Introdução ao CUDA

Material elaborado por Davi Conte.

O objetivo deste material é que o aluno possa iniciar seus conhecimentos em programação paralela, entendendo a diferença da execução de forma sequencial e de forma concorrente, assim como os ganhos de desempenho refletidos desses modelos. Para isso será apresentada a tecnologia por trás do modelo de programação em CUDA. Serão discutidos os conceitos básicos da arquitetura de uma placa de vídeo, o funcionamento da GPU e computação em GPU, conhecimentos necessários para realizar a compreensão de CUDA.

1. A Placa de Vídeo

Antes de começarmos a falar sobre CUDA em si, é necessário entendermos o que é placa de vídeo e GPU (Graphics Processing Unit). Primeiramente, para ter uma breve introdução do que é a placa de vídeo, assista o vídeo referenciado no link abaixo.

Como funcionam as placas de vídeo?

→ https://www.youtube.com/watch?v=Cqht_EfXDLk&t=193s

Continuando, o principal elemento de uma placa de vídeo que vamos estudar é a Graphics Processing Unit, também conhecida como GPU. Este componente se comporta como um processador dedicado especialmente para a renderização de gráficos.

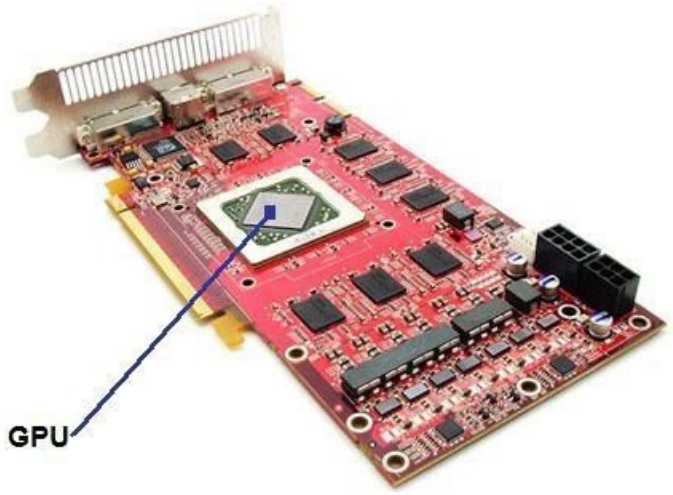

Figura 1. GPU exposta em uma placa de vídeo [1]

A GPU fica encarregada pelo processamento de cada pixel da tela, gerando, assim, a imagem proveniente de de um alto número de operações. Por isso, a GPU possui diversos núcleos que são como pequenos processadores (diferentes de processadores de

propósito geral como os desenvolvidos pela Intel para Notebooks ou Desktops). A Figura 2 ilustra a diferença dos múltiplos cores (núcleos) de uma CPU e os milhares de uma GPU.

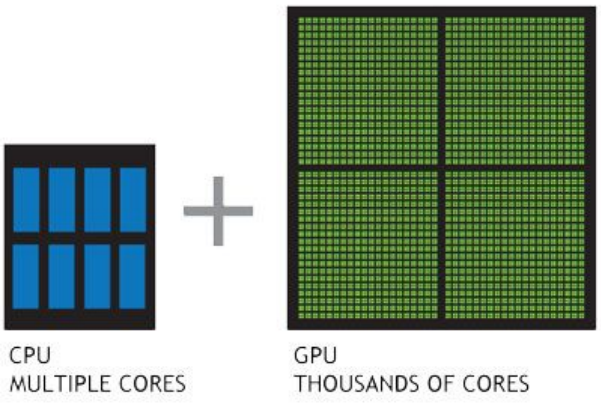

Figura 2. As placas de vídeo têm milhares de núcleos para processar cargas em paralelo [2]

2. Programação Paralela

Muitas vezes quando temos uma tarefa muito complexa a ser feita, podemos dividi-la em partes menores que possam ser mais simples de serem resolvidas. A programação paralela ou concorrente aplica esse conceito aos processadores em um computador. Um processador executa uma única instrução de código por vez, portanto um computador com diversos núcleos pode executar diversas instruções ou operações simultaneamente, resolvendo problemas de menor complexidade paralelamente, ou seja, em um mesmo instante de tempo.

Tradicionalmente, os softwares são construídos em computação serial ou sequencial, onde o problema é dividido em uma série de instruções que são executadas sequencialmente, uma após a outra, em um único processador, onde somente uma instrução pode ser executada em um dado intervalo de tempo. Na computação paralela, existe o uso simultâneo de múltiplos recursos computacionais no intuito de resolver determinado problema. Desta forma, os problemas são divididos em partes menores que podem ser resolvidas concorrentemente, sendo que cada parte é subdividida em uma série de instruções, executadas simultaneamente em diferentes processadores, coordenadas por um mecanismo de controle global [3].

A programação concorrente trata-se de um paradigma que permite a implementação de software com múltiplas tarefas independentes, ainda que em máquinas com um único núcleo de processamento, onde estas atividades disputam recursos computacionais,no intuito de solucionar um problema específico.

O modelo de programação paralela permite ganhos significativos em critérios de tempo de execução, quando comparados com soluções implementadas por meio de programação serial que são refletidos no tempo final de execução.

A Figura 3 mostra a partição de um problema, que pode ser a implementação de um algoritmo, em quatro tarefas cujas instruções são executadas paralelamente por diferentes CPUs.

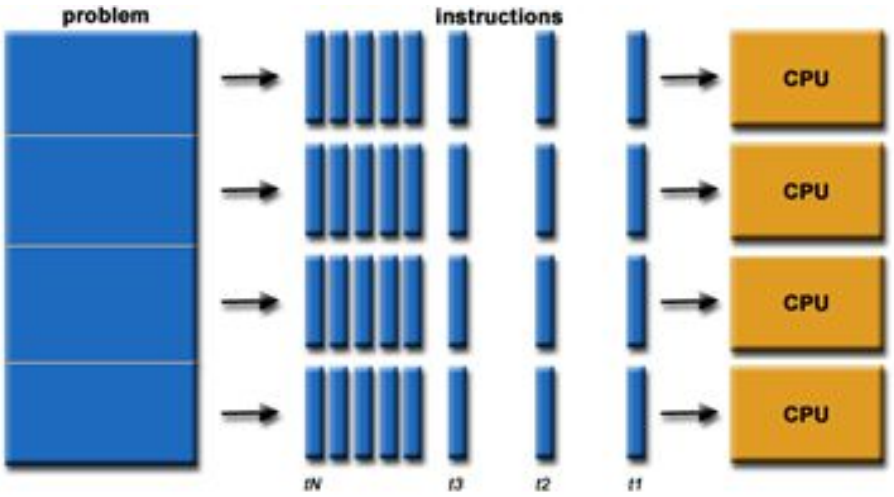

Figura 3. Divisão de tarefas de um problema entre as CPUs.

Considerando programação concorrente, a GPU, por conter diversos processadores, se torna um ambiente onde pode ocorrer grande paralelização de processamento de dados, o que nos leva à computação que pode ser realizada de forma paralela dentro dos núcleos de uma GPU, apresentando melhora na performance de desempenho. A Figura 4 demonstra que parte do código de uma aplicação pode ser executada de forma paralela em uma GPU, como por exemplo um laço de repetição com iterações definidas. Embora a porcentagem de código executada em GPU seja pequena, o aumento de performance é significativo comparado ao desempenho do código executado inteiramente de forma sequencial.

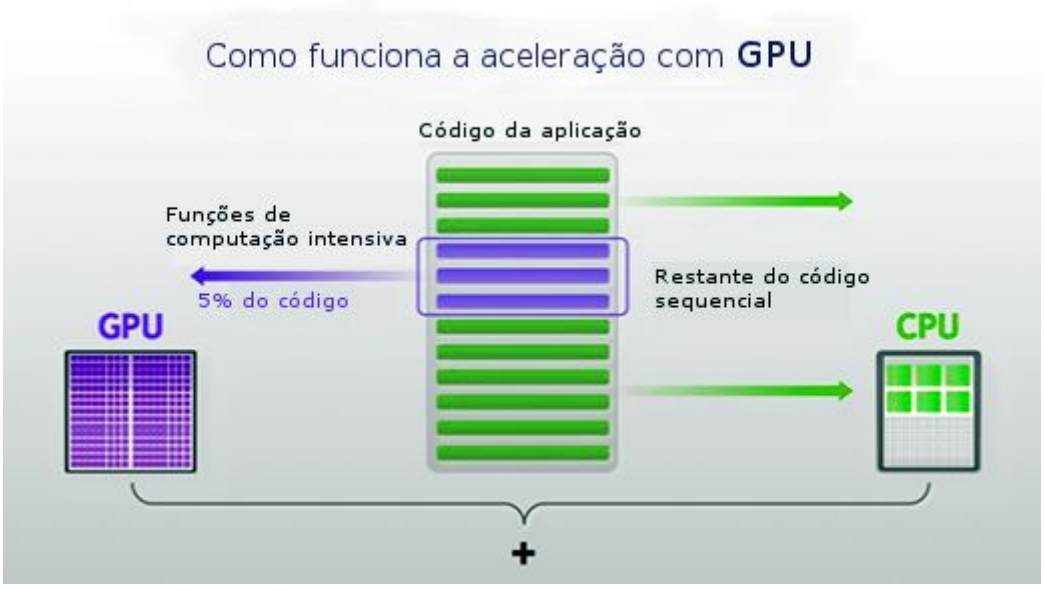

Figura 4. Aceleração de código utilizando GPU [2]

Para maior compreensão desse assunto, temos o próprio site da NVIDIA (produtora de placas de vídeo e proprietária do CUDA) explicando sobre computação em aceleradores gráficos (GPU), seguindo para o link abaixo.

→http://www.nvidia.com.br/object/what-is-gpu-computing-br.html

3. CUDA

CUDA (Compute Unified Device Architecture) é uma plataforma de computação paralela e um modelo de programação criados pela NVIDIA. Ela permite aumentos significativos de performance computacional ao aproveitar a potência da unidade de processamento gráfico (GPU) para o processar dados.

O modelo de programação CUDA utiliza as linguagens C, C++ ou Fortran, sendo necessário possuir uma placa de vídeo da NVIDIA para poder utilizar a linguagem. Recomenda-se a utilização de CUDA quando uma parte menor do processamento pode ser replicada e executada paralelamente em diversos núcleos. Isso porque os núcleos da GPU são menos poderosos, porém mais numerosos.

Para maior detalhamento sobre CUDA, recomenda-se a leitura do seguinte material proposto pela própria NVIDIA, desenvolvedora do CUDA.

→ http://www.nvidia.com.br/object/cuda_home_new_br.html

O gráfico da Figura 5, a seguir, demonstra a performance da execução de um algoritmo de multiplicação de matrizes sequencial (em uma CPU) e em paralelo (GPU) implementado com CUDA. É possível perceber a grande diferença de tempo de execução principalmente nas matrizes de maior tamanho (no gráfico: matrix size). Enquanto no sequencial a diferença da ordem (tamanho) da matriz influencia fortemente no tempo de execução, na execução paralela esse crescimento do tempo não é tão grande. O tempo no gráfico é medido em milissegundos (ms).

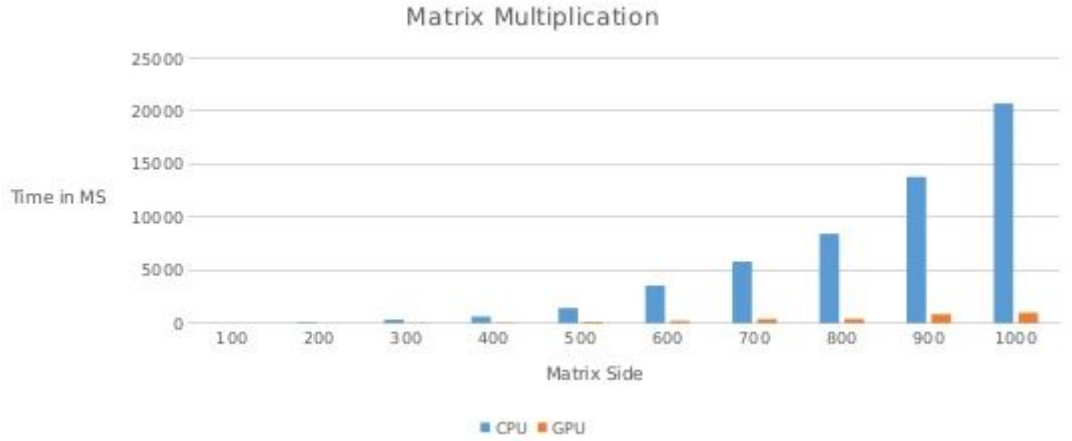

Figura 5. Multiplicação de matrizes paralelo e sequencial. [5]

4. Programação em CUDA

O objetivo deste capítulo é introduzir ao aluno os códigos em CUDA, apresentando seu compilador, funções específicas e exemplos de código. Este capítulo foi escrito baseado no capítulo 3 do livro “Cuda by example” escrito por Jason Sanders e Edward Kandrot [4], abordando diretamente os seguintes temas:

- Chamada de Kernel (__global__)
- Device and host
- Exemplo HELLO, WORLD em C + CUDA
- Kernel<<<1,1>>>();
- Passagem de parâmetros
- Funções básicas e utilização (cudaMalloc(), cudaMemcpy(), cudaFree())
- Exemplo Soma de vetores

4.1. Um primeiro programa

De acordo com as leis da computação, vamos iniciar a programação em CUDA com um bom código de Hello World! Uma observação importante é que o compilador de CUDA interpreta as linguagens C, C++ e Fortran naturalmente, ou seja, podemos compilar um programa de Hello World somente utilizando a linguagem C utilizando compilador CUDA. Porém, se for compilado e executado um código em C puro, mesmo com o compilador de CUDA, ele não será executado na GPU, cuja terminologia será usada como ***device***, mas será executado na CPU, cuja terminologia será usada como ***host***, pois para executar em GPU são necessários comandos específicos de CUDA. Um programa em CUDA tem a extensão .cu e seu compilador é o *nvcc*. Para compilar um código desenvolvido em CUDA, basta utilizar o seguinte comando para um programa .cu:

```
$ nvcc nomedoprograma.cu -o nomedoprograma
```

Para um código ser executado pelo *device* (GPU), é necessário fazer parte de uma função que tipicamente é chamada de ***kernel*.**

4.1.1. A Chamada de Kernel

Uma **chamada de *Kernel*** nada mais é do que a chamada da função que será executada pela GPU de forma paralela. Essa função precisa ser marcada para que o compilador (nvcc) saiba que a função é um código para a GPU. Observe nas imagens abaixo a diferença entre um programa “Hello, World” tradicional em C e outro “Hello, World” com uma chamada de Kernel (execução na GPU).

```c
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

Figura 5. Hello World tradicional em C.

```c
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

__global__ void kernel(void){
    printf("Hello, World from GPU!");
}

int main(void) {
    kernel<<<1,1>>>();
    printf("Fim");
    return 0;
}
```

Figura 6. Hello World com função *Kernel* executada na GPU.

O primeiro código, responde com “Hello, World” da CPU, enquanto que o segundo responde da GPU. Estes dois códigos possuem duas diferenças notáveis:

- Uma função chamada Kernel qualificada com *__global__; e*
- A chamada para esta função utilizando <<<1,1>>>.

O mecanismo da marcação __global__ em uma função alerta o compilador que a função deve ser compilada para executar no *device* (GPU) ao invés do *host* (CPU). No exemplo acima, o compilador (nvcc) envia a função *kernel()* para outro compilador que

gerencia sua execução nos núcleos da GPU enquanto que a função *main()* é executada na CPU. A utilização de <<<1,1>>> para chamadas de *kernel* será explicada posteriormente.

4.1.2. Passagem de Parâmetros

A função *kernel*, por mais que seja executada na GPU, também pode conter parâmetros em sua chamada, que falaremos a seguir. Considere o código em CUDA a seguir que será tomado como exemplo.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

__global__ void add( int a, int b, int *c ) {
        *c = a + b;
}

int main( void ) {
    int c;
    int *dev_c;

    cudaMalloc( (void**)&dev_c, sizeof(int));

    add<<<1,1>>>( 2, 7, dev_c );

    cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);

    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );
    return 0;
}
```

Figura 7. Adição em CUDA.

Você poderá perceber diversas alterações, novas linhas e comandos nesse trecho de código, porém essas mudanças demonstram dois conceitos:

- Podemos passar parâmetros para a função *kernel* como qualquer outra função; e
- Precisamos alocar memória para realizar qualquer atividade no *device*, como retornar valores ao *host*.

Podemos concluir que uma chamada de *kernel* funciona da mesma maneira que uma função comum no padrão C. Podemos passar os parâmetros da mesma forma tradicional e a compilação realiza a parte complexa da passagem dos parâmetros do *host* para *device*.

4.1.3. Alocando memória com *cudaMalloc()*

Um detalhe interessante da implementação da Figura 7 é a alocação de memória usando ***cudaMalloc().*** Esta chamada é semelhante ao ***malloc()*** do padrão em C, porém indica ao compilador de CUDA a alocação de memória no *device* (GPU). Este comando é

necessário para alocar espaço na memória para variáveis que serão utilizadas na chamada de *kernel* (processamento da GPU). Os argumentos desta chamada são os seguintes:
1- um ponteiro para o ponteiro que deseja guardar o endereço do novo espaço de memória alocado.
2- Tamanho da alocação que deseja realizar.

Exemplo dos parâmetros de ***cudaMalloc()***.

```
int *pt_a; //ponteiro para endereço de alocação
cudaMalloc(&pt_a, sizeof(int));
```

O ponteiro de endereço de memória que é passado em ***cudaMalloc()*** é utilizado, basicamente, para os seguintes fins em um código CUDA:

1. O ponteiro pode ser passado para outras funções que executam no *device*.
2. O ponteiro pode ser utilizado para ler ou escrever na memória da GPU a partir de código que executa no *device*.
3. Entretanto, o ponteiro não pode ser usado para ler ou escrever na memória da GPU a partir de código executado no *host.*

4.1.4. Liberando espaço com *cudaFree()*

Se você observou o código com atenção, percebeu que, da mesma forma que é utilizado ***free()*** para liberar espaço alocado em memória com ***malloc()*** no padrão C, também existe o ***cudaFree()*** para liberar espaço alocado da memória pelo ***cudaMalloc()***.

```
int *pt_a; //ponteiro para endereço de alocação
cudaMalloc(&pt_a, sizeof(int));
cudaFree(pt_a); //liberando espaço alocado
```

4.1.5. Acessando dados da memória do dispositivo com *cudaMemcpy()*

Até aqui aprendemos que é possível alocar e liberar espaço de memória no *device* (GPU), porém vimos que não é possível alterar este endereço a partir de código no *host*. Devido a isso, a seguir veremos duas maneiras mais comuns de acessar a memória do *device*: utilizando ponteiros dentro do código do *device* e realizando chamadas ***cudaMemcpy()***.

O primeiro método de ponteiros dentro do código do *device* funciona da mesma forma que o padrão em C utilizado no *host*. Utilizando um ponteiro dentro do *device*, o resultado é armazenado na memória que é apontada. Vamos tomar como exemplo a seguinte expressão executada no *device*: ***c = a + b**. São somados os parâmetros **'a'** e **'b'** e o resultado é armazenado na memória apontada por **'c'**. Para isso, é necessário passar um ponteiro por parâmetro, como no código utilizado na Figura 7.

Além disso, podemos acessar a memória utilizando a função ***cudaMemcpy()*** a partir do código *host*. Esta função funciona exatamente como ***memcpy()*** em C padrão. O primeiro parâmetro é a variável a receber a cópia (destino), o segundo é a variável de origem, o terceiro é o tamanho que será copiado e o último, diferente do C padrão, é uma instrução

informando a origem dos ponteiros, se do *host* ou *device*. No exemplo abaixo, *cudaMemcpyDeviceToHost* informa que o ponteiro de origem pertence ao *device* e o que receberá a cópia será do *host.* O oposto seria *cudaMemcpyHostToDevice,* onde a fonte é no *host* e o destino é no *device*.

```
cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
```

5. Exemplo de Paralelização de Operações em Vetor

Neste capítulo será demonstrado e explicado um algoritmo que atribui a um vetor o valor de seu índice elevado ao quadrado utilizando processamento em GPU com CUDA. Imagine que tenhamos uma lista muito grande de números e desejamos exibir o valor de cada índice do vetor elevado ao quadrado. Isso será feito utilizando vetores de forma paralela. A Figura 8 ilustra nosso objetivo.

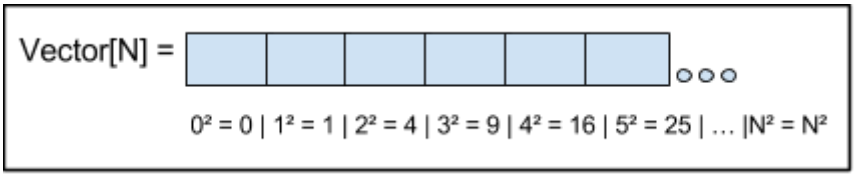

Figura 8. Vetor contém seu índice elevado ao quadrado. [4]

Uma solução tradicional com implementação sequencial, resolveria o problema utilizando um loop de repetição (for, while, etc.) iterando em cada posição do vetor e realizando uma multiplicação. Desta maneira, a CPU iria realizar todo o processamento sequencialmente (uma iteração do loop de cada vez após a outra). Utilizando CUDA, podem ser utilizados os núcleos da GPU para o processamento das iterações do loop de forma totalmente paralela, ou seja, ao mesmo tempo. Cada iteração do loop será executada separadamente em cada *core* (núcleo) de forma paralela. A Figura 9 ilustra a divisão em mais CPUs do processamento referido em uma função que será executada nos núcleos da GPU. A variável **"tid"** representa o índice do loop, que é a posição do vetor que será somada. Cada bloco possui seu **tid** para poder processar o loop completo separadamente. Além disso, a variável **N** é uma constante que define o tamanho dos vetores.

CPU 01

```
void add( int *vet) {
        int tid = 0;
        if (tid < N){
                vet[tid] = tid * tid;
        }
}
```

CPU 02

```
void add( int *vet) {
        int tid = 1;
        if (tid < N){
                vet[tid] = tid * tid;
        }
}
```

Figura 9. Execução do mesmo código em CPUs diferentes com valores diferentes.

5.1. A Função *main()*.

Na imagem a seguir será demonstrado o código em **C** utilizando **CUDA** para realizar o processamento. Mas, por enquanto, vamos observar apenas a função *main()*. Este código não possui nada que não tenhamos estudado ainda.

Podem ser observados alguns padrões utilizados tanto no código da Figura 10 como nos exemplos dos capítulos anteriores:

- Alocamos o array no *device* utilizando ***cudaMalloc()*** (array: dev_vet) para armazenar o resultado.
- Como programador consciente, sempre é utilizado ***cudaFree()*** para limpeza.
- Utilizando ***cudaMemcpy()***, os dados de entrada (vetores) são copiados para o *device* utilizando o parâmetro *cudaMemcpyHostToDevice*. O resultado no *device* é copiado de volta para o *host* utilizando *cudaMemcpyDeviceToHost*.
- O código de *device* é executado na função *multi()* a partir da *main()* do *host* utilizando a sintaxe de ***multi<<<N,1>>>***.

```
int main( void ) {
    int vet[N];
    int *dev_vet;

    // Aloca a memória na GPU
    cudaMalloc(&dev_vet, N * sizeof(int));

    // Copia o array 'vet' para a GPU (device)
    cudaMemcpy(dev_vet, vet, N * sizeof(int), cudaMemcpyHostToDevice);

    // Chamada para função na GPU.
    mult<<<N,1>>>(dev_vet);

    // Copia o array 'vet' de volta da GPU para a CPU
    cudaMemcpy(vet, dev_vet, N*sizeof(int), cudaMemcpyDeviceToHost);

    // Exibe os resultados
    for (int i=0; i<N; i++){
        printf("%d² = %d \n", i, vet[i]);
    }

    // libera memória alocada
    cudaFree(dev_vet);
    return 0;
}
```

Figura 10. Função *main()* vetor ao quadrado em CUDA.

No capítulo anterior, a explicação da chamada da função de *kernel* e os números dentro dos três sinais de maior e menor foi muito sucinta. Nos exemplos do capítulo anterior era utilizado somente **<<<1,1>>>** como parâmetros, no entanto, neste exemplo foi usado um número diferente de 1 da seguinte forma: **<<<N,1>>>**. O que isso significa?

O primeiro número desse parâmetro representa o número de blocos paralelos nos quais é desejado que execute a função *kernel*. Por exemplo, se for passado **<<<2,1>>>**, serão criadas 2 cópias do *kernel* e executadas em blocos paralelos. Isso facilita a programação paralela, pois apenas indicando **<<<256,1>>>**, já é possível obter 256 blocos executando em paralelo. No caso de **<<<N,1>>>**, N representa o tamanho do vetor, ou seja, todas as posições do vetor serão computadas em paralelo.

5.2. A função de *kernel* multi().

A Figura 11 demonstra a implementação da função de *kernel*, executada na GPU, chamada de ***multi()*** para multiplicação dos números e geração do vetor resultante. Observe que os parâmetros passados são os mesmo da chamada da função na *main()*.

```
__global__ void mult(int *vet){
    int tid = blockIdx.x;
    if (tid < N){
        vet[tid] = tid * tid;
    }
}
```

Figura 11. Função de *kernel* nomeada “add”.

À primeira vista, a variável **blockIdx.x** parece causar erro de sintaxe por ser utilizada como valor para **tid**, porém não ter sido declarada ou definida. No entanto, não é necessário definir a variável **blockIdx**. Esta é uma das variáveis incorporadas que CUDA define para nós. É uma variável muito importante, sendo utilizada para definir o **tid**, pois contém o valor do índice do bloco para qualquer bloco que estiver executando o código do *device*. Conforme dito anteriormente, definimos o número de blocos em paralelo no seguinte parâmetro **<<<N,1>>>**. A variável **blockIdx.x** identifica o índice de cada um dos blocos, representando cada iteração do loop sequencial que era necessário para percorrer o vetor. Por isso, não utilizamos um loop (for, while, etc) em qualquer lugar do código, tanto no *host* como *device*. O número de iterações é definido pelo número de blocos paralelos. A Figura 12 ilustra os blocos com cópias da função *kernel* com valores da variável **tid** diferentes executados em paralelo.

