

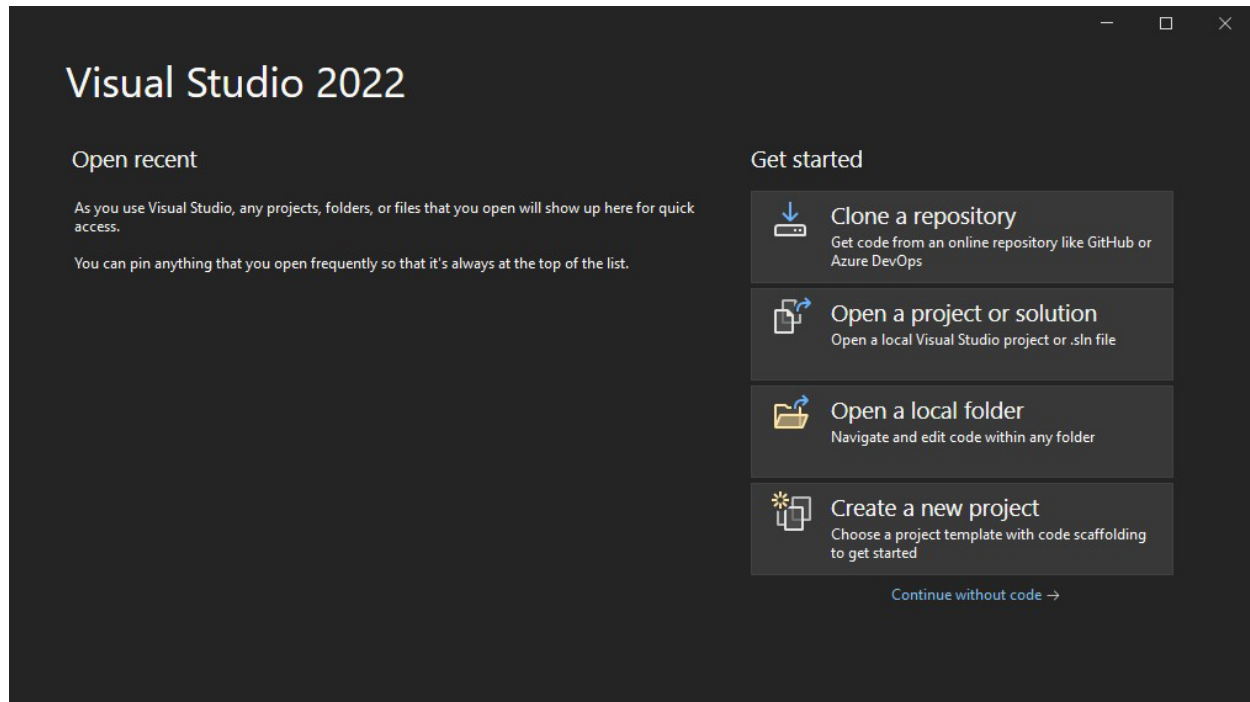
ELEC 522 Visual Studio New CUDA Project

Fall 2022

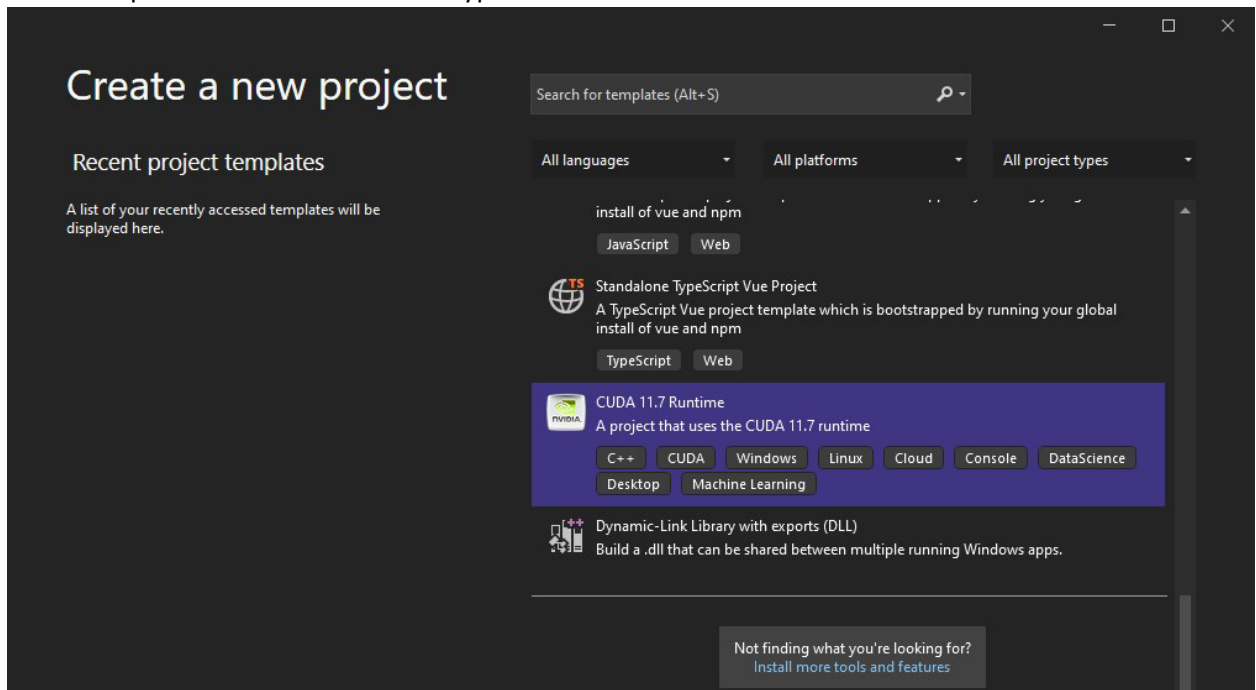
This tutorial runs through creating and running a visual studio 2022 project for a Nvidia CUDA file that starts out as a basic file with a extension of “.cu”. It is a C language file but with extensions for GPU kernel functions. As such it is compiled with “nvcc” which is the Nvidia C compiler with support for CUDA. If you were on a Linux system then you would type just “nvcc” but in Visual Studio we will use a CUDA template project to start and then replace the template file with our own test case. This ensure that we have the proper compiler options and header include file paths.

