Paralelização Automática de Código em CUDA Utilizando Aprendizagem por Reforço

Felipe A. L. Soares, Tiago B. Silveira
Humberto T. Marques-Neto, Henrique C. Freitas

Programa de Pós Graduação em Informática
Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
Belo Horizonte – MG – Brasil

{falsoares, tbsilveira}@sga.pucminas.br, {humberto, cota}@pucminas.br

Abstract. *The use of parallel programming becomes increasingly essential, as well as the use of tools that assist the programmer in the tasks of parallelism and obtaining the best performance of the architecture used. Thus, the objective of this work is to create an optimization of code parallelization in C/C++ that uses Reinforcement Learning to perform the automatic parallelization of CUDA code in GPU. The experiments carried out showed that the optimization created is capable of performing automatic parallelization in CUDA, achieving high speedups.*

Resumo. *O uso de programação paralela torna-se cada vez mais essencial, bem como o uso de ferramentas que auxiliam o programador nas tarefas de paralelismo e obtenção do melhor desempenho da arquitetura utilizada. Assim, o objetivo deste trabalho é a criação de uma otimização de paralelização de código em C/C++ que utiliza Aprendizagem por Reforço para realizar a paralelização automática de código em GPU utilizando a linguagem CUDA. Os experimentos realizados mostraram que a otimização criada é capaz de realizar a paralelização automática em CUDA, conseguindo altos speedups.*

1. Introdução

Devido a necessidade de alto desempenho, otimização do tempo de processamento e melhor exploração de recursos computacionais, a prática de programação paralela torna-se cada vez mais importante na área de desenvolvimento de software [Neshatpour et al. 2016]. Assim, as ferramentas que auxiliem o desenvolvedor de códigos nas tarefas de paralelismo e obtenção de alto desempenho apresentam-se como uma excelente alternativa diante dos desafios relacionados a arquitetura, hardware, compilador e conhecimentos da aplicação e dos algoritmos [Soares et al. 2019].

Uma das maneiras de desenvolver ferramentas dessa natureza é utilizar técnicas de Aprendizado de Máquina. Dentre essas, existe a abordagem conhecida como Aprendizagem por Reforço, em que agentes inteligentes tomam decisões em um ambiente com o intuito de maximizar o acúmulo de recompensas recebidas a longo prazo [Mnih et al. 2013].

Diante desse contexto, o objetivo desse trabalho é a criação de uma otimização de paralelização de código em C/C++ que utiliza a abordagem Aprendizagem por Reforço para realizar a paralelização automática de código em GPU utilizando a linguagem

CUDA. Para validar o sistema criado, três experimentos foram realizados em diferentes códigos[1] com *loops*.

Este artigo está organizado da seguinte maneira: a Seção 2 apresenta o referencial teórico importante para entendimento do trabalho; a Seção 3 descreve alguns trabalhos relacionados com o tema de pesquisa; a Seção 4 define a metodologia utilizada; a Seção 5 apresenta os experimentos e os resultados; por fim, a Seção 6 conclui o trabalho.

2. Referencial Teórico

Esta seção apresenta conceitos importantes para o entendimento do trabalho.

2.1. Aprendizagem por Reforço

A Aprendizagem por Reforço é uma área do Aprendizado de Máquina que originou-se inspirada na psicologia comportamental, em que estuda-se a criação de agentes inteligentes que tomam decisões em um ambiente, com o intuito de maximizar o acúmulo de recompensas recebidas a longo prazo [Mnih et al. 2013]. Um dos métodos mais utilizados nessa área é o Processo de Decisão de *Markov* (MDP, do inglês *Markov Decision Process*). Segundo [Saffran et al. 2020], esse processo é modelado matematicamente como uma quíntupla S, A, P.(., .), R.(., .), γ)), em que:

- S: Conjunto de estados, pode ser finito ou infinito.
- A: Conjunto de ações que podem ser tomadas pelo agente.
- $P_a(s, s')$ = P r($s_{t+1} = s' \mid s_t = s, a_t = a$): Função que determina a probabilidade de em um determinado momento no tempo t estando no estado s e tomando a ação a o agente transitar para o estado s'.
- $R_a(s, s')$: Função que determina a recompensa imediata recebida pelo agente por ter realizado a ação a em um estado s e ter transitado para o estado s'.
- $\gamma \in [0, 1]$: constante conhecida como Fator de Desconto. Determina a diferença de importância entre recompensas recentes e futuras.

