Machine Learning e IA em Ambientes Distribuídos Versão 2.0

APIs de Programação Paralela

Atualmente a computação de alto desempenho dispõe de uma série de APIs para o desenvolvimento de aplicações paralelas. Destacam-se OpenMP para o processamento em CPU, CUDA, OpenCL e OpenACC para processamento em GPU. Vejamos as APIs em detalhes.

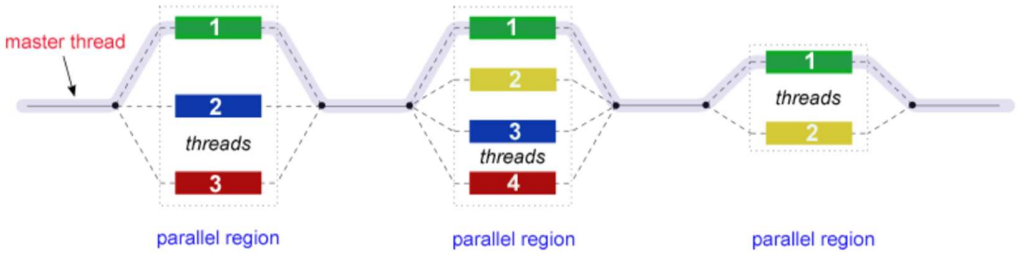
1. OpenMP

OpenMP é uma especificação que fornece um modelo de programação paralela com compartilhamento de memória. Essa API é composta por um conjunto de diretivas que são adicionadas as linguagens C/C++ e Fortran utilizando o conceito de *threads*, porém sem que o programador tenha que trabalhar diretamente com elas. Esse conjunto de diretivas quando acionado e adequadamente configurado cria blocos de paralelização e distribui o processamento entre os núcleos disponíveis. O programador não necessita se preocupar em criar *threads* e dividir as tarefas manualmente no código fonte. O OpenMP se encarrega de fazer isso em alto nível.

O OpenMP não é uma linguagem de programação. Ele representa um padrão que define como os compiladores devem gerar códigos paralelos através da incorporação nos programas sequenciais de diretivas que indicam como o trabalho será dividido entre os *cores*. Dessa forma, muitas aplicações podem tirar proveito desse padrão com pequenas modificações no código. A palavra *Open* presente no nome da API significa que é padrão aberto e está definido por uma especificação de domínio público e MP são as siglas de *Multi Processing*.

No OpenMP, a paralelização é realizada com múltiplas threads dentro de um mesmo processo. As threads são responsáveis por dividir o processo em duas ou mais tarefas que poderão ser executadas simultaneamente. Diferente dos processos em que cada um possui seu próprio espaço de memória, cada thread compartilha o mesmo endereço de memória com as outras threads do mesmo processo, porém cada thread tem a sua própria pilha de execução. O modelo de programação do OpenMP é conhecido por fork-join, onde um programa inicia com uma única thread que executa sozinha todas as instruções até encontrar uma região

paralela que é identificada por uma diretiva OpenMP. Ao chegar nessa região, um grupo de threads é alocado e juntas executam o código paralelizado. Ao finalizar a execução do paralelismo as threads são sincronizadas e a partir desse ponto somente uma thread (inicial) é que segue com a execução do código sequencial. O fork-join pode ocorrer diversas vezes e é dependente do número de regiões paralelas que o programa possui. Esse modelo é ilustrado pela figura abaixo, onde cada uma das três regiões paralelas tem 3, 4 e 2 threads, respectivamente.

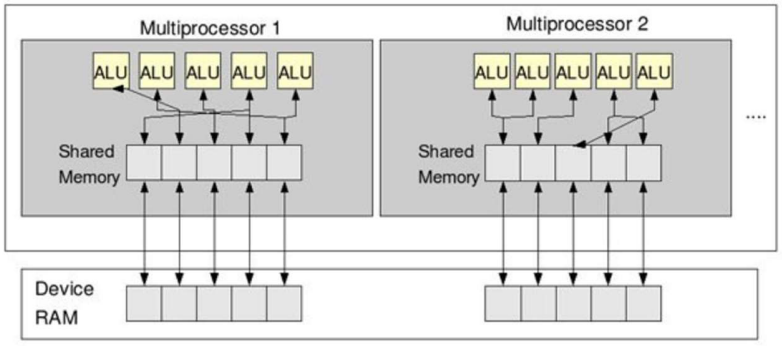


A API OpenMP prove uma série de vantagens na sua utilização. Através da inclusão de diretivas, são mínimas as alterações no código fonte sequencial. Além do mais, possibilita o ajuste dinâmico do número de *threads* com suporte a paralelismo aninhado, apresentando facilidade na compreensão e utilização. As funcionalidades do OpenMP são constituídas através das variáveis de ambiente (*OMP_NOME*), diretivas de compilação *(#pragma omp diretiva [cláusula]*) e bibliotecas de serviço (*omp_serviço*).