The following screen shots will walk you through the process.

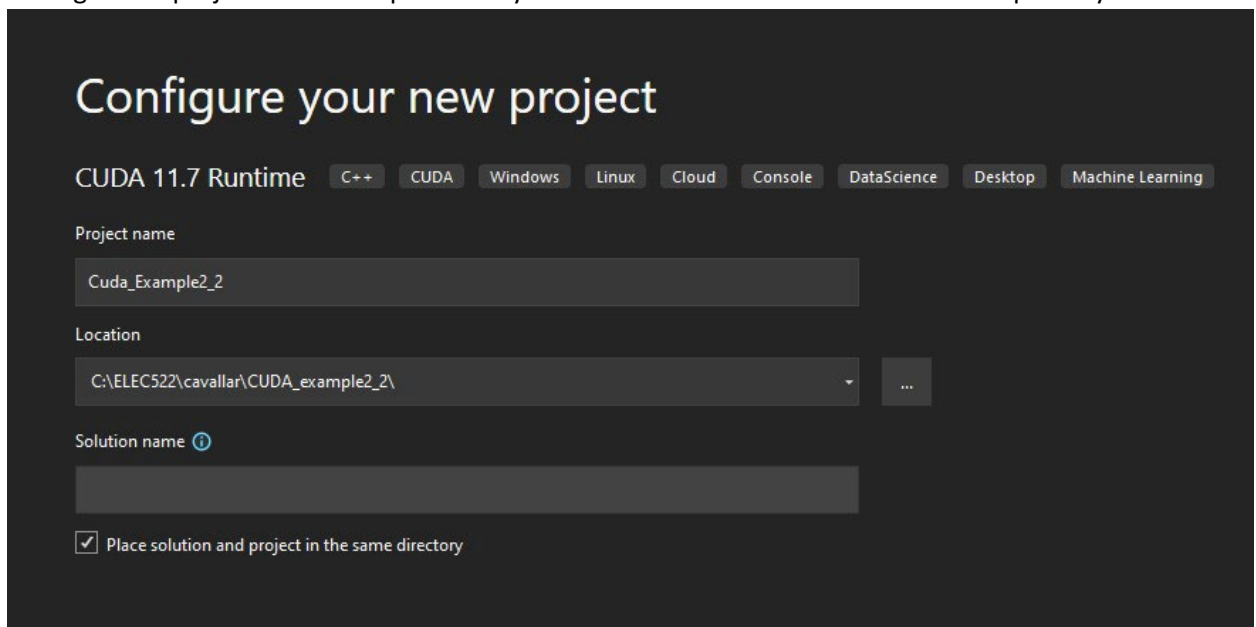
1. Create a new project.



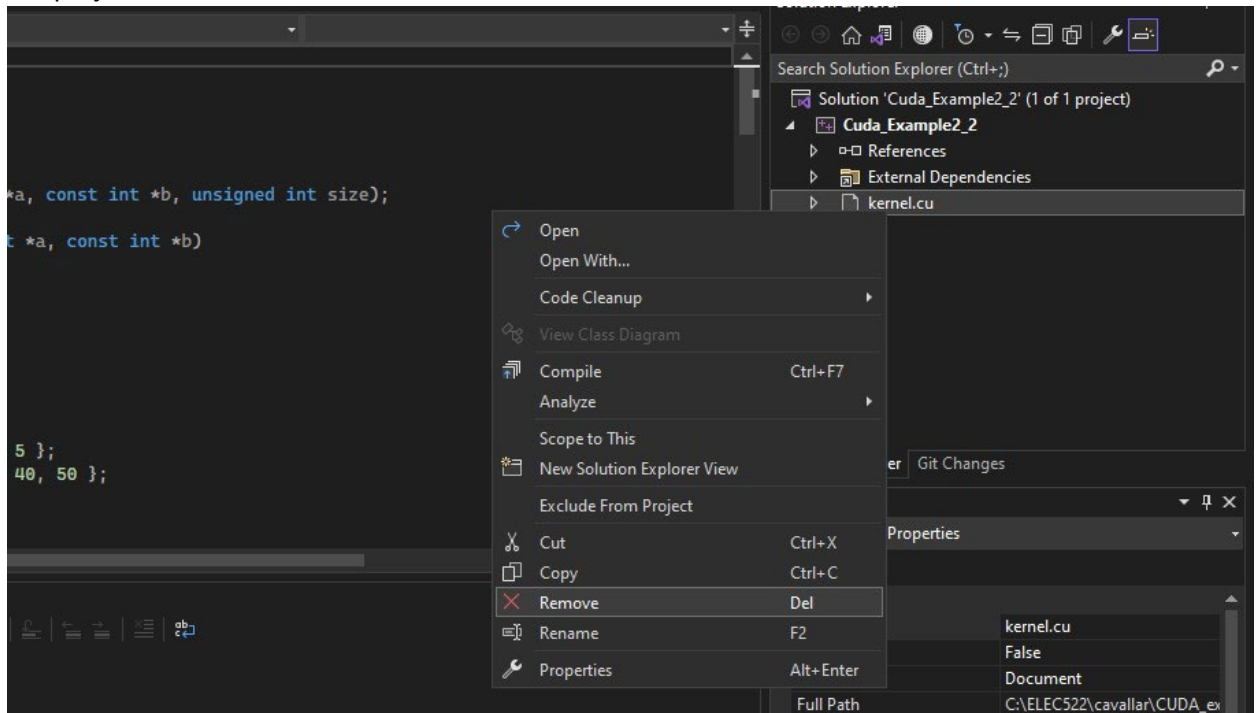
2. Pick a template of the CUDA runtime type.