Após o mapeamento, é necessário a aplicação do algoritmo para determinar qual ação tomar em cada estado para maximizar o total de recompensas. Um dos mais conhecidos e utilizados é o *QLearning*, que aprende uma distribuição de probabilidades para cada estado do MDP, indicando quais ações são mais prováveis de retornar a maior recompensa [Mnih et al. 2013]. A propriedade do MDP pode ser definida pela fórmula:

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, S_2, ..., S_t) \tag{1}$$

Onde S representa os estados do agente e a probabilidade do agente passar para o estado S_{t+1}, dado que ele está no estado S_t, é igual a probabilidade do agente passar para o estado S_{t+1} dado todos os estados em que o agente esteve no passado.

3. Trabalhos Correlatos

[Gonçalves et al. 2014] desenvolveram um ambiente de software que aponta possíveis oportunidades de paralelização em códigos escritos na linguagem C, gerando esse código

[1]https://github.com/cart-pucminas/cuda-automatic-parallelization

paralelizado para GPU usando a linguagem CUDA. [Baskaran et al. 2010] também desenvolveram um sistema para paralelização automática de código em C, transformando o código de sequencial para paralelo em *CUDA* (paralelização em GPU). Os código paralelizados pelo sistema foram comparados com as versões paralelizadas manualmente para uma série de *benchmarks*, sendo que os resultados foram próximos em suas versões.

[Di et al. 2012] descreveram uma estrutura de compilador para agrupar e paralelizar *loops* com dependência uniforme em código CUDA, eliminando dependências falsas que impedem o paralelismo para maximizar o paralelismo entre blocos. A abordagem foi comparada em oito *benchmarks* diferentes, atingindo acelerações de até 5,5 vezes. Já [Moreira et al. 2016] criaram um sistema capaz de realizar a paralelização automática do código usando a análise estática. Os autores utilizaram as diretivas *OpenACC* e várias regras criadas manualmente para a paralelização do código.

Por fim, [Saffran et al. 2020] propõem um algoritmo de paralelização automática de código em nível de instrução, obtendo um *speedup* médio de 1,6. Esse trabalho utiliza Aprendizagem por Reforço como mecanismo para obter a paralelização do código utilizando as diretivas *OpenMP*. Os autores destacam que utilizar essa mesma abordagem para paralelização em GPU com CUDA é uma oportunidade de trabalhos futuros.

Diferentemente do trabalho proposto, os trabalhos correlatos não utilizam Aprendizagem por Reforço como abordagem para obter a paralelização em CUDA de um determinado código.

4. Metodologia

O algoritmo definido para a paralelização automática de códigos em C/C++ utilizando Aprendizagem por Reforço foi o *QLearning*. A abordagem criada realiza a paralelização automática em nível de dados seguindo o método proposto na Figura 1.

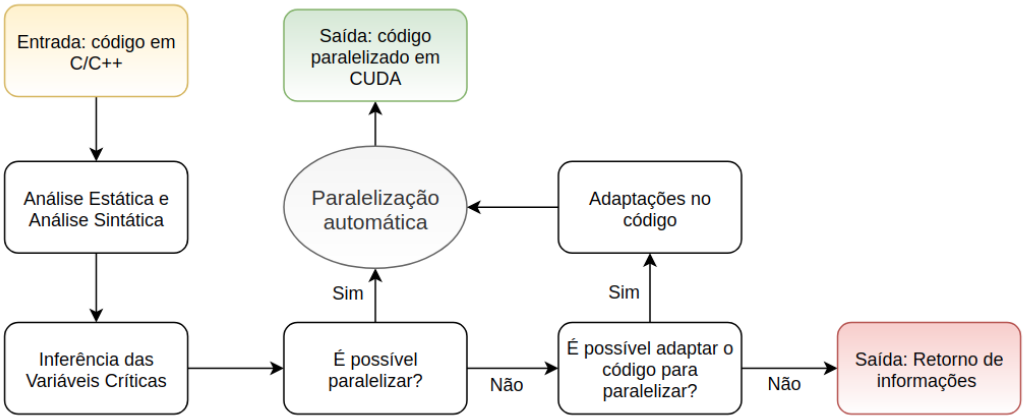

Figura 1. Etapas para realização do método proposto

Para os cenários onde é possível realizar a paralelização em CUDA, inicialmente, é necessário fazer transformações no código em C/C++ visando alterar a sua estrutura em uma estrutura do código CUDA. Para isso, é criado a função *Kernel* para execução em GPU e a substituição do *for* que será paralelizado pela chamada desse *Kernel*. O escopo de código CUDA resultante das alterações para criação do *Kernel* e as variáveis críticas

do código (obtidas por meio da análise estática e sintática) são as entradas para a execução das próximas etapas, responsáveis por aplicar alterações nas variáveis críticas do código, como transformação das variáveis para execução em CUDA e alterações como uso de *Atomic* (soma, máximo, mínimo e afins).

