COMPUTAÇÃO HETEROGÊNEA

Fábio Oliveira Tempesta, Mateus Araújo Cruz

Instituto Federal de Minas Gerais (IFMG) – Campus Bambuí

fabio.oliveira.tempesta@gmail.com, mateuscruz22@gmail.com

RESUMO

Com o passar do tempo, os sistema computacionais precisaram passar por um processo de paralelização dos núcleos de processamento, já que a relação custo-benefício entre poder de processamento de *single-core* e energia consumida chegou próximo ao seu limite. Com isso, a estratégia adotada foi a computação heterogênea, que utiliza vários processadores em uma única aplicação, chamados de aceleradores, gerando um aumento do *speedup* na máquina. O presente trabalho define alguns conceitos básicos da computação heterogênea, referentes à arquitetura e ambientes de programação. Além disso, traz as tecnologias mais usadas no mercado e presume boas perspectivas para futuro.

Palavras-chave: Aceleradores 1. Computação Heterogênea 2. GPUs 3. Paralelismo 4.

1 INTRODUÇÃO

Os ambientes de computação estão se tornando cada vez mais heterogêneos, explorando os recursos de vários microprocessadores multi-cores, como as unidades central de processamento (*Central Process Unit* - CPU), unidades de processamento de gráficos (*Graphics Process Unit* - GPU) e as matrizes de portas programáveis em campo (*Field Programmable Gate Array* - FPGA) (GASTER et al., 2012). Stringhini, Gonçalves e Goldman (2012) fomentam que esta técnica gera um ganho de desempenho, caso seja utilizado satisfatoriamente, e um baixo consumo de energia, implicando em sinais de projetos de chips heterogêneos dos fabricantes Intel, AMD e ARM.

Diante de tanta heterogeneidade, o processo de desenvolvimento de um software eficiente para várias arquiteturas diferentes apresenta uma série de desafio para os programadores (GASTER et al., 2012). "Em GPUs, por exemplo, um mesmo trecho de código, chamado kernel, é replicado em até milhares de *threads* que executam de forma concorrente sobre um grande conjunto de dados."(STRINGHINI; GONÇALVES; GOLDMAN, 2012).

O presente artigo tem como objetivo principal de apresentar como funciona algumas dessas arquiteturas e um breve ensino de programações que permitem que um código sendo executado na CPU possa enviar trabalho para um processador de arquitetura diferente, com maior ênfase nas GPUs. Esses aceleradores não são adequados para todo o tipo de algoritmo, de acordo com Gaster et al. (2012), os códigos tem diferentes comportamentos de carga de trabalho, de controle intensivo (como pesquisa, classificação e análise) até intensivo de dados (como processamento de imagens, simulações e modelagem, e mineração de dados).

Antes do principal objetivo, é importante apresentar os principais conceitos de paralelismo relacionados à computação heterogênea. Começa-se com a terminologia utilizada em HPC (*High Performance Computing*) ou, em português, PAD (Processamento de Alto Desempenho) que serão utilizados ao longo da matéria. Além disso, será mencionado várias vezes algumas estruturas de programação paralela, que é consistido pelo OpenMP (*Open Multi-Processing*) que permite utilizar *threads* na CPU e tem flexibilidade de funcionar juntamente aos ambientes de programação próprios para aceleradores, OpenCL (*Open Computing Language*) que é uma biblioteca de programação heterogênea, e CUDA (*Compute Unified Device Architecture*) que trabalham com programação paralela nas GPUs da NVIDIA (STRINGHINI; GONÇALVES; GOLDMAN, 2012). Por último, mas não menos importante, deve-se conhecer as quatro possíveis classificações de comportamento de um processador, Barney et al. (2010) argumenta que essas classificações é chamada de Taxonomia Clássica de Flynn, que são:

- SISD (*Single Instruction, Single Data*): Não contém nenhum paralelismo, essa classe é dada é representada por computadores que fazem uma unica instrução e um único dado por ciclo de clock. Esta é a forma mais antiga e, até hoje, a forma mais prevalente de computador;
- SIMD (*Single Instruction, Multiple Data*): As unidades de processamento funcionam de forma paralela, mas executando o mesmo fluxo de instruções em todos os ciclos de clock. Esta classe geralmente faz com que cada unidade de processamento opera em um elemento de dados diferente;
- MISD (*Multiple Instruction, Single Data*): As unidades de processamento operam em um único fluxo de dados por fluxos de instruções independentes e separados;
- MIMD (*Multiple Instruction, Multiple Data*): O computador desta classe pode ter um paralelismo total, com os fluxos de dados e instruções independentes, as unidades de processamento consegue operar programas diferentes, independentes que processam entradas diferentes.

2 REFERENCIAL TEÓRICO

Arquiteturas heterogêneas podem ser observadas em diversas aplicações, por consequência disso, Dongarra e Lastovetsky (2009) propõe uma técnica de classificação que divide essas categorias em máquinas paralelas e sistemas distribuídos. Essa divisão resulta nos seguintes tópicos, por ordem crescente de heterogeneidade e complexidade:

- Sistemas heterogêneos projetados por fabricantes específicos;
- Clusters heterogêneos;
- Redes locais de computadores;

- Redes globais de computadores em um mesmo nível organizacional;
- Redes globais de computadores de propósito geral.

O foco deste trabalho é apresentar mais a fundo a arquitetura das GPUs, que por sua vez se encontram no tópico de sistemas heterogêneos projetados por fabricantes específicos. Para isso, a análise será feita sobre os aceleradores, que são dispositivos que trabalham em conjunto com a CPU.

Segundo Borkar e Chien (2011) os microprocessadores evoluíram milhares de vezes nas últimas décadas com o impulso dos transistores descritos na Lei de Moore. É evidenciado ainda que nas próximas décadas novos desafios surgirão, visto que fica cada vez mais difícil aumentar a quantidade de transistores por chip e ainda ter um consumo viável de energia, já que essa complexidade dentro do microchip aumenta significativamente o gasto energético. Logo, uma alternativa mais viável para contornar este obstáculo é a utilização do paralelismo, núcleos heterogêneos e aceleradores. A utilização destes recursos proporciona bastante eficiência de processamento com um gasto de energia viável.

Atualmente, o uso de aceleradores está em alta, pois é uma alternativa de baixo custo e pode trabalhar em conjunto com a CPU, o que melhora o desempenho de aplicações com alta carga de iterações onde não existe dependência exagerada de dados. Dessa forma, a tarefa de otimizar a aplicação a nível de software se torna mais reduzida, visto que o hardware mais adequado para essas tarefas já está presente na máquina.

Stringhini, Gonçalves e Goldman (2012) traz alguns exemplos de aceleradores, suas respectivas arquiteturas e modelos de programação. São eles o Cell BEA (*Cell Broadband Engine Architecture*), os FPGAs (*Field Programmable Gate Array Architecture*) e os GPUs (*Graphics Processing Units*). A seguir, estes aceleradores serão abordados com mais detalhes.

De acordo com Brodtkorb et al. (2010), o Cell BE é um processador heterogêneo que conta com uma CPU (*PEE - Power Processing Element*) e oito núcleos mais simples de propósito específico (*SPE - Synergistic Processing Elements*) encapsulados no mesmo chip. A arquitetura do Cell BEA foi usada, por exemplo, no console do PlayStation 3. O núcleo PEE é típico e muito utilizado, é encontrado em máquinas baseadas em processadores *Power*. Os SPEs, por sua vez são menores e possuem complexidade mais baixa, tendo o intuito de atuar como aceleradores e concentrar a maior carga de processamento do Cell BE. Apesar de serem núcleos com instruções mais simples, os SPEs possuem uma programação complexa, isso porque não possuem hierarquia de cache e sim uma memória local, que deve ser gerenciada diretamente no código. As transferências de memória também são gerenciadas via código. Além disso, possuem seu próprio conjunto de instruções ISA, o que resulta em uma necessidade de compilar seu código separadamente do código do PEE. A arquitetura do Cell BEA já teve várias aplicações, mas não é tão utilizada atualmente.

