cudaFuncSetCacheConfig(dkernel, cudaFuncCachePreferL1);  
//cudaFuncSetCacheConfig(dkernel, cudaFuncCachePreferShared);  
  
  
cudaFuncCachePreferNone: no preference for shared memory or L1 (default)  
cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache  
cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory  
cudaFuncCachePreferEqual: prefer equal size L1 cache and shared memory

??? example “Example: Shared Memory - Matrix Multiplication”

=== "Matrix-multiplication-shared-template"  
  
 ```c  
 // Matrix-multiplication-shared-template.cu  
 //-\*-C++-\*-  
 #include<iostream>  
 #include<cuda.h>  
  
 // block size for the matrix   
 #define BLOCK\_SIZE 16  
  
 using namespace std;  
  
 // Devicae call (matrix multiplication)  
 \_\_global\_\_ void matrix\_mul(const float \*d\_a, const float \*d\_b,   
 float \*d\_c, int width)  
 {  
 // Shared memory allocation for the block matrix   
 \_\_shared\_\_ int a\_block[BLOCK\_SIZE][BLOCK\_SIZE];  
 ...  
  
 // Indexing for the block matrix  
 int tx = threadIdx.x;  
 ...  
  
 // Indexing global matrix to block matrix   
 int row = threadIdx.x+blockDim.x\*blockIdx.x;  
 ...  
  
 // Allow threads only for size of rows and columns (we assume square matrix)  
 if ((row < width) && (col< width))  
 {  
 // Save temporary value for the particular index  
 float temp = 0;  
 for(int i = 0; i < width / BLOCK\_SIZE; ++i)  
 {  
 // Allign the global matrix to block matrix   
 a\_block[ty][tx] = d\_a[row \* width + (i \* BLOCK\_SIZE + tx)];  
 b\_block[ty][tx] = d\_b[(i \* BLOCK\_SIZE + ty) \* width + col];  
  
 // Make sure all the threads are synchronized  
 ....  
  
 // Multiply the block matrix   
 for(int j = 0; j < BLOCK\_SIZE; ++j)  
 {  
 temp += a\_block[ty][j] \* b\_block[j][tx];   
 }  
 // Make sure all the threads are synchronized  
 ...  
 }  
 // Save block matrix entry to global matrix   
 ...  
 }  
 }  
  
 // 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 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 size is N\*N "<<endl;  
 cout << "Matrix dimensions are assumed to be multiples of BLOCK\_SIZE=16" << 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 arrays  
 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);  
 cudaMemcpy(d\_c, c, sizeof(float) \* (N\*N), cudaMemcpyHostToDevice);  
  
 // Thread organization  
 dim3 Block\_dim(BLOCK\_SIZE, BLOCK\_SIZE, 1);   
 ...  
  
 // Device fuction call   
 matrix\_mul<<<Grid\_dim, Block\_dim>>>(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 <<"But,two matrices are not equal" << endl;  
 cout <<"Matrix dimensions are assumed to be multiples of BLOCK\_SIZE=16" << 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;  
 }  
 ```  
  
=== "Matrix-multiplication-shared.cu"  
  
 ```c  
 // Matrix-multiplication-shared.cu  
 //-\*-C++-\*-  
 #include<iostream>  
 #include<cuda.h>  
   
 // block size for the matrix   
 #define BLOCK\_SIZE 16  
   
 using namespace std;  
   
 // Device call (matrix multiplication)  
 \_\_global\_\_ void matrix\_mul(const float \*d\_a, const float \*d\_b,   
 float \*d\_c, int width)  
 {  
 // Shared memory allocation for the block matrix   
 \_\_shared\_\_ int a\_block[BLOCK\_SIZE][BLOCK\_SIZE];  
 \_\_shared\_\_ int b\_block[BLOCK\_SIZE][BLOCK\_SIZE];  
  
 // Indexing for the block matrix  
 int tx = threadIdx.x;  
 int ty = threadIdx.y;  
  
 // Indexing global matrix to block matrix   
 int row = threadIdx.x+blockDim.x\*blockIdx.x;  
 int col = threadIdx.y+blockDim.y\*blockIdx.y;  
  
 // Allow threads only for size of rows and columns (we assume square matrix)  
 if ((row < width) && (col< width))  
 {  
 // Save temporary value for the particular index  
 float temp = 0;  
 for(int i = 0; i < width / BLOCK\_SIZE; ++i)  
 {  
 // Allign the global matrix to block matrix   
 a\_block[ty][tx] = d\_a[row \* width + (i \* BLOCK\_SIZE + tx)];  
 b\_block[ty][tx] = d\_b[(i \* BLOCK\_SIZE + ty) \* width + col];  
  
 // Make sure all the threads are synchronized  
 \_\_syncthreads();   
  
 // Multiply the block matrix  
 for(int j = 0; j < BLOCK\_SIZE; ++j)  
 {  
 temp += a\_block[ty][j] \* b\_block[j][tx];   
 }  
 \_\_syncthreads();  
 }  
 // Save block matrix entry to global matrix   
 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 size is N\*N "<<endl;  
 cout << "Matrix dimensions are assumed to be multiples of BLOCK\_SIZE=16" << 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 arrays  
 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);  
 cudaMemcpy(d\_c, c, sizeof(float) \* (N\*N), cudaMemcpyHostToDevice);  
  
 // Thread organization  
 dim3 Block\_dim(BLOCK\_SIZE, BLOCK\_SIZE, 1);   
 dim3 Grid\_dim(ceil(N/BLOCK\_SIZE), ceil(N/BLOCK\_SIZE), 1);  
  
 // Device fuction call   
 matrix\_mul<<<Grid\_dim, Block\_dim>>>(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 <<"But,two matrices are not equal" << endl;  
 cout <<"Matrix dimensions are assumed to be multiples of BLOCK\_SIZE=16" << 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”

=== "CUDA-version"  
 ```c  
 // compilation  
 $ nvcc -arch=sm\_70 Matrix-multiplication-shared.cu -o Matrix-multiplication-shared  
   
 // execution  
 $ ./Matrix-multiplication-shared  
 Programme assumes that matrix size is N\*N   
 Matrix dimensions are assumed to be multiples of BLOCK\_SIZE=16  
 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?