Paralelismo de dados

(execução de simultaneidade)

Em métodos tradicionais de programação (processamento sequencial), **uma grande quantidade de dados é processada em um único núcleo de uma CPU**, **enquanto os outros núcleos permanecem livres.**

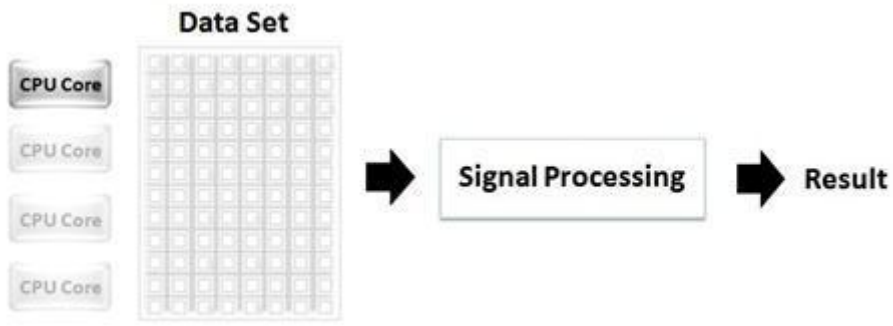


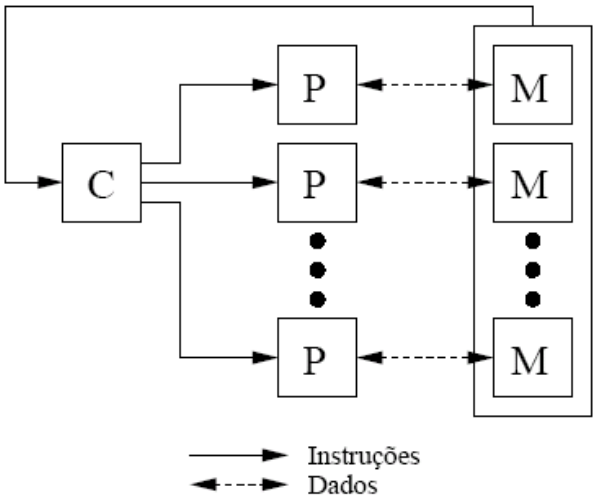
Figura 1 - Em métodos tradicionais de programação, o processamento sequencial de uma grande quantidade de dados é processada em um único núcleo da CPU, enquanto os outros núcleos permanecem livres.

Tipo de arquitetura paralela SIMD

- SIMD (**Single Instruction Multiple Data**)

- Significa que todas as unidades paralelas compartilham a mesma instrução, mas a realizam em diferentes elementos de dados.

Arquitetura SIMD



SIMD - Single Instruction Multiple Data

- A idéia é que podemos adicionar os arrays A=[1,2,3,4] e B=[5,6,7,8] para obter o array S=[6, 8, 10, 12] .
- Para isso, tem que haver quatro unidades aritméticas no trabalho, mas todos podem compartilhar a mesma instrução (aqui, "*add*").

Paralelismo de dados **refere-se a cenários de processamento em que a mesma operação (instrução) é executada simultaneamente (isto é, em paralelo) aos elementos em uma coleção de origem ou uma matriz.**

Em operações paralelas de dados, a coleção de origem é particionada para que vários threads podem funcionar simultaneamente em diferentes segmentos.

O **paralelismo de dados** é uma técnica de programação que divide uma grande quantidade de dados em partes menores que podem ser operadas em paralelo. Após os dados serem processados, eles são combinados novamente em um único conjunto.

Considere o cenário da Figura 1, que ilustra uma grande quantidade de dados sendo operada em um único processador. Neste cenário, os outros 3 núcleos da CPU disponíveis estão livres enquanto o primeiro processador opera os dados do conjunto inteiro.

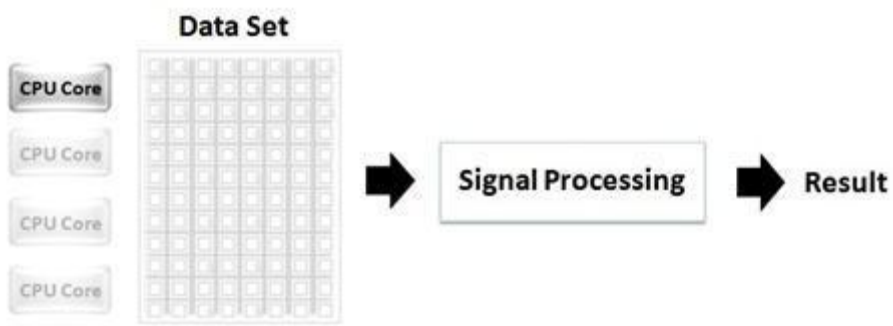


Figura 1. Em métodos tradicionais de programação, uma grande quantidade de dados é processada em um único núcleo da CPU, enquanto os outros núcleos permanecem livres.

