Paralelização Automática de Laços

Automatic Loops Parallelization

Cristiano Oliveira Gonçalves, Roberta Spolon
Departamento de Computação, Faculdade de Ciências
Universidade Estadual Paulista
Bauru, Brazil
roberta@fc.unesp.br

Renata Spolon Lobato, Aleardo Manacero Jr.
DCCE/IBILCE
Universidade Estadual Paulista
São José do Rio Preto, Brazil
renata@ibilce.unesp.br, aleardo@ibilce.unesp.br.

Daniel Correa Lobato
Instituto Federal de Educação, Ciência e Tecnologia de São Paulo
Catanduva, Brazil
daniel@lobato.org

Resumo – **Identificar as oportunidades de paralelismo em software é uma tarefa que consome muito tempo humano, mas uma vez que sejam reconhecidos os padrões de código que caracterizam o paralelismo, um computador poderia realizar rapidamente essa tarefa. Assim, a automatização deste processo traz diversos benefícios, como a economia de tempo e a diminuição de erros causados pelo programador [1]. Este trabalho tem como objetivo o desenvolvimento de um ambiente de *software* que identifica oportunidades de paralelismo em um código-fonte escrito em linguagem C, e gera um programa com o mesmo comportamento, porém com maior nível de paralelismo, e compatível com um processador gráfico que apresente arquitetura CUDA.**

Palavras Chave: CUDA, GPU, paralelização de laços

Abstract — **Identify opportunities for software parallelism is a task that takes a lot of human time, but once some code patterns for parallelism are identified, a software could quickly accomplish this task. Thus, automating this process brings many benefits such as saving time and reducing errors caused by the programmer [1]. This work aims at developing a software environment that identifies opportunities for parallelism in a source code written in C language, and generates a program with the same behavior, but with higher degree of parallelism, compatible with a graphics processor compatible with CUDA architecture.**

Keywords: CUDA, GPU, loops parallelization

I. INTRODUÇÃO

A demanda por capacidade de processamento tem aumentado cada vez mais, devido a fatores como aumento de complexidade do software, popularização do uso de computadores e novas aplicações para a computação na resolução de problemas. Apesar disso, a literatura aponta que a indústria de hardware atingiu uma barreira: não é mais economicamente viável ganhar desempenho computacional através do incremento da frequência dos processadores, pois o consumo de energia e o sobreaquecimento são decorrentes desta técnica.

Esta limitação justifica e fomenta o interesse no estudo da computação paralela, aquela onde um problema pode ser dividido em várias partes processadas concorrentemente.

Tecnologias que viabilizam o paralelismo por hardware adicionam novas possibilidades de ganho de desempenho, desde que os programas sejam escritos de forma apropriada. Porém, este tipo de paralelismo não elimina a necessidade que a indústria tem de manter seus sistemas legados, que são mais exigidos devido ao aumento da demanda por processamento, e incompatíveis com boa parte dessas novas tecnologias. Desta

forma, a paralelização por software se faz uma ótima alternativa.

Identificar as oportunidades de paralelismo em software é uma tarefa que consome muito tempo humano, mas uma vez que sejam identificados os padrões de código que caracterizam o paralelismo, um computador poderia realizar rapidamente a tarefa. Assim, a automatização deste processo traz benefícios como a economia de tempo e a diminuição de erros.

Diversas são as razões pelas quais o interesse na computação de dados em unidades de processamento gráfico, com objetivo diferente da exibição de gráficos, fica cada vez mais popular. Segundo [2, 3], a programação de propósito geral em GPU (*Graphic Processing Unit*) possibilita o acesso a um alto poder computacional e largura de banda para acesso à memória, além da facilidade de programação.

Esta facilidade se deve aos modelos de programação criados para permitir que o programador escreva código para a GPU em um ambiente familiar, em linguagem C, de forma a abstrair os detalhes do hardware do dispositivo. São recorrentes na literatura a arquitetura CUDA [4] e o *framework* OpenCL [5], embora existam outros modelos cujo propósito é facilitar a exploração do paralelismo contido nas GPUs.