FPGAs tem sido empregados como aceleradores para processamento de alto desempenho em razão dos avanços obtidos principalmente nas tecnologias de interconexões de alta velocidade (BRODTKORB et al., 2010). Essa arquitetura proporciona alto grau de paralelismo,

mesmo sendo mais simplificada, pois consiste em um conjunto de blocos lógicos que são reconfiguráveis e blocos de processamento digital de sinais que são conectados por uma interconexão reconfigurável e pode ser usada para diversas aplicações dos FPGAs. Se estes chips forem configurados, podem atuar como circuitos integrados para aplicações específicas, sendo que algumas ferramentas de programação para FPGAs são as linguagens VHDL e Verilog.

O principal exemplo de acelerador com grande aplicação nos dias de hoje são as GPUs, que nos primórdios possuíam apenas função gráfica, mas que rapidamente teve seu potencial percebido para processamento genérico. Atualmente os principais fabricantes de GPUs são a NVIDIA e a AMD, e elas são compatíveis com CPUs da Intel ou AMD. Segundo Stringhini, Gonçalves e Goldman (2012), seu paralelismo é do tipo SIMD, então milhares de *threads* poder executar simultaneamente nas centenas de núcleos da GPU. Dessa forma a CPU fica responsável apenas por executar o programa principal e gerenciar os *threads* na GPU. Neste modelo, memória e os dados são transferidos através de um barramento *PCI express*. Além de serem aceleradores bastante eficientes, as GPUs contam com baixo consumo de energia, ao ser comparada com os processares multicores. A principal fabricante de GPUs para processamento de alto desempenho é a NVIDIA e os principais ambientes de programação são CUDA e OpenCL. A programação CUDA só pode ser usada para as placas da NVIDIA e possui um bom desempenho, a programação OpenCL por sua vez, pode ser usado em diversos dispositivos, mas justamente por ter que manter essa portabilidade, possui um desempenho menor.

Se tratando das arquiteturas das GPUs, uma das principais é a arquitetura Fermi da NVIDIA que aceita execução concorrente de *kernels*. "As GPUs são compostas de centenas de núcleos (cores) simples que executam o mesmo código através de centenas a milhares de *threads* concorrentes" (STRINGHINI; GONÇALVES; GOLDMAN, 2012). Ao comparar este modelo com o modelo multicore, este último possui núcleos independentes e completos que conseguem por si só processar os *threads*. O modelo de execução das GPUs é o SIMT (*Single Instruction Multiple threads*), que é uma derivação do termo SIMD. É importante ressaltar ainda que GPUs possuem memória global que pode ser acessada por todos os *threads*.

A lógica por trás da arquitetura Fermi é reservar uma maior quantidade de transistores para as unidades de execução, em oposição a dedicá-los à unidade de controle e até de cache, proporcionando então execução mais rápida. Tal arquitetura conta com 16 SM (*Streaming Multiprocessors*), que por sua vez conta com 32 cores cada, o que resulta em um total de 512 cores. Além disso, possui memória compartilhada que pode ser gerenciada diretamente pelo programador como memória de rascunho e dois níveis de memória cache. Ainda de acordo com Stringhini, Gonçalves e Goldman (2012), cada SM é contém quatro blocos de execução monitorados por duas unidades de escalonamento de *warps*, grupos de 32 *threads* cada, sendo 16 núcleos simples, um bloco de 16 unidade *load/store* e um bloco com 4 unidades de processamento de instruções especiais. Sendo assim, a GPU nada mais é do que um conjunto de microprocessadores ligados por uma memória global e uma memória cache.

