b

* Allocating the CPU memory for A, B, and C matrix

// Initialize the memory on the host  
float \*a, \*b, \*c;  
  
// Allocate host memory  
a = (float\*)malloc(sizeof(float) \* (N\*N));  
b = (float\*)malloc(sizeof(float) \* (N\*N));  
c = (float\*)malloc(sizeof(float) \* (N\*N));

* Allocating the GPU memory for A, B, and C matrix

// Initialize the memory on the device  
float \*d\_a, \*d\_b, \*d\_c;  
  
// Allocate device memory  
cudaMalloc((void\*\*)&d\_a, sizeof(float) \* (N\*N));  
cudaMalloc((void\*\*)&d\_b, sizeof(float) \* (N\*N));  
cudaMalloc((void\*\*)&d\_c, sizeof(float) \* (N\*N));

* Now we need to fill the values for the matrix a and b.

// Initialize host matrix  
for(int i = 0; i < (N\*N); i++)  
 {  
 a[i] = 2.0f;  
 b[i] = 2.0f;  
 }

* Transfer initialized A and B matrix from CPU to GPU

cudaMemcpy(d\_a, a, sizeof(float) \* (N\*N), cudaMemcpyHostToDevice);  
cudaMemcpy(d\_b, b, sizeof(float) \* (N\*N), cudaMemcpyHostToDevice);

* 2D thread block for indexing x and y

// Thread organization  
int blockSize = 32;  
dim3 dimBlock(blockSize,blockSize,1);  
dim3 dimGrid(ceil(N/float(blockSize)),ceil(N/float(blockSize)),1);

* Calling the kernel function

// Device fuction call  
matrix\_mul<<<dimGrid,dimBlock>>>(d\_a, d\_b, d\_c, N);

* ??? “matrix multiplication function call”
* === "serial"  
   ```c  
   float \* matrix\_mul(float \*h\_a, float \*h\_b, float \*h\_c, int width)  
   {  
   for(int row = 0; row < width ; ++row)  
   {  
   for(int col = 0; col < width ; ++col)  
   {  
   float temp = 0;  
   for(int i = 0; i < width ; ++i)  
   {  
   temp += h\_a[row\*width+i] \* h\_b[i\*width+col];  
   }  
   h\_c[row\*width+col] = temp;  
   }  
   }  
   return h\_c;  
   }  
   ```  
  === "cuda"  
   ```c  
   \_\_global\_\_ void matrix\_mul(float\* d\_a, float\* d\_b,   
   float\* d\_c, int width)  
   {  
   int row = blockIdx.x \* blockDim.x + threadIdx.x;  
   int col = blockIdx.y \* blockDim.y + threadIdx.y;  
    
   if ((row < width) && (col < width))   
   {  
   float temp = 0;  
   // each thread computes one   
   // element of the block sub-matrix  
   for (int i = 0; i < width; ++i)   
   {  
   temp += d\_a[row\*width+i]\*d\_b[i\*width+col];  
   }  
   d\_c[row\*width+col] = temp;  
   }  
   }  
   ```
* Transfer back (copy back to ) the computed c matrix to host from GPU

// Transfer data back to host memory  
cudaMemcpy(c, d\_c, sizeof(float) \* (N\*N), cudaMemcpyDeviceToHost);

* Copy back computed value from GPU to CPU

// Transfer data back to host memory  
cudaMemcpy(c, d\_c, sizeof(float) \* (N\*N), cudaMemcpyDeviceToHost);

* Deallocate the host and device memory

// Deallocate device memory  
cudaFree(d\_a);  
cudaFree(d\_b);  
cudaFree(d\_c);  
  
// Deallocate host memory  
free(a);   
free(b);   
free(c);

### Questions and Solutions

??? example “Examples: Matrix Multiplication”

=== "Serial-version"  
 ```c  
 //-\*-C++-\*-  
 // Matrix-multiplication.c  
   
 #include<iostream>  
 #include<cuda.h>  
   
 using namespace std;  
   
 float \* matrix\_mul(float \*h\_a, float \*h\_b, float \*h\_c, int width)   
 {   
 for(int row = 0; row < width ; ++row)   
 {   
 for(int col = 0; col < width ; ++col)   
 {   
 float temp = 0;   
 for(int i = 0; i < width ; ++i)   
 {   
 temp += h\_a[row\*width+i] \* h\_b[i\*width+col];   
 }   
 h\_c[row\*width+col] = temp;   
 }   
 }   
 return h\_c;   
 }  
  
 int main()  
 {  
  
 cout << "Programme assumes that matrix (square matrix )size is N\*N "<<endl;  
 cout << "Please enter the N size number "<< endl;  
 int N = 0;  
 cin >> N;  
  
 // Initialize the memory on the host  
 float \*a, \*b, \*c;   
  
 // Allocate host memory  
 a = (float\*)malloc(sizeof(float) \* (N\*N));  
 b = (float\*)malloc(sizeof(float) \* (N\*N));  
 c = (float\*)malloc(sizeof(float) \* (N\*N));  
  
 // Initialize host matrix  
 for(int i = 0; i < (N\*N); i++)  
 {  
 a[i] = 1.0f;  
 b[i] = 2.0f;  
 }  
  
 // Device fuction call   
 matrix\_mul(a, b, c, N);  
  
 // Verification  
 for(int i = 0; i < N; i++)  
 {  
 for(int j = 0; j < N; j++)  
 {  
 cout << c[j] <<" ";  
 }  
 cout << " " <<endl;  
 }  
  
 // Deallocate host memory  
 free(a);   
 free(b);   
 free(c);  
  
 return 0;  
 }  
 ```  
  
=== "CUDA-template"  
  
 ```c  
 //-\*-C++-\*-  
 // Matrix-multiplication-template.cu  
   
 #include<iostream>  
 #include<cuda.h>  
   
 using namespace std;  
   
 \_\_global\_\_ void matrix\_mul(float\* d\_a, float\* d\_b,   
 float\* d\_c, int width)  
 {  
  
 // create a 2d threads block  
 int row = ..................  
 int col = ....................  
  
 // only allow the threads that are needed for the computation   
 if (................................)  
 {  
 float temp = 0;  
 // each thread computes one   
 // element of the block sub-matrix  
 for (int i = 0; i < width; ++i)   
 {  
 temp += d\_a[row\*width+i]\*d\_b[i\*width+col];  
 }  
 d\_c[row\*width+col] = temp;  
 }  
 }  
  
 // Host call (matix multiplication)  
 float \* cpu\_matrix\_mul(float \*h\_a, float \*h\_b, float \*h\_c, int width)   
 {   
 for(int row = 0; row < width ; ++row)   
 {   
 for(int col = 0; col < width ; ++col)   
 {   
 float single\_entry = 0;   
 for(int i = 0; i < width ; ++i)   
 {   
 single\_entry += h\_a[row\*width+i] \* h\_b[i\*width+col];   
 }   
 h\_c[row\*width+col] = single\_entry;   
 }   
 }   
 return h\_c;   
 }  
  
 int main()  
 {  
  
 cout << "Programme assumes that matrix (square matrix) size is N\*N "<<endl;  
 cout << "Please enter the N size number "<< endl;  
 int N = 0;  
 cin >> N;  
  
 // Initialize the memory on the host  
 float \*a, \*b, \*c, \*host\_check;   
  
 // Initialize the memory on the device  
 float \*d\_a, \*d\_b, \*d\_c;   
  
 // Allocate host memory  
 a = (float\*)malloc(sizeof(float) \* (N\*N));  
 ...  
 ...  
   
 // Initialize host matrix  
 for(int i = 0; i < (N\*N); i++)  
 {  
 a[i] = 2.0f;  
 b[i] = 2.0f;  
 }  
  
 // Allocate device memory  
 cudaMalloc((void\*\*)&d\_a, sizeof(float) \* (N\*N));  
 ...  
 ...  
   
 // Transfer data from host to device memory  
 cudaMemcpy(.........................);  
 cudaMemcpy(.........................);  
  
 // Thread organization  
 int blockSize = ..............;  
 dim3 dimBlock(......................);  
 dim3 dimGrid(.......................);  
  
 // Device fuction call   
 matrix\_mul<<<dimGrid,dimBlock>>>(d\_a, d\_b, d\_c, N);  
  
 // Transfer data back to host memory  
 cudaMemcpy(c, d\_c, sizeof(float) \* (N\*N), cudaMemcpyDeviceToHost);  
  
 // CPU computation for verification   
 cpu\_matrix\_mul(a,b,host\_check,N);  
  
 // Verification  
 bool flag=1;  
 for(int i = 0; i < N; i++)  
 {  
 for(int j = 0; j < N; j++)  
 {  
 if(c[j\*N+i]!= host\_check[j\*N+i])  
 {  
 flag=0;  
 break;  
 }  
 }  
 }  
 if (flag==0)  
 {  
 cout <<"Two matrices are not equal" << endl;  
 }  
 else  
 cout << "Two matrices are equal" << endl;  
  
 // Deallocate device memory  
 cudaFree...  
  
 // Deallocate host memory  
 free...  
  
 return 0;  
 }  
 ```  
   
=== "CUDA-version"  
  