Essas possibilidades de alterações são utilizadas no *QLearning*. Para isso, é definido um Processo de Decisão de *Markov* (MDP, do *inglês Markov Decision Process*), permitindo definir os comportamentos tanto do mundo quanto do agente, criando um ambiente onde o agente pode aprender e maximizar o recebimento de recompensas a longo prazo. O MDP foi definido para a paralelização como:

- S: conjunto de possibilidades para paralelização em CUDA.
- A: as transformações de variáveis permitidas pelo CUDA.
- $P_a(s, s')$ = P r($s_{t+1} = s' \mid s_t = s, a_t = a$): essa função é constante, sendo a probabilidade igual a 1, uma vez que esse ambiente é totalmente determinista.
- R_a(s, s'): função está definida na Figura 2. Para o cálculo da recompensa são definidas 4 variáveis derivadas de um estado: $T_s s$ tempo de execução do código sequencial, $T_s p$ tempo de execução do código paralelo gerado em um estado s, $R_s s$ resultado da execução do código sequencial e $R_s p$ resultado da execução do código paralelo gerado em um estado s.
- $\gamma \in [0, 1]$: o valor foi definido de maneira empírica (em 0,95) pelos autores [Saffran et al. 2020].

Para a a paralelização, foi utilizada como R_a a equação da Figura 2.

$$R_a(T_s s, T_s p, R_s s, R_s p) = \begin{cases} -1, & T_s p > T_s s \\ -1, & R_s p \neq R_s s \\ speedup(T_s s, T_s p), & (T_s s, T_s p, R_s s, R_s p) \in \mathbb{R}, \end{cases}$$

Figura 2. Equação que define a recompensa recebida em um determinado estado após tomar uma ação

Como o método pode ser demorado para encontrar o código paralelo, o sistema utiliza uma heurística para parar a execução caso o resultado seja correto e o *speedup* seja superior ao parâmetro fornecido pelo usuário. Após o fim da execução do sistema, há a função *Q* definida. Assim, partindo do estado inicial da matriz *Q* e executando as melhores ações para cada estado, obtém-se o código paralelizado que gerou maior ganho de desempenho para a paralelização em CUDA.

5. Experimentos e Resultados

Os testes de desempenho foram realizados em um ambiente *Intel® Core™ i5* 9400f, com frequência de 2.90 *GHz*, 6 núcleos, 6 *threads* sem suporte SMT (do inglês *Simultaneous Multithreading*), 16 GB de memória *RAM* e com a placa de vídeo *Geforce GTX 1660* com 6 GB. O sistema operacional utilizado foi o *Linux Ubuntu* 20.10 64 *bits*.

Para o primeiro experimento, foi realizada a paralelização automática de um código com um *For*, que possui outro *For* e realiza a mesma atribuição em cada uma das iterações, representado em Algoritmo 1, sendo que a quantidade de repetições dos dois *For* foi parametrizada com n = 110000, totalizando 110000x110000 repetições.

```
for (i = 0; i < n; i++) do
    for (j = 0; j < n; j++) do
        resultado = 2;
    end
end
```

Algorithm 1: Pseudocódigo do primeiro trecho paralelizado

Para o segundo experimento, foi realizada a paralelização automática de um código com um *For*, que em cada interação realiza uma atribuição de incremento a uma variável (Algoritmo 2). A quantidade de repetições do *For* parametrizada com n = 20000.

```
resultado = 0;
for (i = 0; i < n; i++) do
    resultado += 2;
end
```

Algorithm 2: Pseudocódigo do segundo trecho paralelizado

Para o terceiro experimento, foi realizada a paralelização automática de um código com um *For* que realiza cálculos e uma atribuição de incremento a uma variável a cada iteração realizada, com o objetivo de encontrar o valor de *pi* (3,1415...), representado no Algoritmo 3, sendo a quantidade de repetições do *For* parametrizada com n = 200000000.

```
resultado = 0;
passo = 1/n;
for (i = 0; i < n; i++) do
    resultado += 4/(1+((i+0.5)*passo)*((i+0.5)*passo)) ;
end
pi = resultado*passo;
```