Segundo a FERMI... (2009), o escalonamento dos *threads* é feito em grupos de 32 que executam em paralelo (*warps*). Além disso, cada multiprocessador contém dois escalonadores

e duas unidades de despacho de instruções, o que possibilita as execuções concorrentes. Para melhorar o desempenho, o escalonador seleciona dois *warps*, que são independentes, e então é possível despachar uma instrução de cada *warp* para o grupo de 16 cores, sem dependências entre as instruções.

Além da arquitetura Fermi, a NVIDIA possui uma arquitetura mais nova implementada nas GPUs, é a arquitetura Kepler. As principais melhorias estão relacionadas com multiprocessadores, paralelismo dinâmico e tecnologia *hyper Q*. Nessa nova implementação, as GPUs contam com até 15 multiprocessadores (SMX) com 192 núcleos cada. Com isso, a GPU pode disparar a execução de novas *threads* de maneira dinâmica, sincronizar resultados e adaptar ao fluxo de execução sem a necessidade de envolver o programa executado na CPU. Isso quer dizer que, se antes os processos precisavam sempre se comunicar com o programa principal para gerenciamento de dados e manipulação de *kernel*, agora a própria GPU pode fazer essas chamadas. Por fim, a tecnologia *Hyper Q* permite que diferentes *threads* do *host* possam disparar simultaneamente a execução de *kernels*. O intuito disso é que a GPU possa ser utilizada para acelerar, por exemplo, programas escritos para plataformas de troca de mensagens, onde os processos poderão disparar a execução simultânea de *kernels* na GPU, e melhorar p desempenho das aplicações. "Na arquitetura Kepler cada SMX possui quatro escalonadores de *warps* e oito unidades de despacho de instruções, possibilitando que quatro *warps* (4 x 32 *threads* paralelas) possam ser escalonados e executados concorrentemente (*quad warp scheduler*)" (STRINGHINI; GONÇALVES; GOLDMAN, 2012).

3 MERCADO ALVO

A programação paralela, como anteriormente salientado no presente artigo, possui um importante papel na computação para a maioria das aplicações, aumentando seu desempenho significativamente. Para esta causa, além de escolher o hardware adequado, deve-se saber selecionar e estar familiarizado com a determinada estrutura de programação paralela. Para Memeti et al. (2017), o desafio do programador é escolher uma de várias estruturas de programação disponível que cumpra as determinadas metas com um alto desempenho e produtividade. A seguir será apresentado alguns desses ambientes de programação.

3.1 OpenMP

O OpenMP (*Open Multi-Processing*) é um ambiente de programação que tem o objetivo de utilizar *threads* nos processadores, seja CPU ou GPU, gerando um processamento paralelo na máquina. Essa ferramenta possui vantagens como a facilidade de programação e popularização do uso dela para programação paralela de alto desempenho.

Stringhini, Gonçalves e Goldman (2012) expõe algumas características desse sistema:

> O padrão é composto por um pequeno conjunto de diretivas de programação mais um pequeno conjunto de funções de biblioteca e variáveis de ambiente que usam como base as linguagens C/C++ e Fortran. Trata-se de um padrão,

portanto várias implementações estão disponíveis. É comum que compiladores já conhecidos, como o próprio **gcc**, possuam opções de compilação para OpenMP.

Esse paralelismo é feito por meio de *threads*, que é composta basicamente por um contexto (estado do hardware, informações ao SO), o código e uma pilha, compartilhando os segmentos de códigos, dados e *heap*, e podendo executar uma linha de código diferente, pois cada uma dela tem o seu próprio contexto (STRINGHINI; GONÇALVES; GOLDMAN, 2012). No ambiente OpenMP, essas *threads* são criadas a partir de fragmentos de códigos anotados que normalmente correspondem a alguma tarefa iterativa. O processamento se dá com o início de uma *thread* única (*master thread*), executando sequencialmente as instruções até encontrar um código que precisa ser paralelizado, surgindo uma operação semelhante a um *fork*, criando um **time de *threads*** para executar este código, sendo que essa equipe é finalizada quando termina a região paralela, uma operação similar a um *join*, restando apenas a *thread* mestre, continuando avançar nas instruções do programa. A figura 1 ilustra este comportamento passando por duas regiões com códigos que necessitam ser executadas paralelamente.