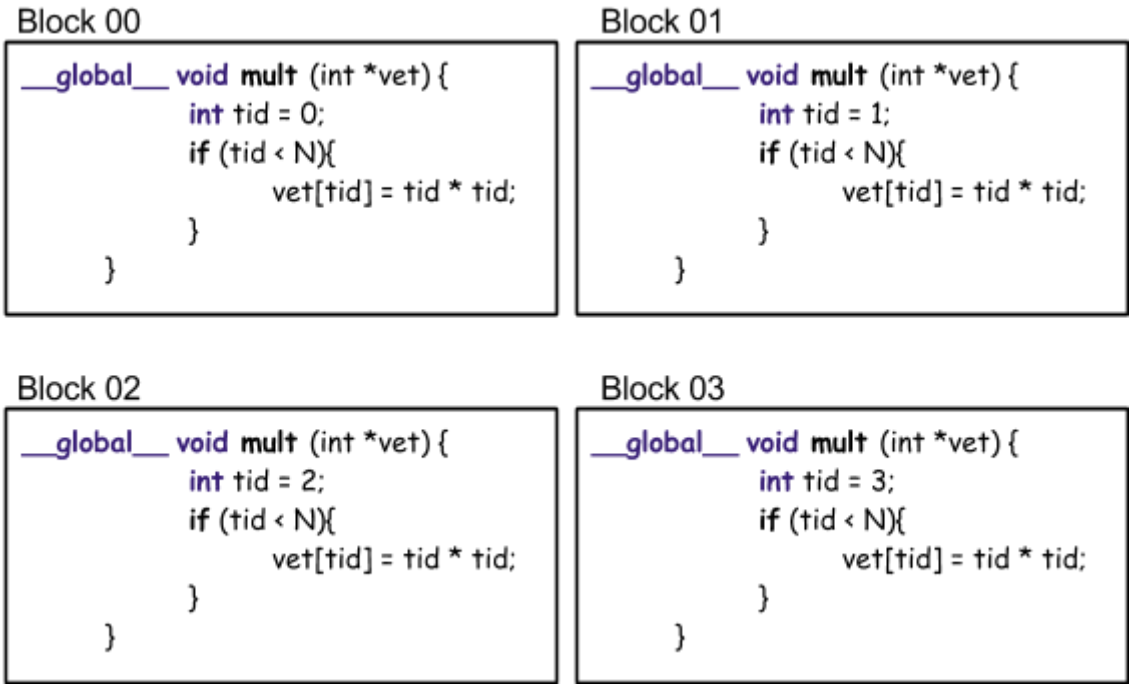

Figura 12. Chamada de *kernel* em diferentes blocos.

Dessa forma, a computação paralela se torna bastante simples. No exemplo, utilizamos um vetor apenas com 10 posições, porém podem ser usados vetores de 1.000, 10.000 ou 100.000 posições, incluindo matrizes e problemas mais complexos e com necessidade de grande processamento de dados.

6. Conclusão

Conforme o conteúdo apresentado, CUDA é uma tecnologia que permite programação paralela utilizando os diversos núcleos presentes em uma placa de vídeo. Pode ser considerada uma linguagem que proporciona grande poder computacional e, ao mesmo tempo, esconde muitas das complexidades que seriam encontradas como na transferência de dados entre memórias ou na distribuição e mapeamento dos blocos paralelos entre os núcleos da GPU. Sendo assim, podemos desenvolver um código de forma que utilize programação concorrente sem muita complexidade de código, resultando em um alto desempenho comparado a outros programas quando executados de forma sequencial. Podemos usufruir desse desempenho para otimização em tempo de resposta para algoritmos e soluções de software desenvolvidas.

7. Referências

[1] MACHADO, J. “O que é GPU?”
Disponível em: https://www.tecmundo.com.br/hardware/1127-o-que-e-gpu-.htm
Acessado em Novembro/2017

[2] NVIDIA, “O QUE É COMPUTAÇÃO ACELERADA POR DE PLACAS DE VÍDEO?”
Disponível em: http://www.nvidia.com.br/object/what-is-gpu-computing-br.html
Acessado em Novembro/2017

[3] Barney, B. (2010). “Introduction to parallel computing.” *Lawrence Livermore National Laboratory*, 6(13):10

[4] SANDERS, J., KANDROT, E., “Cuda by example - An Introduction to General-Purpose GPU Programming”.

[5] MUHAMMAD, A., MOHAMED, G.,“CUDA and Caffe for deep learning”
Disponível em: https://pt.slideshare.net/AmgadMuhammad/cuda-and-caffe-for-deep-learning
Acessado em: Novembro/2017