qobj, permitindo que usuários submetam circuitos quânticos em formato JSON e recebam resultados diretamente em seus ambientes Qiskit. A proposta busca superar os gargalos computacionais que limitam a simulação de circuitos quânticos em hardware local, especialmente quando se lida com um número elevado de qubits e operações unitárias complexas.

Futuramente, o ambiente aqui proposto irá compor uma infraestrutura de computação de alto desempenho capaz de escalar dinamicamente problemas em simulações quânticas, conforme a demanda. Ao combinar elementos da computação quântica com técnicas consolidadas de computação de alto desempenho, este trabalho busca democratizar o acesso remoto desses recursos.

O restante deste artigo está organizado da seguinte forma: na **Seção 2** descrevem-se os trabalhos relacionados presentes na literatura; na **Seção 3**, realiza-se uma revisão dos fundamentos da Qiskit e do processamento em GPU baseado em CUDA; a **Seção 4** detalha a metodologia, cuja abordagem é híbrida, combinando computação clássica acelerada por GPU (via CUDA) e simulação de circuitos quânticos (via Qiskit), integrados por uma camada de comunicação utilizando o formato JSON; na **Seção 5**, apresentamos resultados promissores do cálculo $A \cdot \vec{x}$ (onde A é a matriz e $\vec{x}$ o vetor); por fim, na **Seção 6**, concluímos o projeto com a maior parte dos resultados satisfatórios desta pesquisa que está em estágio inicial.

2. Trabalhos Relacionados

A otimização de simulações quânticas em hardware clássico é um campo de pesquisa ativo. O estudo de [Sarode 2024] apresenta uma análise comparativa de simulação em

GPU, contrastando a partição de circuitos (usando CutQC) com a execução completa (usando Qiskit-Aer-GPU). Conclui-se que a partição reduz o consumo de memória, sendo vantajosa em cenários com recursos limitados, enquanto a simulação completa oferece maior desempenho em sistemas com GPUs abundantes, evitando o custo exponencial do pós-processamento de múltiplos subcircuitos.

No contexto de hardware, o trabalho de [Schieffer et al. 2024] investiga o impacto da memória unificada do superchip Grace Hopper em aplicações de HPC, incluindo simulações no Qiskit. O estudo demonstra que o desempenho da memória integrada CPU-GPU é fortemente influenciado pelo padrão de inicialização e acesso aos dados, oferecendo diretrizes para otimização em arquiteturas híbridas relevantes para a nossa proposta.

A arquitetura aqui proposta, que explora a representação padronizada Qobj em JSON para fragmentar e distribuir cargas de trabalho quânticas, espelha conceitualmente o teorema de repetição paralela de Raz [Raz 1998], o qual demonstra que a repetição simultânea de um protocolo reduz a probabilidade de erro de modo exponencial. Neste trabalho, cada Qobj é particionado em subexperimentos independentes — análogo às "rodadas" paralelas do teorema — e encaminhado a um módulo de aceleração CUDA/GPU capaz de processar vetores e matrizes unitárias em larga escala. Essa combinação não só possibilita ganhos de performance ao explorar o paralelismo massivo da GPU, mas também oferece um banco de experimentos reproduzíveis que pode ser utilizado para medir, de forma empírica, a queda na taxa de falhas das simulações quânticas conforme aumentamos o número de repetições paralelas.

3. Fundamentação Teórica

A fundamentação teórica deste trabalho apresenta os conceitos básicos do Qiskit e do processamento em GPU baseado em CUDA.

3.1. Qiskit

Qiskit (*Quantum Information Science Kit*) é um SDK (*Software Development Kit*) de código aberto para trabalhar com computadores quânticos em nível de circuitos, pulsos e algoritmos. Desenvolvido pela IBM, o Qiskit foi projetado para permitir a construção, simulação e execução de programas quânticos de forma mais acessível. Ele atua como uma interface fundamental entre os algoritmos quânticos abstratos e a execução prática, seja em simuladores quânticos clássicos de alto desempenho ou diretamente em hardware quântico real, incluindo os processadores quânticos da IBM Quantum.

No campo da computação quântica, o Qiskit trabalha com conceitos de superposição, a qual permite que um sistema com n qubits processe simultaneamente todos os 2^n estados em uma única operação, o que explora o paralelismo quântico inerente. Isso possibilita a execução da simulação de algoritmos que, quando executados em computadores quânticos, podem superar exponencialmente a eficiência dos algoritmos clássicos em certas tarefas, como fatoração de números (algoritmo de Shor) e busca em bases de dados não estruturadas(Algoritmo de Grover).