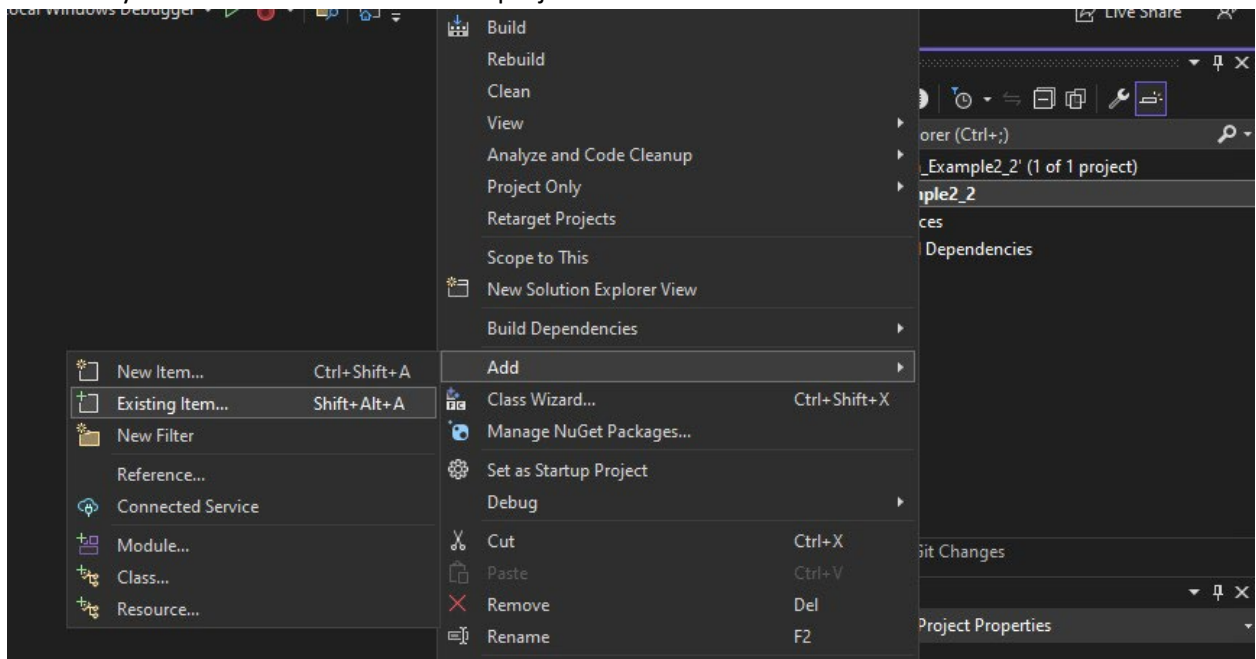
3. Then give it a project name and place it in your local area. Mine is shown as an example only.



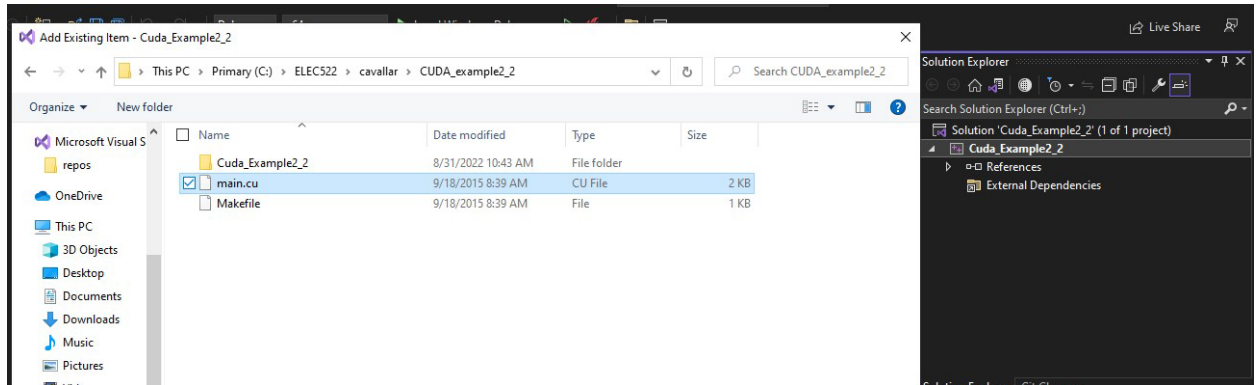
- Now Remove the default “kernel.cu” file. You could delete it if you wish. Remove takes it out of the project but leaves it in the folder.



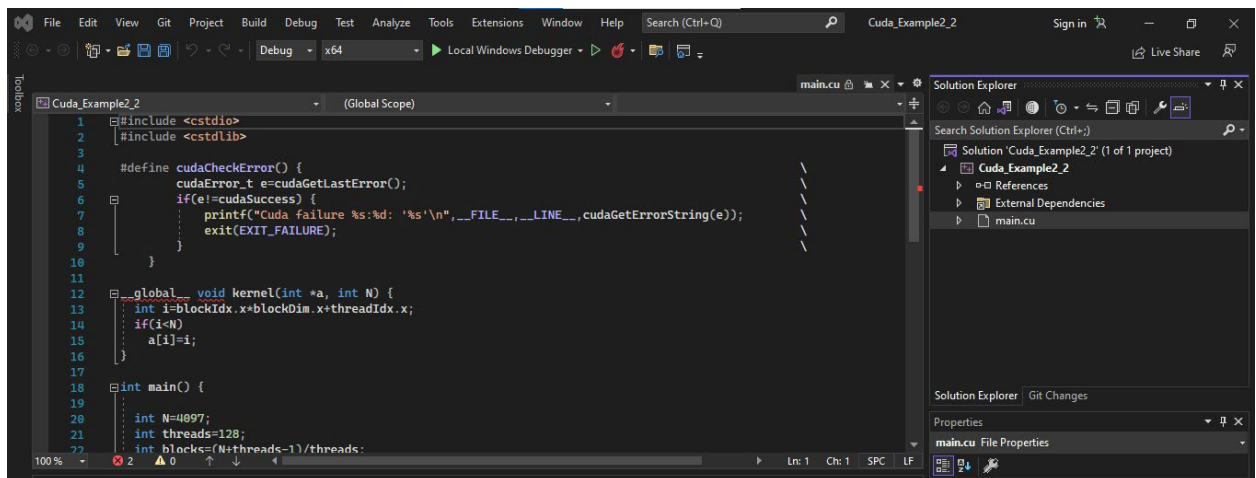
- Now add you CUDA .cu source file to the project.