Este trabalho tem como objetivo apresentar um gerador de aplicação que identifica oportunidades de paralelismo em um software escrito de forma linear, e gera um programa com o mesmo comportamento, porém com maior nível de paralelismo, e pronto para executar num processador gráfico que apresente arquitetura CUDA.

O texto está dividido da seguinte forma: na seção II apresentam-se conceitos relacionados com a paralelização de laços; na seção III é descrita a estrutura do gerador de aplicação desenvolvido; na seção IV são apresentados resultados e conclusões do trabalho.

II. Paralelização de Laços

É possível alcançar o paralelismo tanto por *hardware* quanto por *software*. Segundo [1], o processo de computar dados concorrentemente mantendo a estrutura linear de programação é uma solução não escalável e não flexível, e a modificação dos sistemas já escritos é muito custosa. Portanto, a paralelização automática é uma boa alternativa para se obter benefícios significativos das novas arquiteturas de computadores.

As dependências de dados contidas em um programa dificultam ou impedem a sua divisão entre linhas de execução concorrentes. Elas funcionam como barreiras para o paralelismo, pois, enquanto algumas dependências podem ser eliminadas do programa, outras simplesmente não podem.

Quando há uma decisão a ser tomada com base em outra instrução dentro do programa, encontra-se a dependência de controle. É o que acontece, por exemplo, quando há uma condição do tipo *if{...}else{...}* [1].. Neste caso, a decisão que o programa tomará depende dos parâmetros do argumento *if.*

A dependência de dados ocorre quando duas instruções acessam o mesmo endereço de memória, e pelo menos uma delas escreve neste endereço [6]. O modo como esse evento pode acontecer varia bastante, pois a ordem em que essas instruções aparecem no código modifica o tipo da dependência. Considerando que as instruções podem realizar leitura ou escrita num dado endereço de memória, podemos dividir as dependências da seguinte forma[1, 6]:

- *Readafter Write (RAW)*:também conhecida como dependência verdadeira, ela ocorre quando uma instrução lê um endereço de memória que é escrito por outra instrução. Por exemplo, sejam $a{:}W = X + Y$ e $b{:}Z = W + 15$ as instruções *a* e *b*. A instrução *a* escreve no endereço de memória W o valor da soma de duas variáveis. Na instrução *b*, esse valor é lido, fazendo com que não exista uma forma de alterar a ordem destas instruções, ou computá-las paralelamente, sem que o comportamento do programa seja alterado. Isso não significa, porém, que é impossível paralelizar instruções com esse comportamento. Se W, X, Y e Z forem vetores no mesmo Loop, é possível paralelizá-los usando CUDA, contanto que se mantenha a ordem de ocorrência das instruções.

- *Write afterRead (WAR):*este tipo de dependência ocorre quando um endereço de memória é lido para, em seguida, ser modificado. Isso resulta num valor desatualizado do dado na instrução que faz a leitura, e esse comportamento é conhecido como antidependência. Por exemplo, sejam
  $a{:}\ W = X + Y$
  $b{:}X = Z + 15$
  Neste caso, a instrução *a* lê o valor de X, e o atualiza na instrução *b.* Felizmente, este tipo de dependência pode ser facilmente tratado com a substituição da variável X por uma variável qualquer, que não pertença a outra instrução no programa.
  No entanto, o propósito da antidependência deve ser estudado, pois se as instruções tiverem dentro de um laço, trocar a variável X e manter o comportamento pode requerer uma instrução a mais, para que a instrução *a* tenha sempre o comportamento esperado.

- *Write after Write (WAW):*esse tipo de dependência acontece quando um mesmo endereço de memória é atualizado por instruções diferentes, ou seja, a (n)ésima instrução sobrescreve a (n-1)ésima. É também conhecida como dependência de saída. Por exemplo: sejam
  $a{:}\ W = X + Y$
  $b{:}W= Z + 15$
  Novamente, a substituição da variável W por outra que não esteja sendo acessada em outras instruções, resolve a dependência de saída.

A paralelização automática de laços de repetição não é possível em todos os casos onde se encontra um laço, pois, como discutido anteriormente, existem dependências que impedem a divisão das instruções contidas no corpo do *loop*.