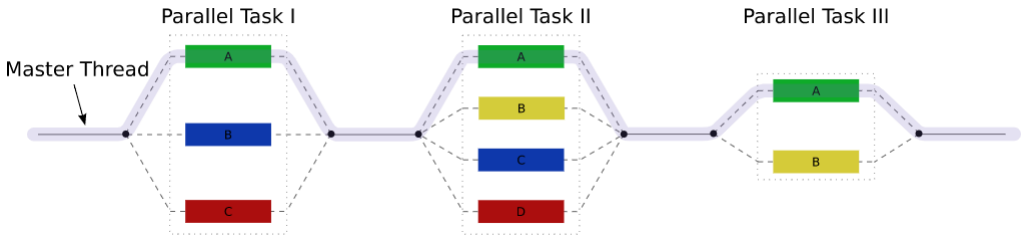

Figura 1 – Modelo fork-join das *threads* utilizadas pelo OpenMP.
Fonte: `https://commons.wikimedia.org/w/index.php?curid=32004077`

Apesar do ambiente construir esse paralelismo, trabalhar apenas com a CPU não chega ser computação heterogênea, para permitir que aplicações paralelas sejam executadas em GPUs é necessário utilizar a biblioteca **OpenCL** (*Open Computing Language*), podendo aproveitar todos os dispositivos presentes na máquina. Stringhini, Gonçalves e Goldman (2012) declara que uma aplicação OpenCL executada num hardware heterogêneo deve seguir os seguintes passos:

- Descobrir os componentes que compõem o sistema heterogêneo;
- Detectar as características do hardware heterogêneo tal que a aplicação possa se adaptar a elas;
- Criar os blocos de instruções (*kernels*) que irão executar na plataforma heterogênea;
- Iniciar e manipular objetos de memória;
- Executar os *kernels* na ordem correta e nos dispositivos adequados presentes no sistema;
- Coletar os resultados finais.

A plataforma é composta por um host e um ou mais dispositivos com capacidade de processamento. O host é conectado nesses dispositivos e é responsável pela inicialização e envio dos *kernels* para a execução nos dispositivos heterogêneos.

3.2 CUDA

CUDA (*Compute Unified Device Architecture*), é uma tecnologia criada pela NVIDIA conhecida como Arquitetura de Computação Paralela de Propósito Geral, que foi disponibilizando um novo modelo de programação paralela e um conjunto de instruções, causando uma melhora de eficiência no processamento de algumas aplicações paralelas executadas em GPUs ao invés de utilizar a própria CPU (STRINGHINI; GONÇALVES; GOLDMAN, 2012). Os desenvolvedores que optarem por esse ambiente podem utilizarem C/C++ como linguagem de desenvolvimento, sendo também fornecidas bibliotecas de aplicações pertencentes ao seu kit de desenvolvimento (SDK).

No modelo de programação CUDA, as GPU's ficarão responsáveis de executar as *threads*, trabalhando como coprocessador do host, da forma que a CPU fica responsável pelo código principal, e faz chamadas a funções que são executadas pela GPU, como mostra a figura 2. As *threads* executam um mesmo código definido em uma função kernel, quando é feita uma chamada dessa função, são executadas N instâncias paralelas por N *threads* CUDA.