2. CUDA

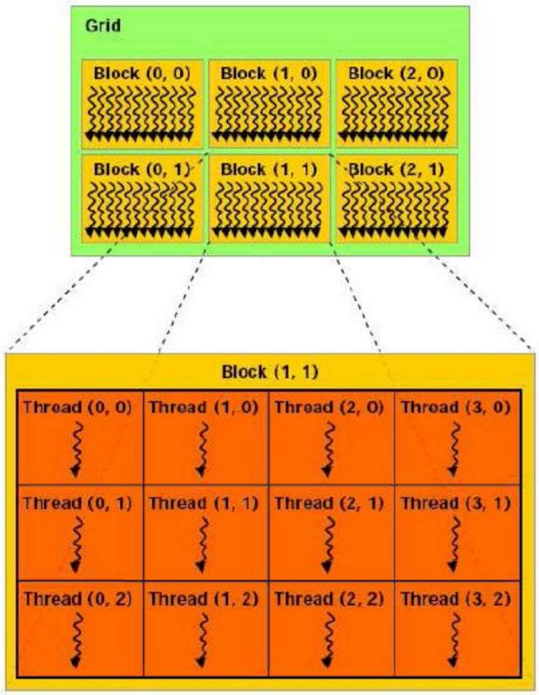
CUDA é uma plataforma de computação paralela e um modelo de programação criados pela NVIDIA em 2006. Seu objetivo é possibilitar ganhos significativos de desempenho computacional aproveitando os recursos das unidades de processamento gráfico (GPU). Através da API CUDA pode-se enviar código C, C++ e Fortran diretamente à GPU (também podemos usar Python com PyCUDA), sem necessitar de uma nova linguagem de compilação. A tecnologia CUDA é de abordagem proprietária, concebida para permitir acesso direto ao *hardware* gráfico específico da NVIDIA.

Ao utilizar CUDA também é possível gerar código tanto para a CPU como para a GPU. CUDA oferece um conjunto de bibliotecas e o compilador NVCC, onde é possível explicitar dentro do código fonte as instruções que devem ser executadas na CPU, na GPU ou em ambas. Para isso tornou-se necessário adicionar algumas extensões à linguagem C padrão. Em CUDA a GPU é vista como um dispositivo de computação adequado para aplicações paralelas. Tem seu próprio dispositivo de memória de acesso aleatório e pode ser executada através de um grande número de *threads* em paralelo. Um aspecto importante da programação CUDA é a gestão de acesso à memória, conforme demonstra a figura abaixo.



Na arquitetura CUDA a GPU é implementada como um conjunto de multiprocessadores. Cada um dos multiprocessadores tem várias *Arithmetic Logic Unit* (ALU) que em qualquer ciclo de *clock* executam as mesmas instruções, mas em dados diferentes. As ALUs podem acessar através de leitura e escrita a memória compartilhada do multiprocessador e a memória RAM (*Random Access Memory*) do dispositivo.

As GPUs atuais suportam até 1024 *threads* em cada bloco. Cada bloco é escalonado em um dos multiprocessadores das GPUs. Um multiprocessador cria, gerencia e executa de modo concorrente todas as *threads* de um bloco, com custo (*overhead*) zero de escalonamento. Quando um multiprocessador encerra o bloco de processamento atual, novos blocos são escalonados para eles. A figura abaixo demonstra a estrutura de um *kernel*.



Um *kernel* nada mais é do que um núcleo composto por blocos de *threads*, onde o processamento no lado da GPU é executado. Os blocos podem ser compostos por uma ou por várias *threads*. A execução de instruções na GPU, na maioria dos casos, tende a apresentar um desempenho maior se comparada a execuções em CPU. Porém, instruções contendo muitos desvios condicionais podem apresentar um baixo desempenho quando executados em GPU. No lado do usuário, não há a necessidade de saber quantos multiprocessadores uma GPU possui, de forma a definir o número de blocos e *threads* por bloco. Essa tarefa é de responsabilidade do sistema que efetua o escalonamento dos blocos nos multiprocessadores da GPU. Para o usuário essa tarefa é completamente transparente.