 ```c  
 //-\*-C++-\*-  
 // Matrix-multiplication.cu  
   
 #include<iostream>  
 #include<cuda.h>  
   
 using namespace std;  
   
 \_\_global\_\_ void matrix\_mul(float\* d\_a, float\* d\_b,   
 float\* d\_c, int width)  
 {  
  
 int row = blockIdx.x \* blockDim.x + threadIdx.x;  
 int col = blockIdx.y \* blockDim.y + threadIdx.y;  
  
 if ((row < width) && (col < width))   
 {  
 float temp = 0;  
 // each thread computes one   
 // element of the block sub-matrix  
 for (int i = 0; i < width; ++i)   
 {  
 temp += d\_a[row\*width+i]\*d\_b[i\*width+col];  
 }  
 d\_c[row\*width+col] = temp;  
 }  
 }  
  
 // Host call (matix multiplication)  
 float \* cpu\_matrix\_mul(float \*h\_a, float \*h\_b, float \*h\_c, int width)   
 {   
 for(int row = 0; row < width ; ++row)   
 {   
 for(int col = 0; col < width ; ++col)   
 {   
 float single\_entry = 0;   
 for(int i = 0; i < width ; ++i)   
 {   
 single\_entry += h\_a[row\*width+i] \* h\_b[i\*width+col];   
 }   
 h\_c[row\*width+col] = single\_entry;   
 }   
 }   
 return h\_c;   
 }  
  
  
 int main()  
 {  
  
 cout << "Programme assumes that matrix (square matrix) size is N\*N "<<endl;  
 cout << "Please enter the N size number "<< endl;  
 int N = 0;  
 cin >> N;  
  
 // Initialize the memory on the host  
 float \*a, \*b, \*c, \*host\_check;   
  
 // Initialize the memory on the device  
 float \*d\_a, \*d\_b, \*d\_c;   
  
 // Allocate host memory  
 a = (float\*)malloc(sizeof(float) \* (N\*N));  
 b = (float\*)malloc(sizeof(float) \* (N\*N));  
 c = (float\*)malloc(sizeof(float) \* (N\*N));  
 host\_check = (float\*)malloc(sizeof(float) \* (N\*N));  
  
 // Initialize host matrix  
 for(int i = 0; i < (N\*N); i++)  
 {  
 a[i] = 2.0f;  
 b[i] = 2.0f;  
 }  
  
 // Allocate device memory  
 cudaMalloc((void\*\*)&d\_a, sizeof(float) \* (N\*N));  
 cudaMalloc((void\*\*)&d\_b, sizeof(float) \* (N\*N));  
 cudaMalloc((void\*\*)&d\_c, sizeof(float) \* (N\*N));  
  
 // Transfer data from host to device memory  
 cudaMemcpy(d\_a, a, sizeof(float) \* (N\*N), cudaMemcpyHostToDevice);  
 cudaMemcpy(d\_b, b, sizeof(float) \* (N\*N), cudaMemcpyHostToDevice);  
  
 // Thread organization  
 int blockSize = 32;  
 dim3 dimBlock(blockSize,blockSize,1);  
 dim3 dimGrid(ceil(N/float(blockSize)),ceil(N/float(blockSize)),1);  
  
 // Device fuction call   
 matrix\_mul<<<dimGrid,dimBlock>>>(d\_a, d\_b, d\_c, N);  
  
 // Transfer data back to host memory  
 cudaMemcpy(c, d\_c, sizeof(float) \* (N\*N), cudaMemcpyDeviceToHost);  
  
 // cpu computation for verification   
 cpu\_matrix\_mul(a,b,host\_check,N);  
  
 // Verification  
 bool flag=1;  
 for(int i = 0; i < N; i++)  
 {  
 for(int j = 0; j < N; j++)  
 {  
 if(c[j\*N+i]!= host\_check[j\*N+i])  
 {  
 flag=0;  
 break;  
 }  
 }  
 }  
 if (flag==0)  
 {  
 cout <<"Two matrices are not equal" << endl;  
 }  
 else  
 cout << "Two matrices are equal" << endl;  
  
 // Deallocate device memory  
 cudaFree(d\_a);  
 cudaFree(d\_b);  
 cudaFree(d\_c);  
  
 // Deallocate host memory  
 free(a);   
 free(b);   
 free(c);  
 free(host\_check);  
  
 return 0;  
 }  
 ```

??? “Compilation and Output”

=== "Serial-version"  
 ```c  
 // compilation  
 $ gcc Matrix-multiplication.c -o Matrix-Multiplication-CPU  
   
 // execution   
 $ ./Matrix-Multiplication-CPU  
   
 // output  
 $ g++ Matrix-multiplication.cc -o Matrix-multiplication  
 $ ./Matrix-multiplication  
 Programme assumes that matrix (square matrix) size is N\*N   
 Please enter the N size number   
 4  
 16 16 16 16   
 16 16 16 16   
 16 16 16 16   
 16 16 16 16   
 ```  
   
=== "CUDA-version"  
 ```c  
 // compilation  
 $ nvcc -arch=compute\_70 Matrix-multiplication.cu -o Matrix-Multiplication-GPU  
   
 // execution  
 $ ./Matrix-Multiplication-GPU  
 Programme assumes that matrix (square matrix) size is N\*N   
 Please enter the N size number  
 $ 256  
   
 // output  
 $ Two matrices are equal  
 ```

??? Question “Questions”

- What happens if you remove the \*\*`\_\_syncthreads();`\*\* from the \*\*`\_\_global\_\_ void vector\_add(float \*a, float \*b,   
 float \*out, int n)`\*\* function.  
- Can you remove the if condition \*\*`if(i < n)`\*\* from the \*\*`\_\_global\_\_ void vector\_add(float \*a, float \*b,  
 float \*out, int n)`\*\* function. If so how can you do that?  
- Here we do not use the \*\*`cudaDeviceSynchronize()`\*\* in the main application, can you figure out why we  
 do not need to use it.   
- Can you create a different kinds of threads block for larger number of array?