Portabilidade, Performance e Produtividade Demonstração de sistemas runtime em Julia

João Rafael Guimarães[1], **Felipe de Alcântara Tomé**[2]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCAR)
São Carlos - SP - Brazil

[2]Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, USA.

joaoguimaraes@estudante.ufscar.br, felipe0@mit.edu

Abstract. *The development of portable, high-level and performant GPU code is a reality in Julia. The language uses the Just-In-Time (JIT) compilation technique and enables the creation of extremely optimized machine code. With the use of widely documented libraries such as KernelAbstractions.jl and Dagger.jl, it is possible to execute the same code in different backends and abstract traditionally complex aspects of parallel development. In this work, we sought to exploit the advantages of Julia when applying a Gaussian Elimination algorithm, a widely known method for solving linear systems.*

Resumo. *O desenvolvimento de código de GPU portátil, de alto nível e performático é uma realidade em Julia. A linguagem utiliza técnica de compilação Just-In-Time (JIT) e permite a criação de código de máquina extremamente otimizado. Com o uso de bibliotecas amplamente documentadas como KernelAbstractions.jl e Dagger.jl, é possível executar um mesmo código em diferentes backends, e abstrair aspectos tradicionalmente complexos do desenvolvimento paralelo. Neste trabalho, buscou-se explorar as vantagens do Julia durante a aplicação um algoritmo de Eliminação Gaussiana, método amplamente conhecido para resolução de sistemas lineares.*

1. Introdução

No cenário atual de desenvolvimento, é comum enfrentar a escolha entre ambientes de alto nível, que favorecem a produtividade mas comprometem o desempenho, e linguagens de baixo nível, que oferecem maior velocidade de execução às custas de uma complexidade maior. A linguagem Julia, que utiliza a compilação *Just-In-Time* (JIT) para reduzir o *overhead* comum à outras linguagens de alto nível (Python, R, etc.), busca justamente resolver este problema: Proporcionar uma plataforma de desenvolvimento onde desempenho e facilidade se possam encontrar [Bezanson et al. 2017].

Este trabalho busca evidenciar a facilidade e portabilidade do desenvolvimento em Julia por meio da implementação de um algoritmo de Eliminação Gaussiana. Através de bibliotecas disponíveis e amplamente documentadas como *KernelAbstractions.jl* e *Dagger.jl*, foi possível trabalhar a paralelização do problema de complexidade $O(n^3)$ e execut-

lo em tempo similar à implementações em C com Cuda. O código utilizado foi desenvolvido no ambiente do *google colab*[1] e está disponível para acesso.

2. Trabalhos Relacionados

O problema de eliminação gaussiana foi amplamente explorado na academia. Em [Milanez et al. 2024], apresenta-se um estudo da implementação desse algorítmo utilizando OpenMP e Cuda, que demonstra a superioridade em tempo de execução do uso da GPU. Os resultados mostram que ambas as estratégias superam a versão sequencial em desempenho, sendo que a versão em CUDA apresentou os maiores *speedups*, alcançando 7.84x para matrizes de 2048×2048. Vale ressaltar que essa implementação contabiliza a alocação da matriz e não apenas a eliminação gaussiana. Este fato justifica a discrepância entre o *speed-up* encontrado neste e naquele trabalho.

Ademais, o tópico que toca sistemas runtime de paralelismo automático e heterogêneo é uma pedra angular na comunidade de computação de alto desempenho, como visto na popularidade dos sistemas IRIS [Kim et al. 2021], OpenMP [Dagum and Menon 1998] e Dagger [Alomairy et al. 2024].

3. Eliminação Gaussiana

A eliminação gaussiana é um método amplamente conhecido para a resolução de sistemas lineares, dados por $AX = B$ [Ali Naqvi et al. 2015]. Sua execução se baseia em dois passos simples:

- Eliminação: transformar o sistema original em uma forma triangular superior.
- Retrosubstituição: resolver o sistema triangular obtido, em ordem reversa.

Vale notar que o algoritmo é instável e, como descrito em [Miller 1972], o pivoteamento parcial é necessário para garantir a estabilidade numérica. Além disso, o mesmo trabalho mostra como pequenos arredondamentos podem acarretar grandes divergências nos resultados, de modo que este método deve ser aplicado com cautela em grandes amostras de dados.

4. Julia e *KernelAbstractions.jl*

Segundo a definição oficial, *KernelAbstractions.jl* é uma biblioteca que permite que se escreva *kernels* de GPU cujos alvos sejam diferentes ambientes de execução [Churavy 2024]. Em outras palavras, é uma plataforma de desenvolvimento que permite que um mesmo código de kernel funcione bem em múltiplas plataformas, inclusive em CPU. Atualmente, a biblioteca possui suporte para *NVIDIA CUDA*, *AMD ROCm*, *Intel oneAPI* e *Apple Metal*.

Para a implementação da eliminação gaussiana, foram criados dois *kernels* específicos, responsáveis pelas funções de pivoteamento parcial e eliminação de fatores. A resolução do sistema final, com a matriz A já triangulizada, foi realizada de modo sequencial. A execução do mesmo código (`main_KA!()`) com diferentes matrizes passadas como parâmetro, mostra um *speedup* de 20,5 para uma matriz A quadrada de tamanho 2048. A linguagem Julia mostra uma de suas grandes vantagens: a única diferença entre a execução dos dois códigos é a alocação inicial das matrizes envolvidas.

[1] https://colab.research.google.com/drive/1TQ0lcVugn2t8h7fsS8mW68C1jceqrHfo?usp=sharing

Alocando as matrizes na GPU, obteve-se um tempo de execução satisfatório (1,330 segundos). Com a alocação em CPU observa-se um tempo de execução elevado (27,277 segundos). Vale dizer que os testes em GPU foram executados no ambiente do *Google Colab*, com *NVIDIA Tesla T4* e ambiente de CPU utilizado foi *Intel – 12th Gen Core i7-1255U* de 12 *cores*.

Finalmente, vale ressaltar que a alocação das matrizes não é considerada no cálculo, apenas a execução do algoritmo sobre matrizes já alocadas. Além disso, ao utilizar o backend CPU() do *KernelAbstractions.jl*, os *kernels* são executados em paralelo utilizando as *threads* disponíveis no ambiente Julia. Isso é feito através da divisão do espaço de iteração entre as threads, como otimização automática da biblioteca. [Churavy 2024].

5. Julia e *Dagger.jl*

A biblioteca Dagger.jl age como um runtime system dedicado a administração e abstração de paralelismo, assim como as implementações ParSEC, IRIS, entre outras. O objetivo principal destas abstrações é fazer com que o usuário final não tenha que lidar explicitamente com a comunicação e administração de recursos para utilizar dos benefícios das novas arquiteturas massivas e heterogêneas de computação.

A implementação pode ser feita como um implementação sequencial com algumas anotações indicando a dependência entre os dados e as funções, similar ao modelo *task-depend* do runtime OpenMP, uma implementação paralela e distribuída, in-node ou multi-node pode ser feita em minutos caso o código sequencial já esteja pronto.

Como pode ser visto na implementação fornecida, é feito um particionamento automático do array utilizado através da função *distribute*, e o código é alterado para comportar o ambiente *datadeps*, o qual permite a anotação de dependências de leitura *In*, escrita *Out* e ambos *InOut*. Feitas as anotações e a distribuição o runtime system lida com comunicação através do código adaptado. Neste caso, caso houvesse acesso a um maior numero de GPUs o código funcionaria perfeitamente, sem maior perda de performance e com overhead fixo.

Um dos autores está implementando a otimização destes passos de comunicação através da utilização do padrão MPI, pois a atual implementação da biblioteca apenas faz uso de comunicação TCP. Apesar de ainda ser um trabalho em execução resultados preliminares serão apresentados eventualmente.

6. Conclusão

Este trabalho evidencia as vantagens de sistemas runtime em Julia. A partir da implementação de um algorítmo de eliminação gaussiana, fica claro como o processo de desenvolvimento se torna mais leve, rápido e, portanto, capaz. Conclui-se que é possível atingir altas métricas de desempenho com código de alto nível, portável para diferentes *backends* e distribuído entre diferentes nós.

Referências

Ali Naqvi, S. R., Gharib, S., Khan, R., Munir, N., and Khanam, M. (2015). System of linear equations, guassian elimination. *Global Journal of Computer Science and Technology*, 15.

Alomairy, R. M., Tome, F., Samaroo, J., and Edelman, A. (2024). Dynamic task scheduling with data dependency awareness using julia. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98.

Churavy, V. (2024). Kernelabstractions.jl.

Dagum, L. and Menon, R. (1998). OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

Kim, J., Lee, S., Johnston, B., and Vetter, J. S. (2021). Iris: A portable runtime system exploiting multiple heterogeneous programming systems. In *Proceedings of the 25th IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE.

Milanez, V., Navas, I., Migliatti, J., Miata, M., and Guardia, H. (2024). Um estudo de implementação paralela da eliminação de gauss usando openmp e cuda. In *Anais da XV Escola Regional de Alto Desempenho de São Paulo*, pages 29–32, Porto Alegre, RS, Brasil. SBC.

Miller, W. (1972). On the stability of finite numerical procedures. *Numerische Mathematik*, 19(5):425–432.