Em CUDA, cada execução em GPU deve ser através de um *kernel*. CUDA, por limitar a execução em dispositivos somente fabricados pela NVIDIA, traz consigo um conjunto de instruções que não são compatíveis com as demais APIs. Ao se migrar uma aplicação para CUDA tem-se que codificar quase que na totalidade o algoritmo. Diferente disso, o OpenACC vem

para ser um novo padrão para programação paralela para diferentes dispositivos, através da introdução de diretivas muito semelhantes ao OpenMP e com pequenas alterações no código fonte sequencial.

3. OpenACC

Desenvolvida por um grupo de empresas incluindo principalmente NVIDIA, Portland Group Inc, CAPS Enterprise e CRAY, o OpenACC define uma especificação para execução de programas desenvolvidos em C, C++ e Fortran a partir de uma CPU para um dispositivo acelerador. Seus métodos provêm um modelo de programação para realizar a aceleração de instruções para diferentes tipos de dispositivos *multi-core* e *many-core*.

Através de um conjunto de diretivas, o OpenACC analisa a estrutura e os dados do programa e quais partes foram divididas entre o *host* (CPU) e o dispositivo acelerador. A partir dessa etapa, um mapeamento otimizado é gerado para ser executado em *cores* paralelos. Além da aceleração, que é o principal objetivo dessa API, o OpenACC fornece uma forma de migrar aplicativos através de pequenas mudanças na forma sequencial do algoritmo.

O modelo de execução alvo do OpenACC são *hosts* (CPUs) em conjunto com dispositivos aceleradores, como é o caso da GPU. A responsabilidade do *host* é receber a carga do aplicativo e direcionar as ações para o dispositivo acelerador. Essas ações são compostas geralmente por regiões que contém instruções de repetição ou *kernels*. O dispositivo acelerador apenas executa as instruções que lhe foram repassadas pelo *host*. O host deve ainda gerenciar a alocação de memória no dispositivo acelerador, transferir os dados do e para o *host*, enviar as instruções de execução e aguardar o término da execução.

4. OpenCL

OpenCL (Open Computing Language) é uma arquitetura para escrever programas que

funcionam em plataformas heterogêneas, consistindo em CPUs, GPUs e outros processadores. OpenCL inclui uma linguagem para escrever kernels (funções executadas em dispositivos OpenCL), além de APIs que são usadas para definir e depois controlar as plataformas heterogênea. OpenCL permite programação paralela usando, tanto o paralelismo de tarefas, como de dados. OpenCL é o padrão aberto para programação paralela desenvolvida pelo consórcio Khronos Group em 2008. Esse padrão permite a você desenvolver aplicações que podem ser executados em paralelo em GPUs ou em CPUs com diferentes arquiteturas em um sistema heterogêneo.

Em outras palavras, o OpenCL torna possível o uso de todos os núcleos do CPU ou a enorme capacidade de computação de GPUs ao calcular uma tarefa, reduzindo assim o tempo de execução do programa. A utilização do OpenCL é, portanto, muito benéfica para lidar com tarefas associadas com computações trabalhosas e consumidoras de recursos.

Ela foi adotada para controladores de placas gráficas pela AMD/ATI, que a tornou na sua única oferta de GPU como Stream SDK, e pela Nvidia, que oferece também OpenCL como a escolha para o seu Compute Unified Device Architecture (CUDA) nos seus controladores. A arquitetura OpenCL partilha uma série de interfaces computacionais, tanto com CUDA, como com a concorrente DirectCompute da Microsoft.

A proposta OpenCL é similar às propostas OpenGL e OpenAL, que são padrões abertos da indústria para gráficos 3D e áudio, respectivamente. OpenCL estende o poder da GPU além do uso gráfico (GPGPU). OpenCL é gerido pelo consórcio tecnológico Khronos Group.

Referências:

Introduction to Parallel Computing
https://computing.llnl.gov/tutorials/parallel_comp/

OpenMP
https://computing.llnl.gov/tutorials/openMP/

Algorithms and Parallel Computing
https://www.amazon.com.br/Algorithms-Parallel-Computing-Wiley-Distributed-ebook/dp/B005CDYQNM/ref=sr_1_1?ie=UTF8&qid=1496297062&sr=8-1&keywords=Algorithms+and+Parallel+Computing

Introduction to Parallel Computing
https://www.amazon.com.br/Introduction-Parallel-Computing-Zbigniew-Czech-ebook/dp/B01MYVA2TU/ref=sr_1_1?ie=UTF8&qid=1496297105&sr=8-1&keywords=Introduction+to+Parallel+Computing

Programming on Parallel Machines
http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf

CUDA Toolkit Documentation
http://docs.nvidia.com/cuda/index.html#axzz4ijDhFtfQ

OpenCL Specification
https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf