**In proiectul meu**

Pentru comoditate, threadIdx este un vector cu trei componente, astfel încât firele pot fi identificate utilizând un indice de fir unidimensional, bidimensional sau tridimensional, formând un bloc unidimensional, bidimensional sau tridimensional Fire. Aceasta oferă o modalitate naturală de a invoca calculul pe elementele dintr-un domeniu, cum ar fi un vector, o matrice sau un volum.

Indicele unui fir și ID-ul firului acestuia se relaționează unul cu celălalt într-un mod simplu: pentru un bloc unidimensional ele sunt aceleași; Pentru un bloc de dimensiuni bidimensionale (Dx, Dy), ID-ul firului unui fir de index (x, y) este (x + y Dx); Pentru un bloc tridimensional de dimensiune (Dx, Dy, Dz), ID-ul firului unui index al indexului (x, y, z) este (x + y Dx + z Dx Dy).

CUDA C extinde C, permițând programatorului să definească funcții C, denumite kernels, care, atunci când sunt chemați, sunt executate N ori în paralel cu N diferite fire CUDA, spre deosebire de o singură dată ca și funcțiile regulate C.

Un nucleu este definit folosind specificatorul declarației \_\_global\_\_ și numărul de fire CUDA care execută acest kernel pentru un anumit apel de kernel este specificat utilizând o nouă sintaxă de configurare de execuție <<< ... >>> (vezi Extensii de limbă C). Fiecare fir care execută nucleul are un ID de thread unic, accesibil în cadrul kernelului, prin intermediul variabilei threadIdx încorporate.

Așa cum am menționat în programarea eterogenă, modelul de programare CUDA presupune un sistem compus dintr-o gazdă și un dispozitiv, fiecare cu memorie proprie separată. Kernelurile funcționează din memoria dispozitivului, astfel încât timpul de execuție oferă funcții de alocare, dealocare și copiere a memoriei dispozitivului, precum și transferul de date între memoria gazdă și memoria dispozitivului.  
  
Memoria dispozitivului poate fi alocată fie ca memorie liniară, fie ca matrice CUDA.  
  
Tablourile CUDA sunt opțiuni de memorie opace optimizate pentru preluarea texturii.Ele sunt descrise în textură și memorie de suprafață.  
  
Există o memorie liniară pe dispozitiv într-un spațiu de adrese pe 40 de biți, entitățile atribuite separat se pot referi reciproc prin pointeri, de exemplu, într-un arbore binar.  
  
Memoria liniară este, de obicei, alocată folosind cudaMalloc () și eliberată utilizând cudaFree (), iar transferul de date între memoria gazdă și memoria dispozitivului se face de obicei folosind cudaMemcpy (). În proba de cod de adăugare a vectorilor de kernel, vectorii trebuie să fie copiați din memoria gazdă în memoria dispozitivului:

// Device code

\_\_global\_\_ void VecAdd(float\* A, float\* B, float\* C, int N)

{

int i = blockDim.x \* blockIdx.x + threadIdx.x;

if (i < N)

C[i] = A[i] + B[i];

}

// Host code

int main()

{

int N = ...;

size\_t size = N \* sizeof(float);

// Allocate input vectors h\_A and h\_B in host memory

float\* h\_A = (float\*)malloc(size);

float\* h\_B = (float\*)malloc(size);

// Initialize input vectors

...

// Allocate vectors in device memory

float\* d\_A;

cudaMalloc(&d\_A, size);

float\* d\_B;

cudaMalloc(&d\_B, size);

float\* d\_C;

cudaMalloc(&d\_C, size);

// Copy vectors from host memory to device memory

cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);

// Invoke kernel

int threadsPerBlock = 256;

int blocksPerGrid =

(N + threadsPerBlock - 1) / threadsPerBlock;

VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C, N);

// Copy result from device memory to host memory

// h\_C contains the result in host memory

cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost);

// Free device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free host memory

...

}