As dependências deste tipo podem ser difíceis de serem identificadas, pois a repetição de instruções gera comportamentos que não são facilmente previsíveis. Além disso, existe a possibilidade de a dependência não ser gerada em toda iteração de um mesmo laço, fazendo com que este não seja completamente paralelizável.

Para facilitar a análise das dependências, frequentemente usa-se da técnica de normalização dos laços contidos no código a ser tratado [1].

Um laço normalizado é também conhecido como laço bem-comportado (*well-behaved loop*), e consiste em um laço cuja variável de controle começa com valor 1, e recebe um incremento de 1 a cada iteração [1].

Obviamente, alguns laços não normalizados devem manter seu incremento diferente de um, e/ou sua variável de controle com valor inicial maior que 1, para que ele atenda às necessidades da aplicação que o emprega.

Assim, para garantir que os índices que o programa deve acessar não serão alterados pela normalização dos laços, usa-se um índice normalizado em laços malcomportados. O processo de normalização pode ser visto em [1], e no desenvolvimento deste trabalho, serão abordados os laços bem-comportados.

III. GERADOR DE APLICAÇÃO

O gerador de aplicação analisa as dependências existentes nos laços de execução dos códigos de entrada e, se possível, trata-as, para então gerar o código paralelo. Este novo código mantém o mesmo comportamento que seu "código-pai", porém, é executado na *GPU*, através da aplicação de CUDA C [7]. Portanto, o desenvolvimento do gerador foi dividido em dois módulos, a fim de melhorar sua reutilização.

O código de entrada (CE) é o programa original, escrito em linguagem C, que será analisado e transformado pelo gerador.

O código intermediário (CI) é a primeira saída do gerador, que consiste no código de entrada adicionado de informações sobre os laços que devem ser paralelizados. Finalmente, o código de saída (CS) é o produto final do gerador.

O gerador de aplicação pode ser visualizado como uma função que lê um arquivo de entrada e gera um arquivo intermediário, e deste, um arquivo de saída. O tratamento dos arquivos é feito em duas etapas: a geração de código intermediário (GCI) e geração de código de saída (GCS). O gerador, durante a etapa GCI (Geração de Código Intermediário), analisa o CE (Código de Entrada), trata as dependências que identifica, e escreve um código tratado intermediário (CI). Em seguida, durante a etapa GCS (Geração de Código de Saída), o CS (Código de Saída) é gerado a partir do CI. Na Figura 1, os arquivos são representados por retângulos, enquanto os componentes do gerador são representados por círculos. Os componentes são abstrações de fases diferentes do processo de paralelização.

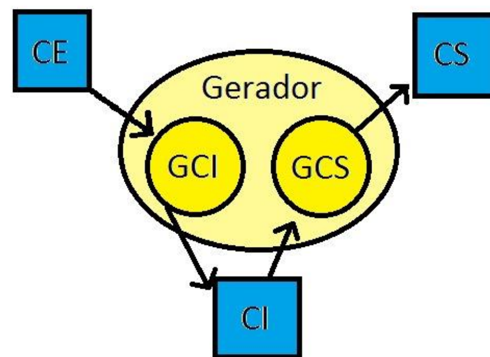

Figura 1: Modelo Gerador em Alto Nível de Abstração

A. *Geração do Código Intermediário*

Para classificar as dependências entre duas instruções, um algoritmo deve ser desenhado de forma a analisá-las [5]. Nosso interesse está nas instruções contidas em laços de execução, que devem ser analisados pelo gerador de aplicação para identificar o tipo de dependência presente. Esta análise poderá resultar em quatro casos: não há dependências no laço, há dependências que podemos tratar, há dependências que não podemos tratar, e não podemos analisar o laço. Cada caso é um conjunto de padrões e procedimentos de resposta que foram modelados para auxiliar na construção de um diagrama.

- Não há dependência no laço

Quando o gerador percorrer um laço, e não encontrar um vetor cujos argumentos entre os colchetes são operações aritméticas, entrará neste caso, pois quaisquer que sejam as instruções no corpo do *loop*, este não possui dependências carregadas. Deverá, então, copiar o laço para o arquivo intermediário, sem alterações.