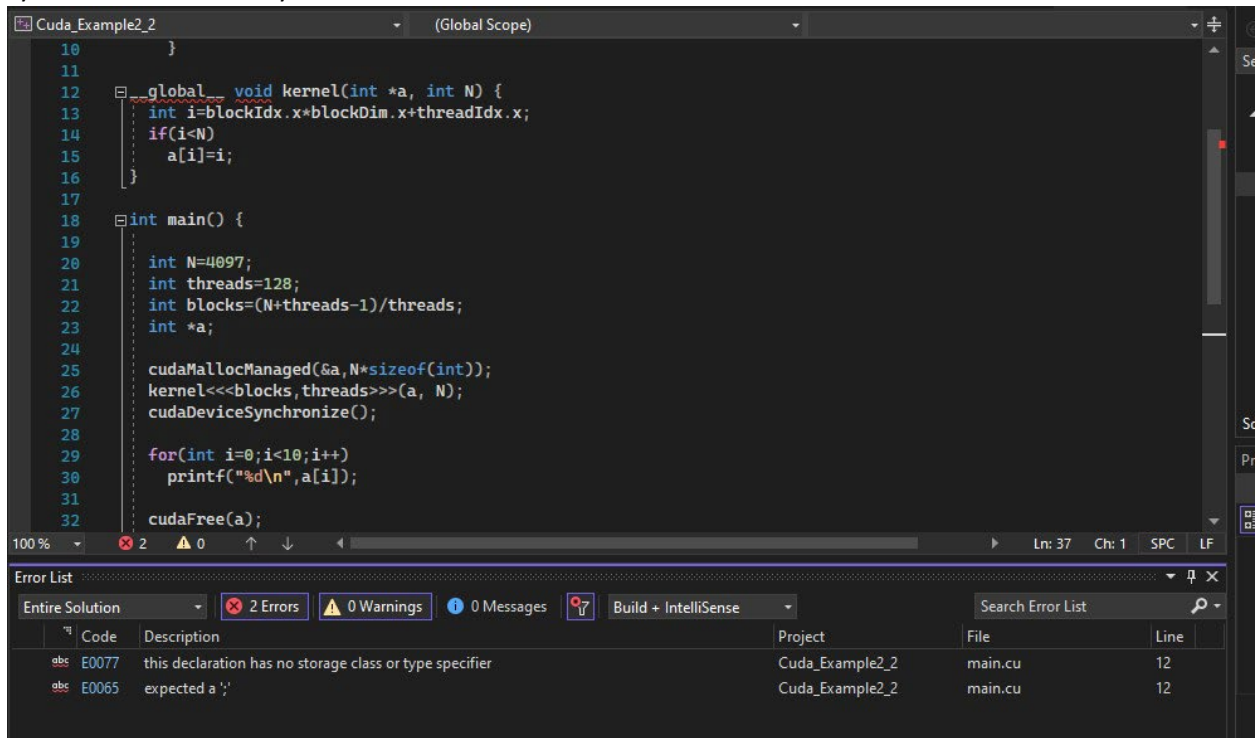
6. Find the source file in the folder where you may have saved it. This is “add existing.” You often may be starting a new CUDA file from an existing previous file.



7. You can now view the source code of this test file which will print out the 10 numbers 0 to 9 on a new line. The GPU creates the numbers in the kernel. The CPU does the printing to the screen later in the file. Notice that the Visual Studio intellisense guide is set up for C++ and does not understand some of the CUDA commands in the kernel definition and later in calling the kernel. This is OK.



8. Visual studio again may show some “errors” but these can be ignored as they relate to C++ syntax and not CUDA syntax.



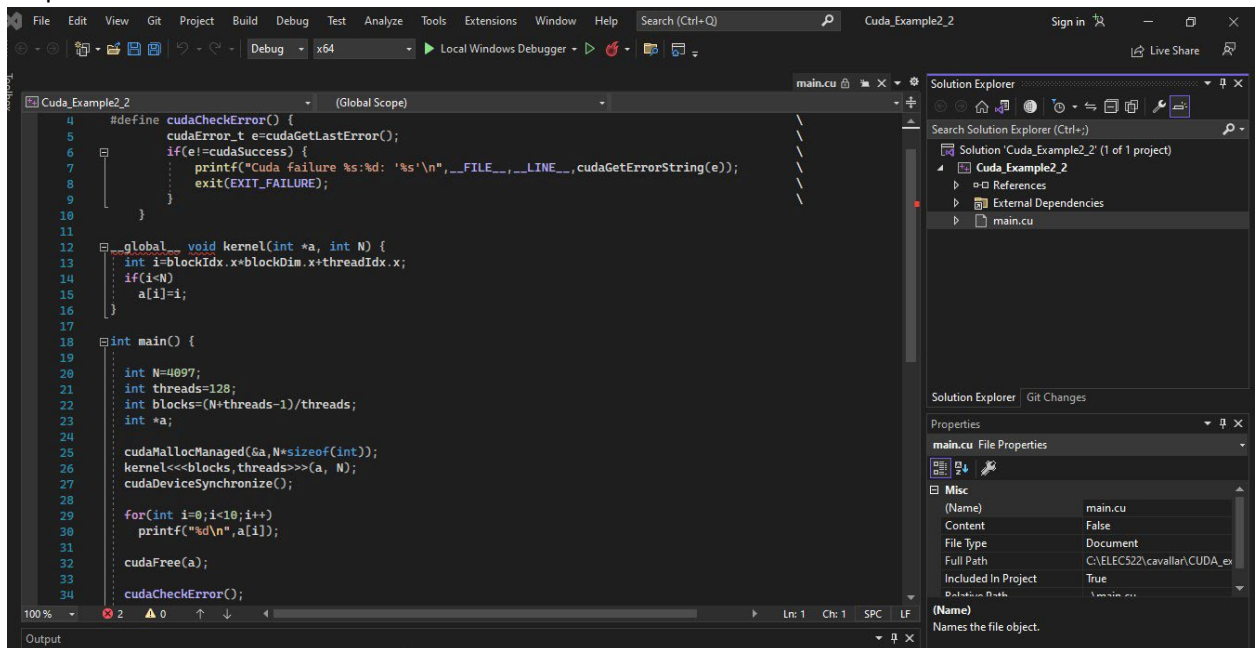
```
10 }
11
12 global void kernel(int *a, int N) {
13     int i=blockIdx.x*blockDim.x+threadIdx.x;
14     if(i<N)
15         a[i]=i;
16 }
17
18 int main() {
19     int N=4097;
20     int threads=128;
21     int blocks=(N+threads-1)/threads;
22     int *a;
23
24     cudaMallocManaged(&a,N*sizeof(int));
25     kernel<<<blocks,threads>>>(a, N);
26     cudaDeviceSynchronize();
27
28     for(int i=0;i<10;i++)
29         printf("%d\n",a[i]);
30
31     cudaFree(a);
32 }
```