Mas, pode-se aplicar paralelismo de dados para grandes conjuntos de dados como vetores e matrizes (arrays), dividindo esse conjunto em subgrupos, realizando operações e combinando seus resultados, como na Figura 2.

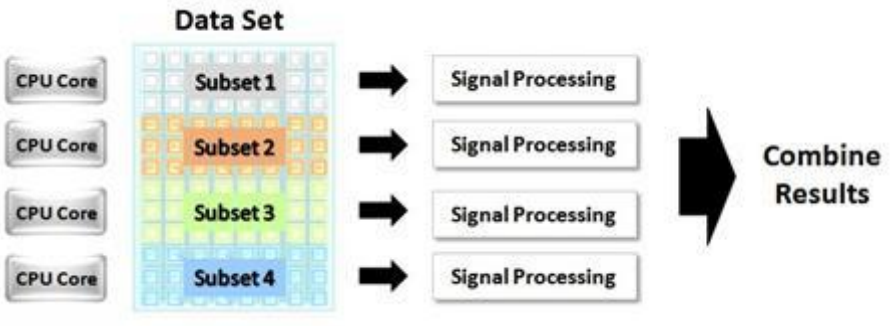


Figura 2 - Utilizando a técnica de programação com paralelismo de dados, uma grande quantidade de dados pode ser processada em paralelo em múltiplos núcleos da CPU/GPU.

Com esta técnica, programadores podem modificar um processo que tipicamente não seria capaz de utilizar as potencialidades dos processadores **multicore** (CPU) ou **manycore** (GPU), e assim, particionando os dados (paralelizando operaçoes sobre os mesmos), pode-se utilizar toda a força de processamento disponível.

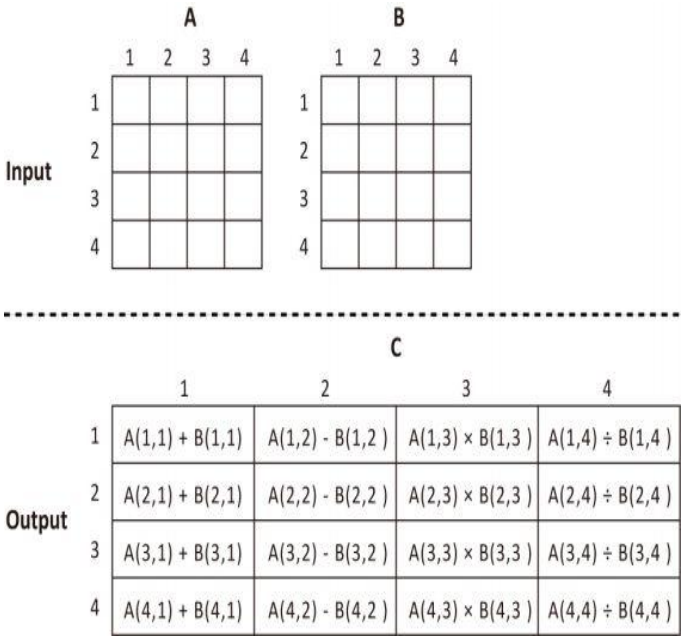
Visto de outra forma, em primeiro lugar, considere a aplicação seqüencial, na qual uma única CPU (um único núcleo) processa todo o conjunto de dados. Neste caso, tem-se um **único núcleo de processamento**.

Em vez disso, considere o exemplo de um mesmo conjunto de dados divididos em quatro partes. Você pode distribuir este conjunto de dados em todos os núcleos disponíveis para alcançar um aumento significativo de velocidade.

Ou múltiplos núcleos de processamento -

Em aplicações computacionais de alto desempenho (High-Performance Computing - HPC) em tempo real, como sistemas de controles, uma estratégia comum e eficiente é a realização paralela de multiplicações de matrizes de dimensões consideráveis. Normalmente a matriz é fixa, e é possível realizar sua decomposição. A medição colhida dos sensores fornece o vetor (ou a matriz) a cada iteração do loop. Você pode, por exemplo, usar os resultados da matriz para controlar atuadores.

Figure 4.4:
Basic arithmetic operations between floats



Na Figura 4.4

- Como mostra a Figura 4.4, os dados de entrada consistem em 2 conjuntos de matrizes 4x4, A e B. Os dados de saída são uma matriz 4x4, C.