- Há dependências que podemos tratar

Se o laço analisado contiver vetores que possuam operações aritméticas de soma entre os colchetes que descrevem seus índices, ou quando houver sobrescrita de dados num endereço de memória, a cada iteração do laço, existem as dependências do tipo RAW e WAW e, portanto, um tratamento possível. O gerador deverá aplicar o tratamento, criando as novas variáveis e reescrevendo o laço tratado no código intermediário com um comentário que o precede. Nesta aplicação, o comentário usado é "/*CUDA*/".

- Há dependências que não podemos tratar

Na seção II foi descrita a dependência RAW, para que o comportamento de programas com este tipo de dependência seja mantido, a ordem de execução das instruções que configuram este tipo de dependência deve ser mantida. Portanto, será feita a cópia deste laço para o código intermediário, sem alterações.

- Não podemos analisar o laço

Os laços de execução que não são aninhados, mas contêm vetores cujas fórmulas dos índices são complexas

e dinâmicas, introduzem a necessidade de análises muito sofisticadas, além da capacidade do modelo apresentado. A mesma solução adotada anteriormente será utilizada.

Os casos de tratamento citados requerem uma análise completa de alguns aspectos do código a ser tratado. É necessário que tenhamos armazenadas todas as variáveis declaradas no código, para que, ao encontrarmos um laço de execução, tenhamos como identificar quais são os tipos das variáveis que estão sendo usadas nele. Identificar e armazenar as declarações de variáveis existentes no código é tarefa da função *armVars(),* que percorre todo o código de entrada, e adiciona em uma lista os nomes, tipos e tamanhos.

Depois disso, o código de entrada é novamente percorrido de início a fim, linha a linha, pela função *buscaLaço(),* que, ao encontrar um *loop*, executa a função *analiLaço()*, que por sua vez determinará se o laço deve receber algum tratamento para ser paralelizado, ou deverá ser marcado com um comentário que servirá de gabarito para o Gerador de Código de Saída. Na Figura 2 é apresentado o digrama de fluxo de atividades.

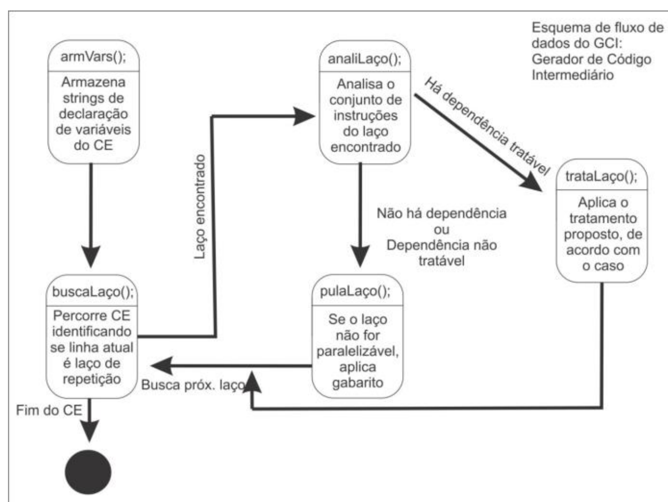

Figura 2: Diagrama de Fluxo de Atividades do GC

B. *Geração do Código de Saída*

Dentro do modelo proposto, o código intermediário é o que deve ser efetivamente paralelizado, pois nele, as oportunidades de paralelismo que o gerador identificou já foram devidamente tratadas. Os comentários presentes nos laços de execução, quando realmente ocorrerem, vão ser usados para determinar se o laço em questão deve ser paralelizado ou não.

As técnicas utilizadas para paralelizar os laços dependem da tecnologia empregada pelo gerador. Em CUDA C, os laços serão reescritos no código de saída como uma função, com o descritor __global__ precedendo sua definição. Como não podemos prever quantos laços existirão no código de entrada, o nome das funções que os representam serão a concatenação da palavra "func", e o número da primeira linha do laço. Por exemplo, suponhamos que a primeira ocorrência de laço aconteça na linha 13. A declaração da função que o representa será "__global__ func13(tipo *arg1, tipo *arg2,..., tipo *argn)". A chamada a essa função será escrita no arquivo de saída através do seguinte modelo: "func13<<<int par1, int par2>>>(&arg1, &arg2,..., &arg3)".

