

Heterogeneous Computing

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

int main() {
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc( (void **) &d_a, size);
    cudaMalloc( (void **) &d_b, size);
    cudaMalloc( (void **) &d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(d_c, &c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a);  cudaFree(d_b);  cudaFree(d_c);

    return 0;
}
```

Computação Usando Threads

```
_global_ void add (int *a, int *b, int *c ) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

#define N 512

int main(void) {
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc( (void **) &d_a, size);
    cudaMalloc( (void **) &d_b, size);
    cudaMalloc( (void **) &d_c, size);

    a = (int *) malloc(size); random_ints(a, N);
    b = (int *) malloc(size); random_ints(b, N);
    c = (int *) malloc(size);

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    add<<<1,N>>>(d_a, d_b, d_c);

    cudaMemcpy(d_c, c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a);  cudaFree(d_b);  cudaFree(d_c);

    return 0;
}
```

Computação Usando Blocks

```
_global_ void add (int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

#define N 512

int main(void) {
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size = N*sizeof(int);

    cudaMalloc( (void **) &d_a, size);
    cudaMalloc( (void **) &d_b, size);
    cudaMalloc( (void **) &d_c, size);

    a = (int *) malloc(size); random_ints(a, N);
    b = (int *) malloc(size); random_ints(b, N);
    c = (int *) malloc(size);

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // N cópias de add() em N blocks e 1 thread por block
    add<<< N,1 >>>(d_a, d_b, d_c);

    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a);  cudaFree(d_b);  cudaFree(d_c);

    return 0;
}
```

Combinando Blocks e Threads



Com **M threads** por **Block**, um único índice para cada **thread** é dado por:

```
int index = threadIdx.x + blockIdx.x * M;
```

Qual thread operará sobre o elemento em vermelho ?



Usamos a variável embutida (built-in) **blockDim.x** para **threads** por **block**.

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

```

_global_ void add (int *a, int *b, int *c ) {

    int index = a[blockIdx.x] + b[blockIdx.x] * blockDim.x;
    c[index]= a[index]+b[index];
}

#define N (2048*2048)
#define nTHREADS_PER_BLOCK 512

int main(void) {
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size = N*sizeof(int);

    cudaMalloc( (void **) &d_a, size);
    cudaMalloc( (void **) &d_b, size);
    cudaMalloc( (void **) &d_c, size);

    a = (int *) malloc(size); random_ints(a, N);
    b = (int *) malloc(size); random_ints(b, N);
    c = (int *) malloc(size);

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    int nTHREADS_PER_BLOCK = 512; // múltiplo de 32
    int nBLOCKS = N/nTHREADS_PER_BLOCK

    // chamada do kernel

    add<<<nBLOCKS, nTHREADS_PER_BLOCK>>>
        (d_a, d_b, d_c);

    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    cudaFree(d_a);  cudaFree(d_b);  cudaFree(d_c);
}

```

```
return 0;
}
```

Programa CUDA Simples

```
// Device code
```

```
__global__ void VecAdd(float* A, float*  
                    B, float* C, int n)
```

```
{  
    int i = threadIdx.x;  
    if (i < n)  
        C[i] = A[i] + B[i];  
}
```

```
// Host code
```

```
// "h significa host, enquanto d significa device)"
```

```
int main() {  
    int n = 5;  
    size_t size = n * sizeof(float);  
    float *d_A, *d_B, *d_C;
```

// "void*" é um ponteiro para algo. Mas `cudaMalloc()` precisa modificar o ponteiro dado (o próprio ponteiro, não para o qual o ponteiro aponta), então você precisa passar `"void **"` que é um ponteiro para o ponteiro (geralmente um ponteiro para a variável local que aponta para o

endereço de memória) tal que **cudaMalloc()** pode modificar o valor do ponteiro.

```
cudaMalloc((void**)&d_A, size);  
cudaMalloc((void**)&d_B, size);  
cudaMalloc((void**)&d_C, size);
```

// Entrada de dados dos vetores A e B via Host.

```
float h_A[] = {1,2,3,4,5};  
float h_B[] = {10,20,30,40,50};  
float h_C[] = {0,0,0,0,0};
```

// Copia os vetores A e B do Host para o Device.

```
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);  
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

// Define o número de threads por bloco.

```
int nThreadsPerBlock = 256 (múltiplo de 32);  
int nBlocks = n/nThreadsPerBlock; ????
```

// Chamada do kernel.

```
VecAdd<<<nBlocks,  
    nThreadsPerBlock>>>(d_A, d_B, d_C);
```

```

// Copia o vetor C da memória da GPU para a memória no
HOST.
    cudaMemcpy(h_C, d_C, size,
               cudaMemcpyDeviceToHost);

// Libera memória ocupada pelos vetores.
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}

```

Notar que a palavra-chave **__global__** significa a construção de um em CUDA. Isso simplesmente indica que essa função pode ser chamada do Host ou do dispositivo CUDA.

O próximo fato que você deve notar é como cada thread () descobre exatamente qual elemento de dados é responsável pela computação.

Cada thread executa o mesmo código, portanto, a única maneira de se diferenciar threads é usar o **threadIdx** () e o **blockIdx** ().

Organizando Threads:

Uma parte crítica do projeto de aplicativos CUDA é organizar threads, blocos de threads e grids de forma apropriada.

Para este aplicativo, a escolha mais simples é fazer com que cada thread calcule um elemento (uma entrada), e apenas uma thread, no array do resultado final.

Uma orientação geral é que um bloco deve consistir em pelo menos 192 threads para ocultar a latência do acesso à memória (Tempo de latência, é o tempo que ela demora para entregar os dados... são os tempos de espera para troca de dados... sendo assim, quanto menor o tempo para a entrega, mais rápido fica).

Portanto, 256 e 512 threads são números comuns e práticos. Para os propósitos deste exemplo, são selecionadas 256 threads por bloco.

Se nós temos n elementos de dados, nós necessitamos somente n threads, no sentido de computar a soma dos vetores ...

Assim, necessitamos o menor múltiplo de `threadsPerBlock` que é maior ou igual a n , por computar:

$$(n + (\text{threadsPerBlock} - 1)) / \text{threadsPerBlock}$$

Assim, o número de blocos lançados `nBlocks` deve ser 32 ou

$$(n + (\text{threadsPerBlock} - 1)) / \text{threadsPerBlock} .$$

Ver em: <https://books.google.com.br/books?id=.....>

Para compilar o código, basta utilizar o comando:

```
nvcc -o ex1 ex1.cu
```

Perguntas:

1) O programa funciona corretamente?

Teste seu funcionamento imprimindo o resultado obtido.

2) Corrija o programa.

3) Aumente o tamanho dos vetores para, por exemplo, 1024. Teste o resultado e, se não for o esperado, corrija o programa.

Obs: Você deve manter o número de threads por bloco em 256.

Referências:

https://eradsp2010.files.wordpress.com/2010/10/curso2_cuda_camargo.pdf

<https://en.wikipedia.org/wiki/CUDA>

<http://supercomputingblog.com/cuda/cuda-tutorial-2-the-kernel/>