A arquitetura do Qiskit é composta por várias camadas que cobrem diferentes aspectos do desenvolvimento quântico. O Qiskit Terra é a base que fornece as ferramentas fundamentais para a criação e manipulação de circuitos quânticos, gerenciamento de

portas quânticas, medições e registros clássicos. Complementando isso, simulador Aer do Qiskit, desenvolvido pela IBM, foi utilizado para as execuções locais dos circuitos e oferece simuladores otimizados para testar e depurar programas quânticos em máquinas clássicas antes de enviá-los para o hardware [IBM 2023]. Além dessas funcionalidades essenciais, o ecossistema Qiskit inclui bibliotecas para algoritmos quânticos avançados, como otimização, química quântica e aprendizado de máquina quântico. Ademais, o SDK disponibiliza ferramentas para caracterização de ruído e mitigação de erros.

3.1.1. Qobj

O Quantum Object (Qobj) é uma estrutura de dados que representa um pacote completo de informações necessárias para a execução de um experimento quântico em um simulador ou em um backend real. Trata-se de um objeto serializável em formato JSON que descreve, entre outros elementos, o circuito quântico a ser executado, o backend de destino e as instruções de execução. Sua estrutura é composta por três elementos principais: um Qobj header, que contém informações gerais como nome do job, backend e versão; uma lista de experimentos, que inclui os circuitos ou *schedules* a serem executados; e um conjunto de parâmetros globais de configuração, como o número de *shots* e a *seed* para controle de aleatoriedade em simulações.

3.2. CUDA

CUDA (*Compute Unified Device Architecture*) é um modelo de programação paralela desenvolvido pela NVIDIA que permite a utilização de unidades de processamento gráfico (GPUs) para computação de propósito geral (GPGPU).

A hierarquia de memória é um fator crítico para o desempenho em CUDA. Para a aceleração, foi utilizada a arquitetura CUDA da NVIDIA, que permite computação de propósito geral em GPUs. O desempenho em CUDA é sensível à hierarquia de memória, sendo crucial o uso de registradores rápidos e memória compartilhada para a comunicação entre *threads*, enquanto a memória global, de maior latência, é usada para dados massivos.

No contexto deste trabalho, CUDA é especialmente eficiente para:

1. **Operações Tensoriais**: cálculo paralelo de amplitudes de estado, por exemplo,
$$|\psi'\rangle = U_k \otimes I_{n-k}|\psi\rangle,$$
onde portas quânticas são aplicadas como produtos tensoriais paralelizáveis.
2. **Paralelismo de Portas**: execução concorrente de portas independentes via múltiplas *streams* CUDA.
3. **Manipulação de Estados**: atualização paralela de vetores de estado com 2^n componentes, proporcionando *speedup* aproximado dado por:
$$\text{Speedup} = O\left(\frac{\#\text{ elementos}}{\#\text{ CUDA cores}}\right).$$

4. Metodologia

O projeto a que este trabalho está incluído tem como etapas: (i) projetar uma API RESTful que possibilite a comunicação entre o Qiskit e uma infraestrutura baseada em CUDA; (ii)

implementar algoritmos de simulação de circuitos quânticos otimizados para execução paralela em GPU ao realizar cálculos de matrizes e vetores relacionados a circuitos quânticos de forma mais eficiente.; (iii) avaliar experimentalmente os ganhos de desempenho obtidos com a paralelização em comparação com abordagens puramente baseadas em CPU; e (iv) propor uma arquitetura escalável e replicável que possa ser utilizada por pesquisadores e instituições com recursos computacionais limitados.

Assim, a arquitetura proposta neste trabalho, e que corresponde ao estágio inicial do projeto, cumpre as três primeiras etapas e estabelece um fluxo coeso a ser detalhado nas subseções desta metodologia: em **Seção 4.1** detalha a abstração via API; a **Seção 4.2** explica como a aceleração via CUDA realiza operações lineares complexas delegadas à GPU e, por fim, a **Seção 4.3** projeta a orquestração Python, como a ponte que integra o Qiskit, a serialização e CUDA.