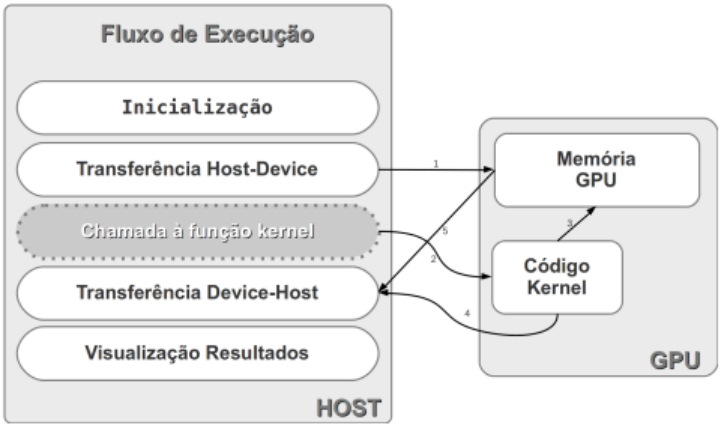

Figura 2 – Fluxo de execução do CUDA.
Fonte: (STRINGHINI; GONÇALVES; GOLDMAN, 2012)

4 CONCLUSÃO

Como visto, sistemas heterogêneos estão sendo cada vez mais utilizados em ambientes computacionais, combinando o poder de processamento de CPUs, FPGAs e GPUs. Em razão disso, diversas arquiteturas diferentes precisaram ser construídas e aperfeiçoadas para suprir as necessidades de computadores cada vez mais eficientes.

O uso de núcleos heterogêneos e aceleradores se mostraram uma ótima alternativa aos sistemas de apenas um núcleo de processamento, onde, para ter melhor performance, era necessário o aumentar a densidade de transistores por chip. Esse aumento implica em maior consumo de energia de forma que, a partir de certo ponto, a relação custo-benefício se torna inviável. A estratégia do uso de paralelismo se tornou muito interessante, já que a carga de trabalho pode ser dividida entre dezenas ou até centenas de núcleos de processamento. Estes núcleos, por sua vez, são menos complexos do ponto de vista de hardware e do conjunto de instruções, mas estão presentes em maior quantidade e permitem execução de forma paralela.

As arquiteturas cujo funcionamento foi aprofundado neste trabalho, foram as de Fermi, que dedica uma grande quantidade de transistores para as unidades de execução, e a de Kepler, que melhorou significativamente a performance ao executar novas *threads* de maneira dinâmica. É importante destacar o uso de GPUs como aceleradores e sua relevância para o mercado, já que elas melhoraram bastante o desempenho de computadores e possuem uma série de opções de ferramentas e ambiente de programação, como é o caso do CUDA e OpenCL.

Esta área da computação heterogênea está em constante evolução, sendo que arquiteturas e ambientes de programação mais eficientes poderão surgir em breve. Modelos de paralelismo do tipo SIMT ou SIMD estão sendo amplamente utilizados e estão com boas perspectivas para o futuro.

REFERÊNCIAS

BARNEY, B. et al. Introduction to parallel computing. **Lawrence Livermore National Laboratory**, v. 6, n. 13, p. 10, 2010.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. **Communications of the ACM**, p. 67–77, 2011.

BRODTKORB, A. et al. State-of-the-art in Heterogeneous Computing. **Scientific Programming**, v. 18, p. 1–33, 10/2010. DOI: 10.1155/2010/540159.

DONGARRA, J.; LASTOVETSKY, A. L. **High performance heterogeneous computing**. John Wiley & Sons, 2009. v. 78.

FERMI Compute Architecture White Paper. 2009. Disponível em: <https://www.nvidia.com.br/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Acesso em: 31/08/2021.

GASTER, B. et al. **Heterogeneous computing with OpenCL: revised OpenCL 1.2 edition**. 2. ed.: Morgan Kaufmann, 2012.

MEMETI, S. et al. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In: PROCEEDINGS of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing. 2017. P. 1–6.

STRINGHINI, D.; GONÇALVES, R. A.; GOLDMAN, A. Introdução à computação heterogênea. In: ANAIS da XXXI Jornada de Atualização em Informática do XXXII Congresso da Sociedade Brasileira de Computação [Internet]. 2012. P. 16–19.