- Veja primeiro a implementação paralela de dados (Lista 4.8, Lista 4.9).

- Este programa trata cada linha de dados como um *work-group* para executar a computação.

Podemos ver no link seguinte, um exemplo do modelo de paralelismo de dados.

Exemplo - *List-4-8-List-4.9* onde a mesma instrução atua sobre diferentes elementos de dados, tratando cada linha de dados como um work-group em OpenCL, ou o que é o mesmo como um bloco de threads em CUDA.

Na Lista 4.8, a linha de código abaixo define um kernel para o modelo de paralelismo de dados, como segue:

```
__kernel void dataParallel(__global float * A, __global float * B, __global float * C)
```

Quando o kernel para paralelismo de dados é enfileirado, work-items (threads) são criados. Cada destes work-items (threads) executa o mesmo kernel em paralelo.

```
int base = 4*get_global_id(0);
```

Na Lista 4.8, esta linha de código significa que o get_global_id(0) obtém o ID global do work-item (thread), o qual é usado para decidir qual dado é para processar, de modo que cada work-item (thread) pode processar diferentes conjuntos de dados em paralelo.

Em geral, o processamento paralelo de dados é feito usando as seguintes etapas.

1. Obter ID do work-item.

2. Processe o subconjunto de dados correspondentes ao ID dowork-item.

Um diagrama de blocos do processo é mostrado na Figura 4.5.

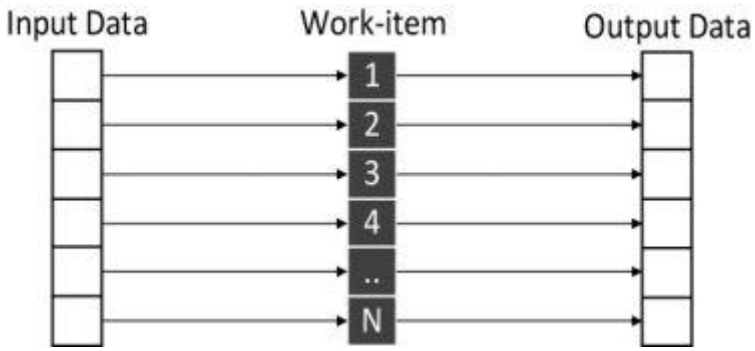


Figura 4.5 - Diagrama de blocos do modelo paralelo de dados em relação aos itens de trabalho.

Neste exemplo, o work-item global é multiplicado por 4 e armazenado na variável "base". Esse valor é usado para decidir qual elemento da matriz A e B é processado.

```
1. C[base+0] = A[base+0] + B[base+0];
2. C[base+1] = A[base+1] - B[base+1];
3. C[base+2] = A[base+2] * B[base+2];
4. C[base+3] = A[base+3] / B[base+3];
5.
```

Uma vez que cada work-item tem IDs diferentes, a variável "base" também possui um valor diferente para cada work-item (base+0, base+1, base+2, base+3), o qual mantém os work-items de processar os mesmos dados (processar os dados na mesma linha das matrizes). Desta forma, uma grande quantidade de dados pode ser processada simultaneamente.

Discutimos que vários work-items foram criados, mas não abordamos como decidir o número de work-items a serem criados. Isso é feito no seguinte segmento de código do código do host (processado na CPU).

```
1. size_t global_item_size = 4;
2. size_t local_item_size = 1;
3.
4. /* Execute OpenCL kernel as data parallel */
5. ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
6. &global_item_size, &local_item_size, 0, NULL, NULL);
```

O `clEnqueueNDRangeKernel()` é um comando OpenCL, usado para uma fila de **tarefas paralelas de dados**. Os 5º e 6º argumentos determinam o tamanho do work-item. Nesse caso, o `global_item_size` é definido como 4 e o `local_item_size` é definido como 1. Os passos gerais são resumidos da seguinte forma.

1. Criar work-items no host

2. Processa dados correspondentes ao ID do work-item global no kernel.

Paralelismo de Tarefas

(execução de simultaneidade)

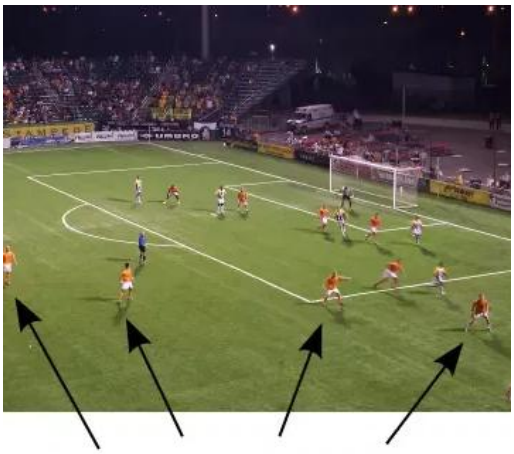
Publicado: julho de 2016 - https://msdn.microsoft.com/pt-br/library/dd492427.aspx

No que segue, passaremos o código-fonte (Lista 4.10 e Lista 4.11) para o modelo paralelo da tarefa. Neste modelo, diferentes kernels podem ser executados em paralelo. Observe que os kernels diferentes são implementados para cada uma das 4 operações aritméticas.