100% 2 0 0 Ln: 37 Ch: 1 SPC LF

Error List

| Code | Description | Project | File | Line |
|-------|---|-----------------|---------|------|
| E0077 | this declaration has no storage class or type specifier | Cuda_Example2_2 | main.cu | 12 |
| E0065 | expected a ';' | Cuda_Example2_2 | main.cu | 12 |

9. Further down in the source file you can see the lines where the PC prints out the results in a for loop.



```
4 #define cudaCheckError() {
5     cudaError_t e=cudaGetLastError();
6     if(e!=cudaSuccess) {
7         printf("Cuda failure %s:%d: '%s'\n",__FILE__,__LINE__,cudaGetErrorString(e));
8         exit(EXIT_FAILURE);
9     }
10 }
11
12 global void kernel(int *a, int N) {
13     int i=blockIdx.x*blockDim.x+threadIdx.x;
14     if(i<N)
15         a[i]=i;
16 }
17
18 int main() {
19     int N=4097;
20     int threads=128;
21     int blocks=(N+threads-1)/threads;
22     int *a;
23
24     cudaMallocManaged(&a,N*sizeof(int));
25     kernel<<<blocks,threads>>>(a, N);
26     cudaDeviceSynchronize();
27
28     for(int i=0;i<10;i++)
29         printf("%d\n",a[i]);
30
31     cudaFree(a);
32
33     cudaCheckError();
34 }
```

100% 2 0 0 Ln: 1 Ch: 1 SPC LF

Output

Solution Explorer

Search Solution Explorer (Ctrl+Q)

Solution 'Cuda_Example2_2' (1 of 1 project)

- Cuda_Example2_2
 - References
 - External Dependencies
 - main.cu

Solution Explorer | Git Changes

Properties

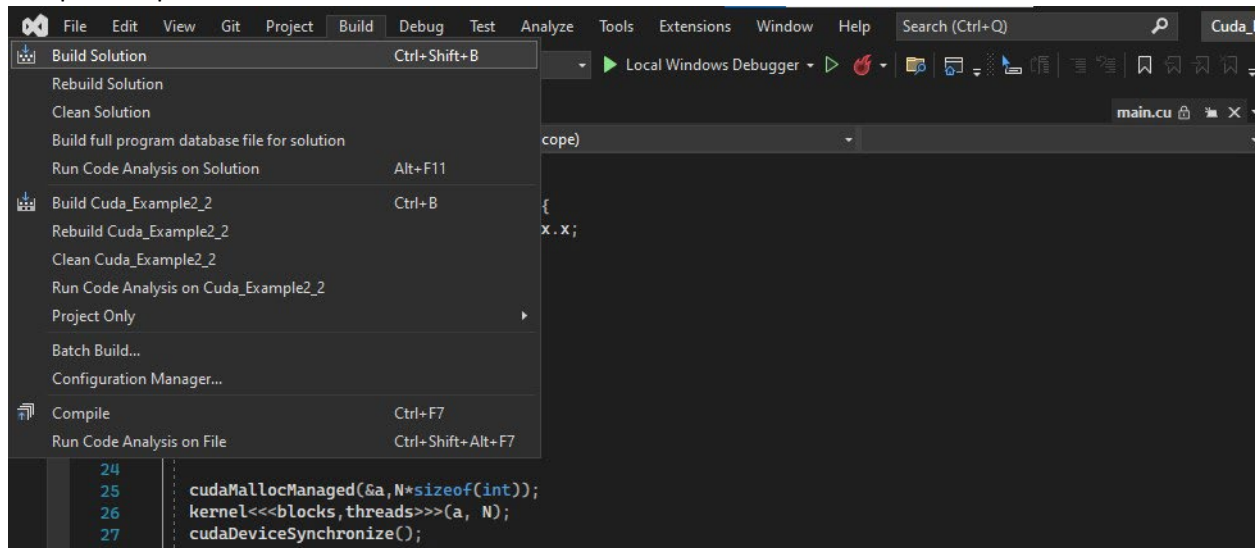
main.cu File Properties

| (Name) | main.cu |
|---------------------|-----------------------------|
| Content | False |
| File Type | Document |
| Full Path | C:\ELEC522\cavallar\CUDA_ex |
| Included In Project | True |
| Relative Path | main.cu |

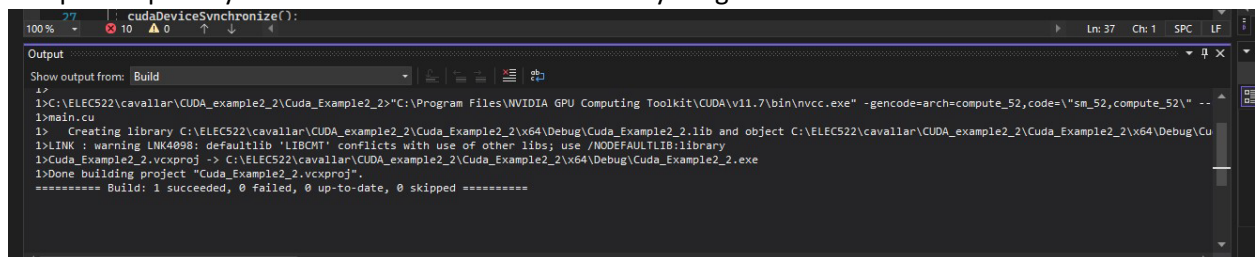
(Name)

Names the file object.