Como apresentado no exemplo anterior, o gerador deverá identificar todas as variáveis e seus respectivos tipos usados no laço de repetição. Os tipos são essenciais, pois a declaração da função que representa o laço exige que os tipos dos parâmetros que recebe sejam conhecidos. Portanto, antes de realizar qualquer escrita no código de saída, o gerador deve percorrer o código intermediário, armazenando todas as declarações de variáveis, como descrito anteriormente.

Algumas linhas de código podem declarar variáveis dentro de parâmetros de funções. Sabemos que não se tratam necessariamente de declarações, mas sim de parâmetros a serem recebidos pela função que os declara. Todavia, estes parâmetros podem ser usados dentro de laços de execução pertencentes à função em questão, o que cria a necessidade de armazená-los. Os parâmetros da função que representa o laço devem ser todas as variáveis manipuladas por ele. Uma vez que o gerador tem todas as declarações de variáveis do código intermediário armazenadas, e os nomes das variáveis usadas no laço, ele pode escrever a função CUDA que representa o *loop*.

Antes da escrita, porém, o cabeçalho do CI deve ser copiado para o CS, de forma a manter no código todas as suas dependências. Neste trabalho, o cabeçalho foi entendido como todo o texto que antecede a primeira declaração de variáveis, funções, estruturas e definições de tipo.

Para escrever a função no código de saída, é preciso verificar se o laço é crescente ou decrescente, pois o número de instâncias concorrentes deve ser o mesmo número de repetições do laço. Sem conhecer o comportamento do laço de execução, não é possível determinar esse valor corretamente, pois o número de repetições se localiza, dentro dos parâmetros do *for*, em posições diferentes da *string* que os contém. Podemos visualizar isso no trecho de código a seguir, onde os laços possuem 15 iterações, mas percorrem os vetores de formas diferentes.

for(iCt = 0; iCt< 15; iCt++){...}
for(iCt = 14; iCt>=0;iCt--){...}

O gerador deverá analisar o último parâmetro do *for,* e a partir daí, armazenar o número de instâncias que deverão ser criadas pela função CUDA C correspondente. Na Figura 3 é apresentado o funcionamento do Gerador de Código de Saída, através de um diagrama de fluxo de atividades.

C. *Interface Gráfica*

A interface gráfica com o usuário torna transparentes as operações envolvendo a análise de dependências e as particularidades da programação em CUDA C. Ela também terá a função de conectar os dois módulos do programa – o GCI e o GCS. Na Figura 4 é mostrado um esquema que representa uma abstração de alto nível de seu funcionamento.

