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;
}
```

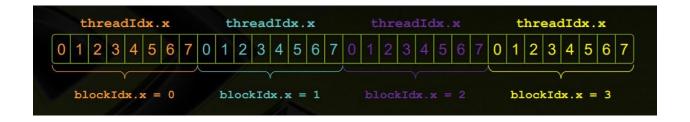
Computação Usando Threads

```
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;
}
```

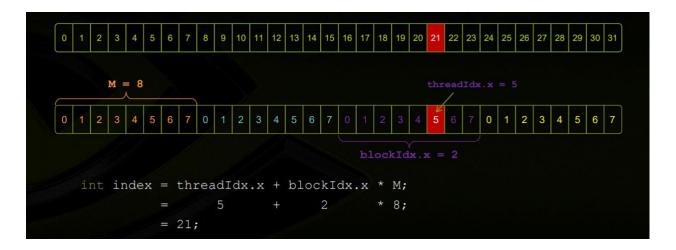
Combinando Blocks e Threads



Com M threads por Block, um único índex 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; // multiplo 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 + (threadsPerBlock-1))/threadsPerBlock

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

(n + (threadsPerBlock -1))/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/