No tempo de execução de simultaneidade, um *tarefa* é uma unidade de trabalho que executa um trabalho específico e normalmente é executado em paralelo com outras tarefas.

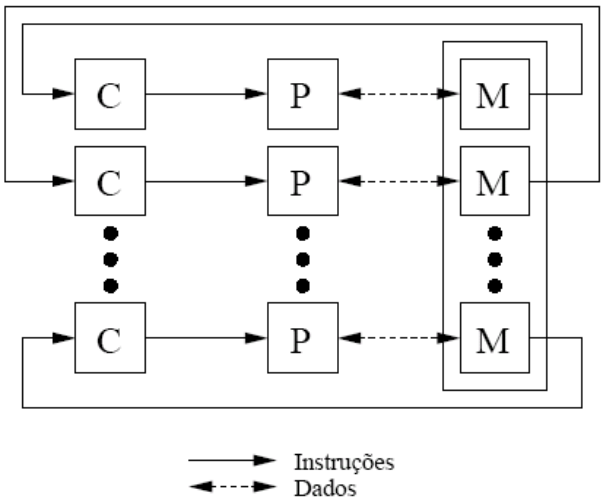
Tipo de arquitetura paralela MIMD

- MIMD (**Multiple Instruction Multiple Data**)
- Significa que as unidades paralelas têm instruções distintas, então cada uma delas pode fazer algo diferente em um dado momento.



Workers with same objective,
doing completely different things

Do ponto de vista da arquitetura visualizada em termos de hardware, você pode ver a figura que segue:



MIMD

Uma tarefa pode ser decomposta em tarefas mais refinadas adicionais que são organizadas em um ***grupo tarefas*** (podendo-se definir um **bloco** de tarefas).

Paralelismo de tarefa é a forma mais simples de programação paralela, onde as aplicações estão divididas em tarefas exclusivas que são independentes umas das outras e podem ser executadas em processadores diferentes.

Como exemplo, considere **um programa com dois loops (loop A e loop b)**; o **loop A executa uma rotina de processamento de sinais e o loop b realiza atualizações na interface de usuário**. Isto representa paralelismo de tarefas, em que uma aplicação multithread pode executar dois loops em threads separadas para utilizar dois núcleos de uma CPU.

Você usa tarefas quando você quer escrever código assíncrono e deseja alguma operação após a operação assíncrona ser concluída.

Num código assíncrono, tarefas do seu programa poderão ser executadas sem interferir em nada do fluxo de execução dos códigos de outras tarefas. Essas tarefas são conhecidas como fluxos assíncronos.

Você pode disparar uma série dessas tarefas sem precisar esperar que cada uma se complete para prosseguir.

Por exemplo, você poderia usar uma tarefa para ler de um arquivo de forma assíncrona e, em seguida, usar outra tarefa — um ***tarefa de continuação***, que é para processar os dados depois que ele fica disponível.

Por outro lado, você pode usar grupos de tarefas para decompor o trabalho paralelo em partes menores.

Por exemplo, suponha que você tenha um algoritmo recursivo que divide o trabalho restante em duas partições. Você pode usar grupos de tarefas para executar essas partições simultaneamente e aguarde até que o trabalho dividido para ser concluído.

Você pode ver, no link seguinte, um exemplo de código usando o modelo de paralelismo de tarefas.

Exemplo *List-4-10-List-4-11* onde as tarefas tem instruções (operações) diferentes, e cada uma delas faz algo diferente em um dado momento, de forma paralelizada.

```
1. /* Execute OpenCL kernel as task parallel */
2. for (i=0; i < 4; i++) {
3. ret = clEnqueueTask(command_queue, kernel[i], 0, NULL, NULL);
4. }
```