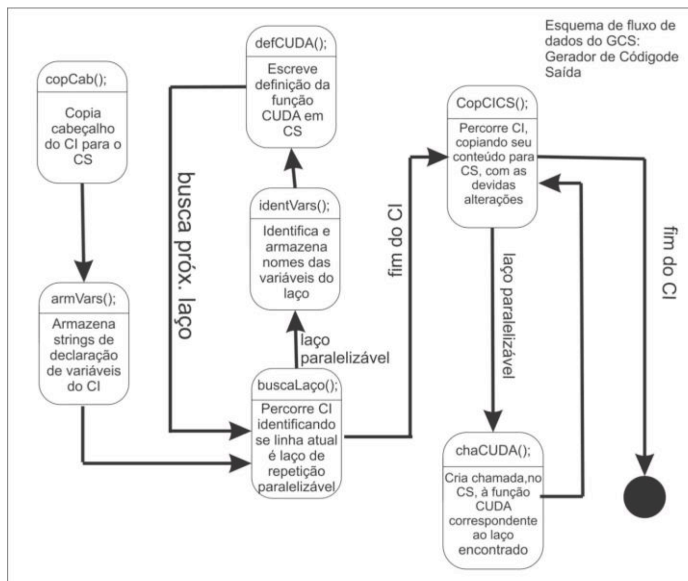

Figura 3: Diagrama de Fluxo de Atividades do GCS

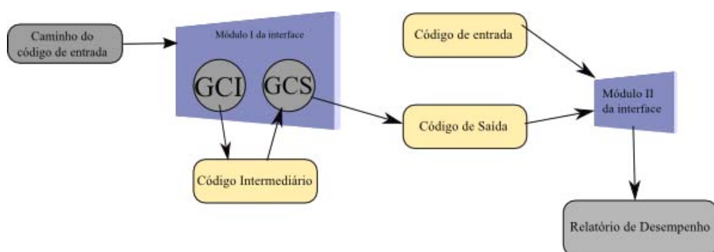

Figura 4: Diagrama da Interface Gráfica

IV. Resultados e Conclusões

Testes foram realizados considerando a operação de multiplicação de vetores, com dimensão de 25 milhões de elementos. A comparação entre o tempo de execução do programa sequencial e o programa obtido pelo gerador foi realizada em um computador com as características: processador Intel Core I7 CPU 870@2,93GHz x8; processador gráfico NVIDIA GTX 465; 7,8 GB RAM; HD 472,3 GB; Sistema operacional Linux Ubuntu 12.04.

Os tempos médios de execução obtidos, considerado a versão sequencial e a versão paralelizada, são respectivamente 0,183485 segundos e 0,197118 segundos (considerando o overhead da transferência de dados entre a memória da GPU e a memória principal). Como pode ser observado, a versão com laços paralelizados obteve melhor desempenho quando comparada com a versão sequencial.

A computação paralela pode diminuir o tempo de execução de programas que antes, eram executados de forma linear. Com a crescente demanda por processamento devido à popularização dos computadores e às novas oportunidades de aplicar a computação para resolver problemas de forma mais eficiente, conseguir lidar com um volume maior de dados e instruções é uma necessidade cada vez mais urgente.

Diversos programas importantes em funcionamento hoje são lineares. Embora não seja inadmissível pensar que, para obter as vantagens que o paralelismo oferece, basta mudar o ensino de programação, esta não deve ser uma tarefa simples, e os resultados possivelmente demorarão para serem percebidos. Como a tecnologia está em constante mudança, as soluções requerem rapidez, e uma alteração na forma como uma hipotética futura geração de programadores escreverá os códigos não resolve, ao menos inicialmente, o problema dos *software*s legados que estão sobrecarregados devido à incapacidade de processar várias tarefas ao mesmo tempo. Assim, a paralelização automática de *software* se faz uma boa saída para estes problemas, pois permite que programas antigos façam uso de tecnologias recentes [1].

A arquitetura CUDA é uma tecnologia que viabiliza o aproveitamento das capacidades de um *hardware* específico, porém poderoso e cada vez mais comum. Este trabalho apresentou um gerador capaz de identificar os laços num código-fonte e, se possível, reescrever um código fonte para gerar um programa que execute os laços no processador gráfico. Foi desenvolvida também uma interface com o usuário, capaz de ampliar as possibilidades de uso do gerador por pessoas que desconhecem as particularidades da programação em CUDA, bem como da computação paralela.

Referências

[1] C. M. VIEIRA, Automatic loops parallelization to multicore architectures. MSc dissertation. Instituto de Computação. Universidade Estadual de Campinas. 2010. Paralelização automática de laços para arquiteturas multicore. Dissertação (mestrado). Instituto de Computação. Universidade Estadual de Campinas. 2010.

[2] M. DIMITROV, M. Mantor and H. Zhou, Understanding software approaches for GPGPU reliability. *In Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*. ACM, Nova York, NY, 94-104.

[3] D. LUEBKE et al., GPGPU: general purpose computation on graphics hardware. *ACM SIGGRAPH 2004 Course Notes*. Los Angeles, CA. 2004. Disponível em: http://doi.acm.org/10.1145/1103900.1103933.

[4] NVIDIA. Parallel Programming and Computing Platform. http://www.nvidia.com/object/cuda_home_new.html. Data de acesso: 20/12/2013.

[5] AMD. OpenCL Zone.http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/. Data de acesso: 2/03/2013.

[6] NVIDIA. Parallel Programming and Computing Platform. http://www.nvidia.com/object/cuda_home_new.html. Data de acesso: 20/12/2013.

[7] NVIDIA; CUDA C Programming Guide. Disponível em: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Acessado em: 05/03/2013.