4.1. Design da API integrada ao Qiskit

A API foi implementada como um serviço RESTful em Python, utilizando o framework Flask para modularizar a comunicação entre o cliente e os componentes internos do sistema. Sua função principal é automatizar a criação, armazenamento e consulta de objetos Qiskit do tipo "Qobj" em um servidor local. Embora todos os módulos estejam hospedados na mesma máquina, a utilização do Flask permite uma separação lógica entre as etapas do pipeline, facilitando a organização do código e futuras extensões para arquiteturas distribuídas. O ambiente Python foi configurado com Qiskit Aer (simulador de alto desempenho), Flask e bibliotecas padrão de manipulação e serialização de arquivos, como pickle e os.

O fluxo apresentado na Figura 1 descreve a interação entre os principais componentes da arquitetura: o servidor Flask, o módulo intermediário ponte.py e o núcleo de execução em GPU (cuda_exec). Inicialmente, ponte.py envia uma requisição GET ao servidor Flask (/readqobj) para obter um objeto quântico (Qobj) em formato JSON, previamente armazenado. Após receber esse JSON, ponte.py o converte em um objeto QuantumCircuit e extrai a matriz do operador e o vetor de estado associados, que são serializados em arquivos binários (matrix.bin e vector.bin). Em seguida, esses arquivos são passados como entrada para o executável CUDA (cuda_exec), responsável por realizar o cálculo em GPU. O resultado, salvo em result.bin, é então lido e convertido novamente para JSON por ponte.py. Por fim, esse resultado é enviado de volta ao servidor Flask por meio de uma requisição POST (/result). O servidor responde com um status HTTP 200, confirmando o recebimento e o encerramento do fluxo.

Os efeitos práticos da implementação desse fluxo de infraestrutura cliente-servidor, bem como seu desempenho e limitações observadas durante a execução de simulações, serão devidamente analisados e discutidos na Seção de Resultados. Essa avaliação visa fornecer uma compreensão crítica sobre a eficiência da arquitetura proposta e seu potencial de aplicação em cenários reais de simulação quântica.

4.2. Processamento Paralelo em GPU com CUDA

O núcleo da aceleração computacional é um *kernel* desenvolvido em CUDA, encapsulado no arquivo `cuda_exec.cu`, cuja responsabilidade é executar a operação de produto matriz-vetor com números complexos. Essa operação é central para a simulação de

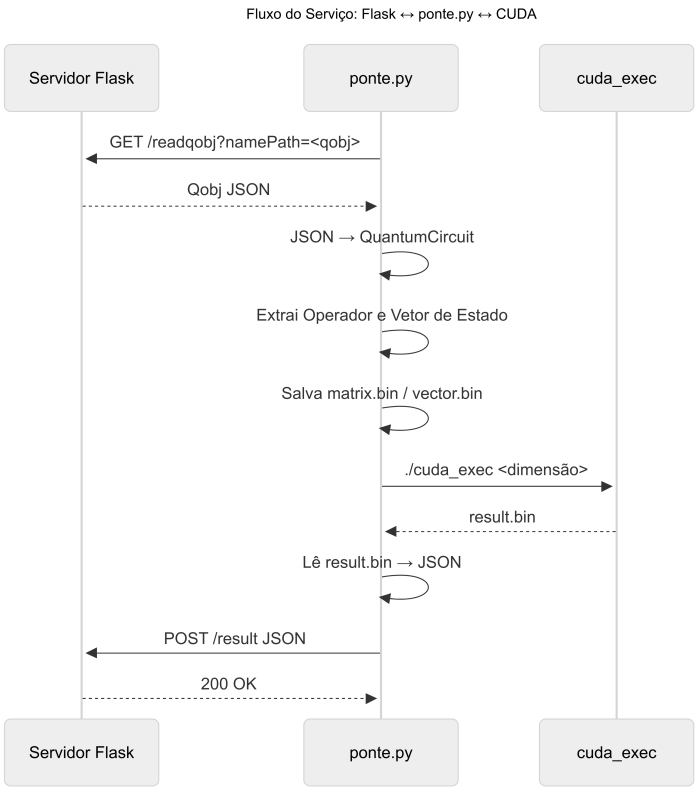

Figura 1. Diagrama de sequência do fluxo da arquitetura integrada ao Qiskit.

estados quânticos, pois aplica a matriz unitária, que representa o circuito quântico, sobre o vetor de estado do sistema. O processo inicia-se com a alocação de memória no hospedeiro (*host*) para a matriz (`h_matrix`), o vetor de entrada (`h_vector`) e o vetor de resultado (`h_result`). Em seguida, os dados são carregados a partir dos arquivos binários `matrix.bin` e `vector.bin`, que armazenam as informações no formato nativo `cuFloatComplex`.