Algorithm 3: Pseudocódigo do terceiro trecho paralelizado

Diferentemente do primeiro, o segundo e terceiro códigos possuem incrementos em uma variável a cada iteração, assim a paralelização em GPU precisou usar *Atomic*. Além disso, o terceiro experimento ainda possui mais variáveis utilizadas dentro do *For* do que o segundo experimento, podendo apresentar um desafio maior na paralelização automática. Para todos os experimentos, a abordagem conseguiu paralelizar em CUDA e encontrar a mesma solução que a versão sequencial em um tempo de processamento menor. A Tabela 1 ilustra o resumo dos resultados e o tempo de execução do *QLearning*, sendo que o cálculo do *speedup* foi realizado dividindo o tempo de execução em sequencial pelo tempo de execução da versão paralelizada (CUDA).

Tabela 1. Resultados encontrados nos experimentos

Experimento	Sequencial(s)	CUDA(s)	Speedup	QLearning(s)
1	1775,3067	0,1260	14091,6105	49,2729
2	71,3697	0,9285	76,8641	3,1737
3	8,6178	0,7356	11,7145	107,3719

O Experimento 1 foi o que teve o maior *speedup* como resultado, chegando a 14091,6105, principalmente devido ao fato de ter uma quantidade maior de iterações e, consecutivamente, ter o maior tempo de execução em sequencial. Inclusive, por ser um código relativamente simples, a paralelização do código desse experimento resultou em

uma execução com menor tempo de processamento entre todas as paralelizações. Além disso, vale destacar que a execução do QLearning foi mais demorada no Experimento 3, pois o código desse experimento é o mais complexo entre os três experimentos, implicando em mais dificuldade para paralelização automática. Por fim, vale ressaltar que para todos os experimentos, a abordagem teve sucesso na paralelização em CUDA, alcançando altos *speedups* em um tempo relativamente pequeno para executar o *QLearning*.

6. Conclusão

O objetivo desse trabalho foi a criação de uma otimização de paralelização de código em C/C++ que utiliza Aprendizagem por Reforço para realizar a paralelização automática de código utilizando a linguagem CUDA (para GPU). Assim, o sistema desenvolvido recebe como entrada o código a ser paralelizado e é capaz de transformá-lo para uma versão de execução em GPU, bem como encontrar as melhores alternativas para essa paralelização.

Os três experimentos realizados mostraram que a otimização criada é capaz de realizar a paralelização automática para GPU, conseguindo altos *speedups* e executando em tempo relativamente baixo, tendo potencial para ser uma boa alternativa para auxiliar os programadores no desenvolvimento de códigos paralelos. Como trabalhos futuros, sugere-se a aplicação do sistema em *benchmarks* e códigos que demandem mais tempo de processamento e a realização de experimentos comparativos com os trabalhos correlatos.

Referências

Baskaran, M. M., Ramanujam, J., and Sadayappan, P. (2010). Automatic c-to-cuda code generation for affine programs. In Gupta, R., editor, *Compiler Construction*, pages 244–263, Berlin, Heidelberg. Springer Berlin Heidelberg.

Di, P., Ye, D., Su, Y., Sui, Y., and Xue, J. (2012). Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus. In *2012 41st International Conference on Parallel Processing*, pages 350–359.

Gonçalves, C. O., Spolon, R., Lobato, R. S., Manacero, A., and Lobato, D. C. (2014). Automatic loops parallelization. In *2014 9th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–5.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

Moreira, K. C. A., Mendonça, G. S. D., Guimarães, B., Alves, P., and Pereira, F. M. Q. (2016). Paralelização automática de código com diretivas openacc. *SBLP. SBC*.

Neshatpour, K., Sasan, A., and Homayoun, H. (2016). Big data analytics on heterogeneous accelerator architectures. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2016 International Conference on*, pages 1–3. IEEE.

Saffran, J., Rocha, R. C., and Góes, L. F. (2020). Neuromp: Paralelização automática de código utilizando aprendizagem por reforço. In *Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 85–94, Porto Alegre, RS, Brasil. SBC.

Soares, F., Nobre, C., and Freitas, H. (2019). Parallel programming in computing undergraduate courses: a systematic mapping of the literature. *IEEE Latin America Transactions*, 17(08):1371–1381.