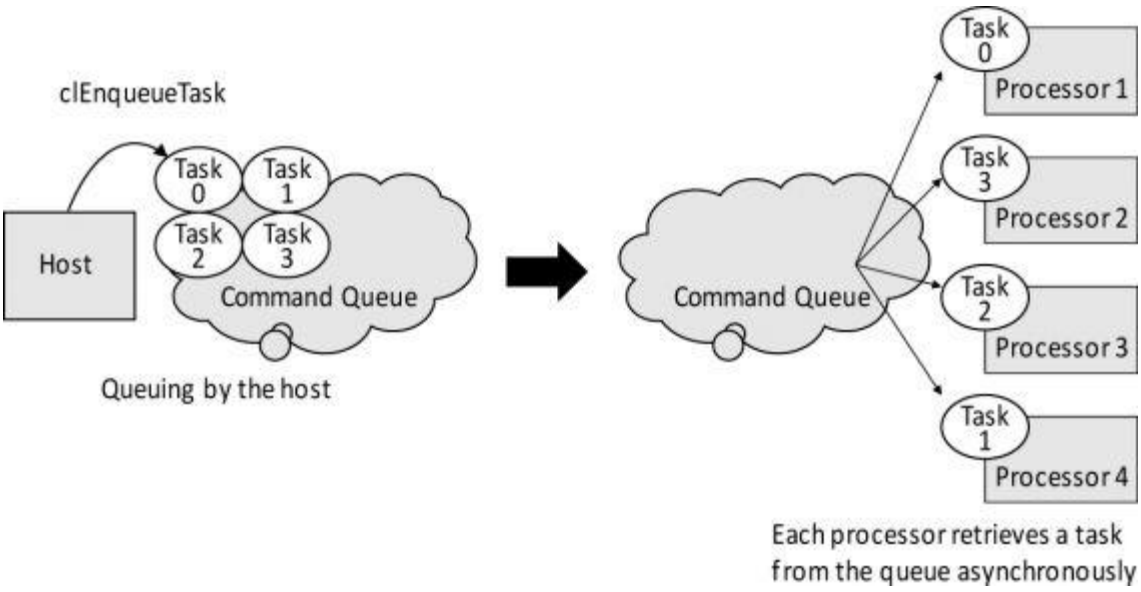
O segmento de código acima enfileira os 4 kernels correspondentes as quatro operações envolvidas entre as matrizes A e B.

No OpenCL, **para executar o modelo paralelo de tarefas**, o **modo fora de ordem** (out-of-order) deve ser ativado quando a fila do comando é criada. Usando este modo, a tarefa em fila não aguarda até que a tarefa anterior seja concluída se houver unidades de computação ociosas disponíveis que possam estar executando essa tarefa.

```
1. /* Create command queue */
2.  command_queue = clCreateCommandQueue(context, device_id,
   CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &ret);
```

O diagrama de blocos ilustrando as filas de comando e execução paralela é mostrado na Figura 4.6, seguinte.

Figura 4.6 - Filas de comando e execução paralela (lado CPU e lado GPU).



O `clEnqueueTask()` é usado como um exemplo na Figura 4.6 acima, mas um processamento paralelo semelhante pode ocorrer para outras combinações de funções de *enqueue* (enfileiramento), como `clEnqueueNDRangeKernel()` , `clEnqueueReadBuffer()` e `clEnqueueWriteBuffer()`.

Por exemplo, uma vez que o *PCI Express* (*o padrão de comunicação criado em 2004 pelo grupo Intel, Dell, HP e IBM para fazer a comunicação entre placas de expansão e a placa mãe, utilizadas em computadores pessoais para transmissão de dados*), suporta transferências simultâneas de memória bidirecional, enfileirar `clEnqueueReadBuffer()` e `clEnqueueWriteBuffer()` podem executar comandos de leitura e gravação simultaneamente, desde que os comandos sejam executados por diferentes processadores.

No diagrama acima, podemos esperar que **as 4 tarefas sejam executadas em paralelo**, uma vez que estão sendo enfileiradas em uma fila de comando que está habilitada fora de execução.

• Conclusão -

Em OpenCL, O código paralelizável é implementado como:

"**Paralelismo de dados**" ou "**Paralelismo de Tarefa**"

- No OpenCL, **a diferença entre os dois modelos de paralelismo** é, se o **mesmo kernel** (dados, região paralela em OpenMP) ou **kernels diferentes** (tarefas, seções diferentes em OpenMP) são executados em paralelo.

Modelo Paralelo de Dados x Modelo Paralelo de Tarefas

- Pode parecer que, uma vez o modelo paralelo de tarefas requer mais código, ele também deve executar mais operações.
- No entanto, independentemente do modelo que seja utilizado para este problema, o número de operações efetuadas pelo dispositivo (GPU) é realmente o mesmo.
- Apesar desse fato, o desempenho pode variar, escolhendo um ou outro modelo, de modo que o modelo de paralelização deve ser sempre considerado com sabedoria na fase de planejamento do aplicativo.

==========