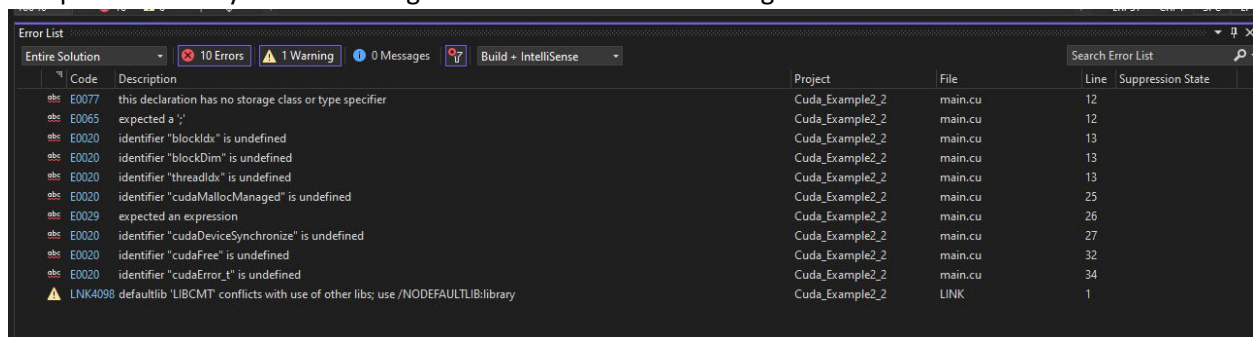
10. Now we are ready to compile the CUDA code. We will use the Build Solution to start the compilation process.



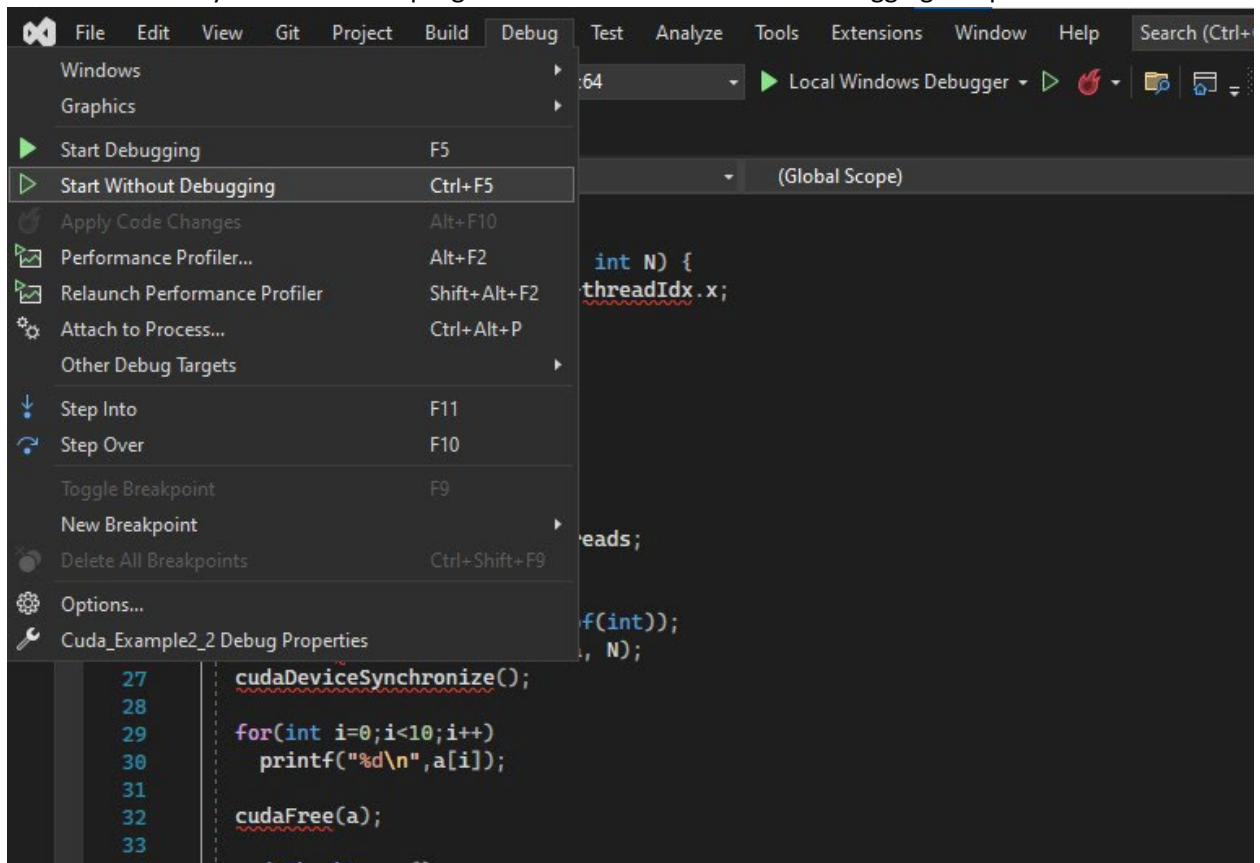
11. Here you can see the call to nvcc to do the actual compiling. The flag -gencode=arch=compute_52 should be OK for most modern boards as in the lab. If you have an older 700 series board you may need to change the CFLAGS to compute_50 which reflects the compute capability. This version of 52 will handle everything newer.