Com os dados preparados no *host*, a etapa seguinte consiste na transferência para a memória do dispositivo de processamento (GPU). A API do CUDA é utilizada para alocar memória na GPU com `cudaMalloc` e para copiar os dados com `cudaMemcpyHostToDevice`. Uma vez que a matriz e o vetor residem na memória da GPU, o *kernel* é invocado. A grade de *threads* (`gridSize`) é dimensionada para cobrir todas as linhas da matriz, permitindo que cada *thread* calcule, em paralelo, um único elemento do vetor de resultado. Conforme o trecho de código a seguir, cada *thread* computa o produto escalar entre a sua linha correspondente na matriz e o vetor de estado.

```
cuFloatComplex sum = make_cuFloatComplex(0,0);
for (int j = 0; j < cols; j++)
    sum = cuCaddf(sum, cuCmulf(matrix[row*cols + j], vector[j]));
result[row] = sum;
```

Finalizada a execução do *kernel*, os resultados são transferidos de volta para o *host* com a função `cudaMemcpyDeviceToHost`, salvos no arquivo `result.bin` e, por

fim, toda a memória alocada em ambos os dispositivos é liberada para evitar vazamentos de recursos.

4.3. Ponte de Integração entre Qiskit e o Núcleo de Aceleração CUDA

A interface entre o ecossistema Qiskit e o módulo de aceleração baseado em GPU (CUDA) é realizada pelo script Python `ponte.py`, que funciona como camada de abstração e orquestração. Esse componente recebe requisições REST no endpoint `/readqobj` do serviço Flask, obtém o Qobj em JSON e o converte em um objeto `QuantumCircuit`. Em seguida, extrai a matriz unitária do operador via `Operator(circ).data` e o vetor de estado com `Statevector.from_instruction(circ).data`. Ambos os arrays são serializados em formato `np.complex64` e gravados em dois arquivos binários — `matrix.bin` e `vector.bin` — como pré-requisito para a execução em GPU.

O "núcleo CUDA" propriamente dito é um executável compilado em C++/CUDA (`cuda_exec`) que realiza a multiplicação de matrizes e vetores de forma massivamente paralela na GPU. Ele é invocado por `ponte.py` via `subprocess`, recebendo a dimensão do sistema (2^n) como argumento de linha de comando e carregando `matrix.bin` e `vector.bin` diretamente na memória de dispositivo. Após a computação, produz um arquivo de saída `result.bin`, contendo o vetor resultante também em `np.complex64`.

`ponte.py` então lê `result.bin`, converte os dados de volta para estruturas Python (dicionário/JSON) e envia o resultado ao Flask por meio de uma requisição POST ao endpoint `/result`, encerrando o ciclo de processamento.

Ao empregar CUDA, o sistema explora a arquitetura SIMT (Single Instruction, Multiple Threads) das GPUs, dividindo cada operação de multiplicação em milhares de threads que trabalham em paralelo. Essa abordagem não só reduz drasticamente o tempo de execução para grandes dimensões de circuito (por exemplo, matrizes 1000×1000), mas também demonstra a relevância do núcleo CUDA como componente crítico para viabilizar simulações quânticas em larga escala. Future works incluirão otimizações de gerenciamento de memória GPU e balanceamento de carga para maximizar ainda mais o desempenho e reduzir a latência do pipeline.

5. Resultados e Discussão

O projeto apresenta resultados experimentais, embora ainda em fase inicial, com testes focados em circuitos quânticos simples enviados via qobj para o CUDA, que multiplicou matrizes unitárias de baixa dimensão e vetores de estado reduzidos. Com isso, avaliando o potencial de aceleração, foi implementado um *testbench*.

Seguindo o fluxo de arquitetura descrito na Figura 1, um cliente envia uma requisição POST para /generateqobj, o serviço lê um payload JSON contendo: (1) o número de qubits, (2) uma lista ordenada de definições de portas (cada uma especificando o tipo e os qubits-alvo) e (3) um prefixo de nome de arquivo opcional. Internamente, é instanciado um QuantumCircuit com o tamanho de registrador solicitado, itera-se sobre cada instrução de porta (por exemplo, Hadamard ou CNOT) para aplicá-la, e então o circuito é transpilado para o backend qasm_simulator do Aer. O Qobj resultante é serializado via pickle e gravado em disco como ≪namePath>.qobj.