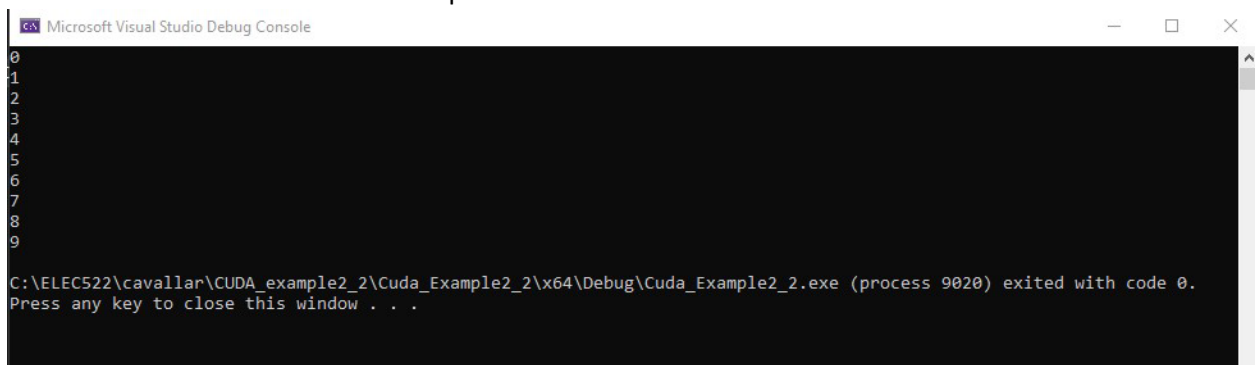
12. Again, there may be some warnings and errors reported by Visual Studio but this reflects the C++ parser not fully understanding CUDA extensions. It can be ignored.



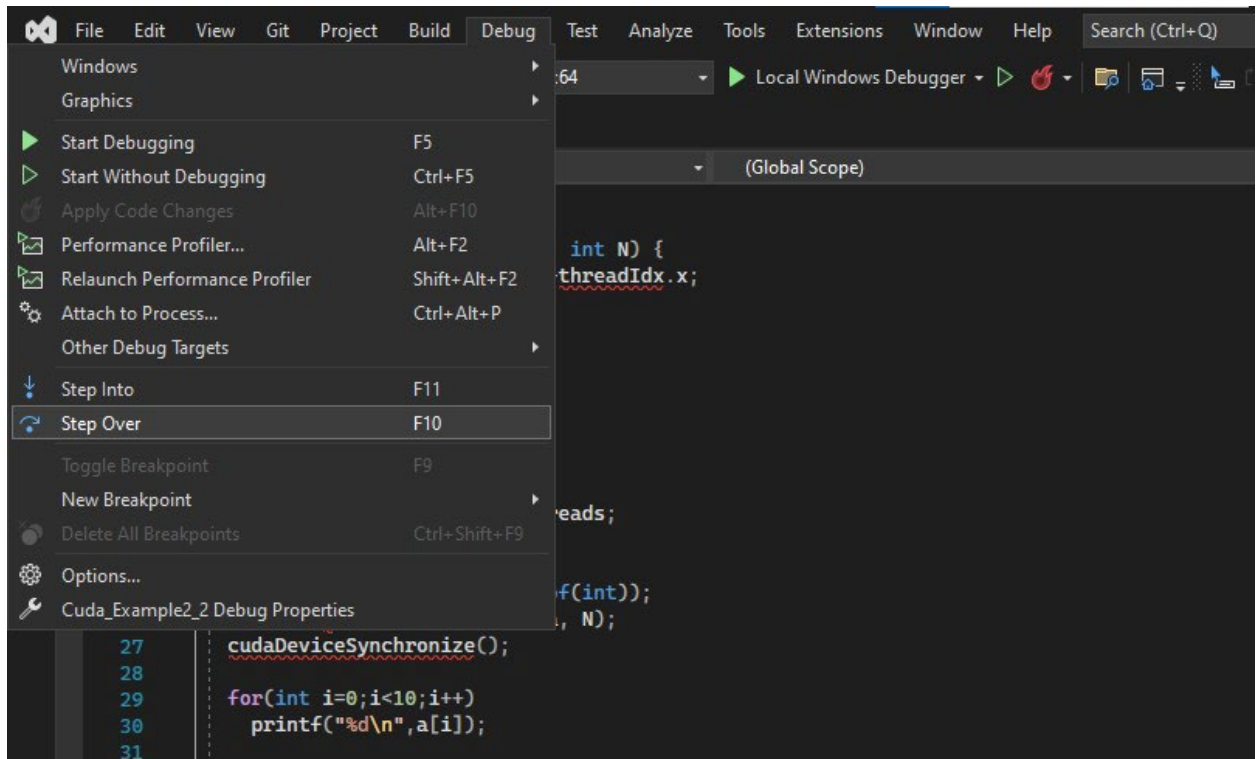
13. Now we are ready to execute the program with the “Start Without Debugging” step.



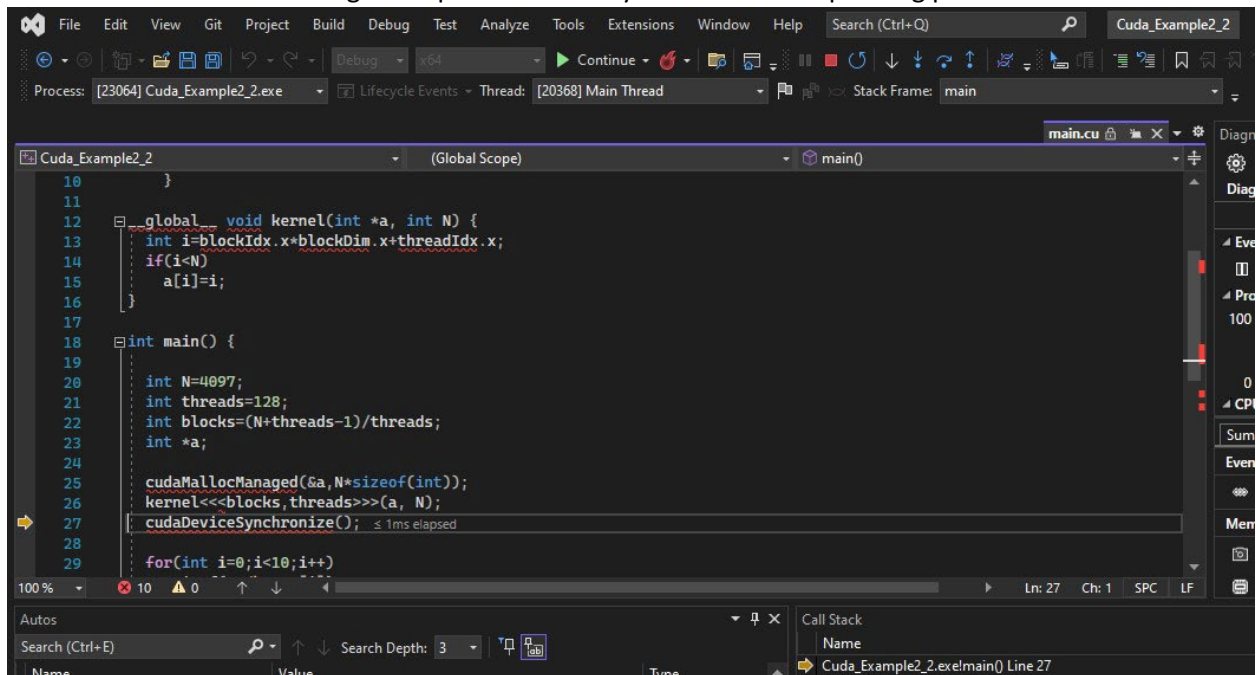
14. The results show the numbers 0 to 9 printed on the screen.



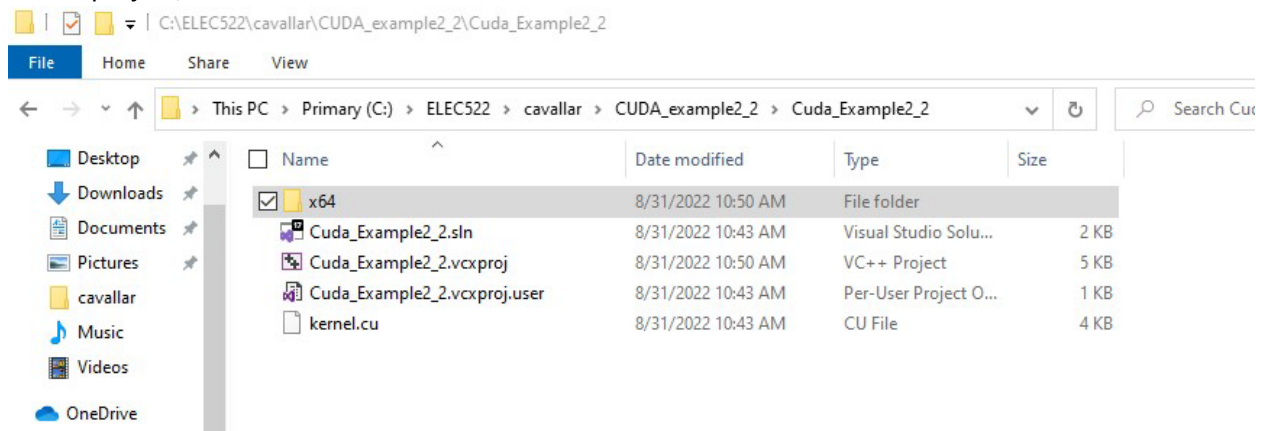
15. You can also single step either with “Step Into” or “Step Over” to see each line of code being executed.



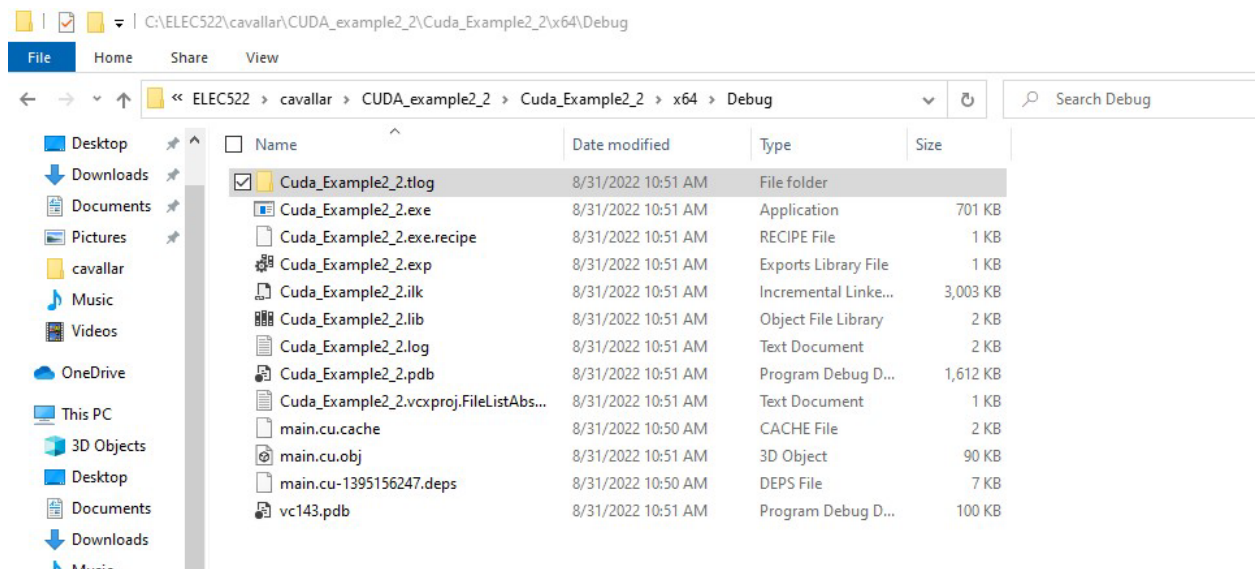
16. The Yellow Arrow in the margin will point to where you are in the step debug process.



17. After you are done with testing, you can look at the files in the folder and see that now a Visual Studio project, called a solution with a .sln file extension now exists.



18. The x64 folder contains all of the results of compilation. We have chosen the Debug mode which adds in more “label” information for debugging as opposed to the “Release” version that has the debugging information removed. If you view the files, you will see the Cuda_Example2_2.exe file that is the actual executable file that is run.



19. This completes building a new CUDA project and adding in your own CUDA code.