Para permitir que clientes inspecionem os objetos salvos, implementou-se um endpoint GET em /readqobj. Dado um parâmetro de consulta namePath, o serviço carrega o arquivo pickle e determina dinamicamente se ele contém um Qobj (com .config, .header e .experiments) ou um QuantumCircuit bruto. No caso de um Qobj, metadados, como nome e versão do backend, número de shots, tamanhos de registradores, configurações de medição e sequências de instruções detalhadas por experimento, são extraídos acessando os atributos com verificações de existência. Se o arquivo contiver apenas um circuito, extraímos o número de qubits e os dados de instrução diretamente do atributo **.data** do circuito. Por fim, o endpoint retorna um objeto JSON estruturado contendo qobj_data, header e uma lista de experiments, cada um com sua própria configuração e lista de instruções.

Para permitir a integração direta com a interface padrão de execução de circuitos do Qiskit, desenvolveu-se um conjunto de componentes adicionais que seguem as abstrações exigidas pela arquitetura modular do framework. A seguir, detalhamos a implementação desses elementos:

- **Provider**: Implementou-se um provider customizado do Qiskit que encaminha a execução de circuitos para o serviço RESTful criado anteriormente. Para isso, estendeu-se as classes base BackendV1 e JobV1 do Qiskit, além de criar um Provider que registra o nosso backend.
- **Design do Backend**: A classe WebserviceBackend herda de BackendV1 e recebe, no construtor, a URL do serviço Flask (http://127.0.0.1:5000). No método run(), aceita-se um único QuantumCircuit por vez, é extraído seu número de qubits e as instruções — mapeando portas H e CX para um JSON compatível com o endpoint /generateqobj — e fazemos o POST para gerar e salvar o Qobj no servidor. Caso o servidor responda com erro, uma exceção é lançada para interromper o fluxo.
- **Gerenciamento de Jobs**: A classe WebserviceJob armazena o **job_id**, o backend e o circuito de entrada. Seu estado inicial é RUNNING. Ao chamar result(), fazemos um GET em /readqobj para recuperar o arquivo .qobj serializado, extraímos o JSON de metadados e construímos um objeto Result do Qiskit com um resultado simulado (contagens de exemplo e sucesso). Após obter o resultado, atualizamos o status para DONE e retornamos o Result para o cliente.
- **Registro de Backend**: Em WebserviceProvider, estendemos Provider para instanciar um QasmBackendConfiguration que descreve nosso backend (nome, versão, número de qubits, gates suportadas, etc.). O método backends() permite listar todos os backends registrados ou filtrar por nome, integrando-se ao fluxo padrão de seleção do Qiskit.

Os resultados obtidos demonstram que a arquitetura proposta consegue reproduzir, com fidelidade e flexibilidade, o ciclo completo de construção, serialização e recuperação de circuitos quânticos, integrando-se de forma transparente à interface do Qiskit. A criação de uma camada RESTful entre o cliente e o backend facilita o desacoplamento entre a lógica de construção dos circuitos e o ambiente de execução, permitindo futuras extensões para execução remota. Esses aspectos indicam caminhos promissores para trabalho futuro, como o suporte a múltiplas instruções nativas e a adaptação do serviço para backends reais de clusters personalizados.

Na Figura 2, pode-se observar o ambiente de teste para uma matriz de dimensão 4×4. Como esperado para um sistema de pequena escala, os resultados não indicaram *speedup* em relação à execução na CPU.

Essa falta de aceleração ocorre porque, em matrizes pequenas, o *overhead* associado à transferência de dados para a memória da GPU e à inicialização do processamento paralelo domina o tempo total de execução. O verdadeiro ganho de desempenho com CUDA só se manifesta em problemas de grande porte, onde a capacidade de paralelização massiva da GPU é plenamente explorada.

Para ilustrar esse ponto, a Tabela 2 reúne os resultados de um código genérico de multiplicação de matrizes em duas escalas distintas:

- **Matrizes** 3×3**:** O *speedup* é desprezível ($\approx 0{,}002\times$, conforme Tabela 2), pois a CPU é mais eficiente para uma carga de trabalho tão baixa, evitando o custo do *overhead*.
- **Matrizes** 1000×1000**:** O *speedup* torna-se claro e significativo ($\approx 130\times$, conforme Tabela 2), uma vez que o processamento paralelo de milhões de operações na GPU supera em muito o tempo de inicialização.

Tabela 1. Comparative Performance between GPU (CUDA) and CPU in Matrix Multiplication

Dimension	GPU Time (ms)	CPU Time (ms)	Speedup (×)
3×3	1.992	0.004288	0.00215
1000×1000	17.203	2235.564	129.95

Source: Developed by the authors

Tabela 2. Desempenho comparativo entre GPU (CUDA) e CPU na multiplicação de matrizes

Dimensão	Tempo GPU (ms)	Tempo CPU (ms)	Speedup (×)
3×3	1,992	0,004288	0,00215
1000×1000	17,203	2.235,564	129,95

Fonte: Elaborado pelos autores

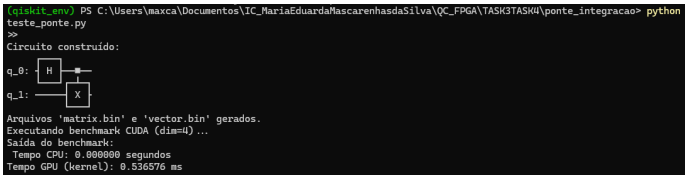

Figura 2. *Benchmark* da multiplicação de uma matriz por vetor complexo (dimensão 4×4).

Portanto, embora o *benchmark* atual não mostre aceleração para pequenas dimensões, ele valida a implementação. Os resultados apresentados na Tabela 2 mostram que, à medida que a escala do problema cresce, a paralelização em GPU passa a oferecer ganhos de desempenho substanciais. Assim, ao avançar para a simulação de sistemas quânticos maiores, o projeto consiga explorar o potencial da GPU para obter ganhos de desempenho expressivos.

6. Conclusão Parcial e Perspectivas Futuras

Este trabalho demonstrou a viabilidade de uma arquitetura clássico-quântica híbrida, construindo um *backend* local integrado ao `Qiskit`. A API RESTful em `Python` com `Flask` automatizou a gestão de objetos `Qobj`, e um `Provider`/`Backend` customizados viabilizaram o encaminhamento de circuitos.

Um pilar foi a otimização de operações quânticas via GPU. Com `CUDA` e `cuComplex`, desenvolveu-se um sistema para multiplicação de matriz por vetor. Os resultados experimentais são um teste inicial, demonstrando a capacidade de aceleração da GPU para certas operações. Embora preliminares, os dados são promissores: a GPU superou a CPU em larga escala, com um *speedup* de aproximadamente 130× para matrizes 1000×1000, reforçando o potencial da computação paralela em simulações quânticas.

Futuramente, a pesquisa expandirá o sistema, implementando o produto de Kronecker ($C = A \otimes B$) em `CUDA`. Serão realizados testes mais robustos para uma avaliação completa. No entanto, em seu estágio inicial, não é possível comprovar todas as potencialidades, pois o trabalho e sua base experimental serão expandidos. O objetivo é consolidar a arquitetura, robustecendo a integração `Qiskit`/GPU, para criar um ambiente de simulação remoto, rápido e confiável.

Atualmente, o foco é a integração Qiskit com o módulo GPGPU. O servidor de simulação proposto incluirá enfileiramento de prioridade, agendamento de jobs, despacho inteligente e drivers para hardware especializado. Este trabalho integra um projeto maior, onde a infraestrutura será expandida para abarcar toda a cadeia de processamento — do recebimento de Qobjs via WebServices à execução em variadas arquiteturas de aceleração — consolidando a arquitetura de um backend independente.

Referências

Company, M. . (2024). What is quantum computing? `McKinsey.com`.

IBM (2023). Qiskit: An open-source framework for quantum computing. `https://qiskit.org`.

Preskill, J. (2018). Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79.

Raz, R. (1998). A parallel repetition theorem. *SIAM Journal on Computing*, 27(3):763–803.

Sarode, K. (2024). Circuit partitioning and full circuit execution: A comparative study of gpu-based quantum circuit simulation. Cited by: 0; All Open Access, Green Open Access.

Schieffer, G., Wahlgren, J., Ren, J., Faj, J., and Peng, I. (2024). Harnessing integrated cpu-gpu system memory for hpc: A first look into grace hopper. Cited by: 3; All Open Access, Green Open Access, Hybrid Gold Open Access.

Schmidt, B. and Hildebrandt, A. (2024). From gpus to ai and quantum: three waves of acceleration in bioinformatics. *Drug Discovery Today*, 29(6):103990.