ReSYCLator: Transforming CUDA C++ source code into SYCL

Tobias Stauber
Peter Sommerlad
tobias.stauber@hsr.ch
peter.sommerlad@hsr.ch
IFS Institute for Software at FHO-HSR Hochschule für Technik
Rapperswil, Switzerland

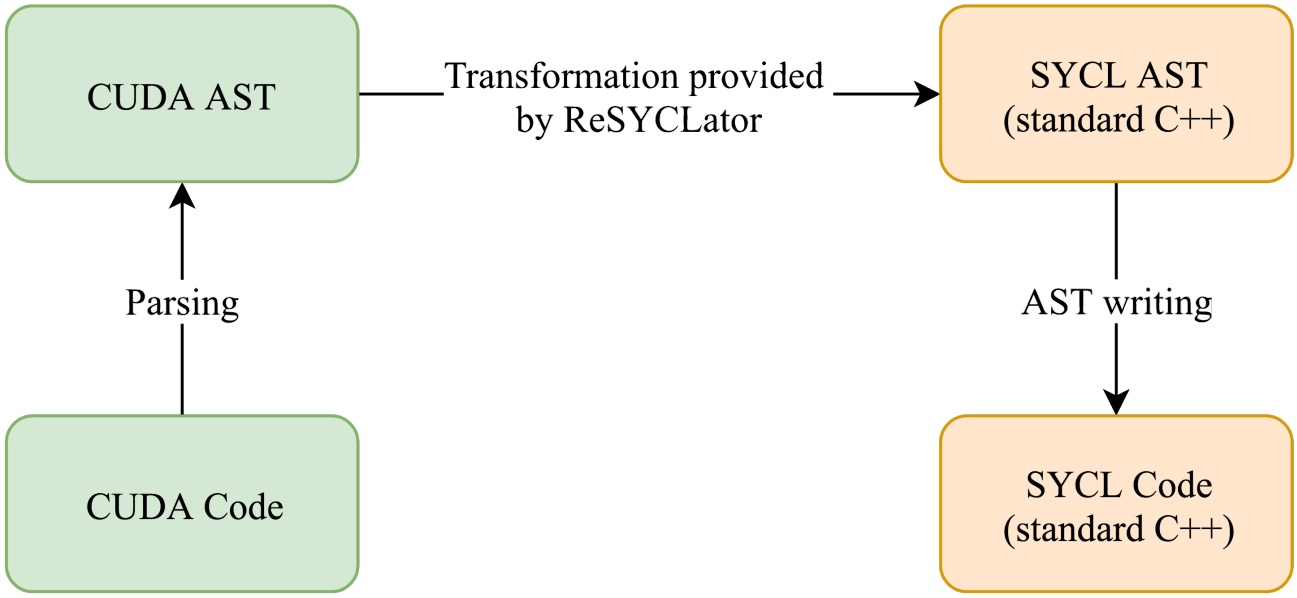

Figure 1: Transforming CUDA C++ code to SYCL using C++ AST Rewriting.

ABSTRACT

CUDA™ while very popular, is not as flexible with respect to target devices as OpenCL™. While parallel algorithm research might address problems first with a CUDA C++ solution, those results are not easily portable to a target not directly supported by CUDA. In contrast, a SYCL™ C++ solution can operate on the larger variety of platforms supported by OpenCL.

DHPCC++ '19, May 13–15, 2019, Boston, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ReSYCLator is a plug-in for Cevelop[2] an extension of Eclipse-CDT that bridges the gap between algorithm availability and portability, by providing automatic transformation of CUDA C++ code to SYCL C++. A first attempt basing the transformation on NVIDIA®'s Nsight™Eclipse CDT plug-in showed that Nsight™'s weak integration into CDT's static analysis and refactoring infrastructure is insufficient. Therefore, an own CUDA-C++ parser for Eclipse CDT was developed that is a sound platform for transformations from CUDA C++ programs to SYCL based on AST transformations.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; *Source code generation*; *Software maintenance tools.*

KEYWORDS

CUDA C++, SYCL, C++, Eclipse CDT, integrated development environment

ACM Reference Format:
Tobias Stauber and Peter Sommerlad. 2019. ReSYCLator: Transforming CUDA C++ source code into SYCL. In *Proceedings of Distributed & Heterogeneous Programming for C/C++ (DHPCC++) (DHPCC++ '19).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

NVIDIA®'s CUDA language is very popular but in addition to being bound to devices from a single vendor also introduces special syntax to C respectvely C++ for kernel function calls. This limits CUDA support in integrated development environments (IDEs) to what is provided by NVIDIA®, such as the Nsight™plug-in for Eclipse CDT. However, working with a single source language and its relatively long availability makes it still attractive for developers, such as in parallel algorithm research.

The vendor lock-in is a reason why some industries would like to to switch to more open solutions that allow more heterogeneous target hardware. OpenCL itself also has a long history, but its classic separate compilation model of kernels, e.g., as strings in the host language passed to the run-time compiler, is a limiting factor in IDE support. The quite recently developed SYCL circumvents the limitations of CUDA and OpenCL by integrating heterogeneous parallel computation in standard C++ syntax.

With the plethora of existing CUDA parallel algorithm implementations it would be great to ease their porting to SYCL to mitigate the vendor lock-in and to allow additional heterogeneous platforms, such as FPGAs to run them.

1.1 Institute for Software's history in Refactoring

Our institute has a long history in implementing refactoring tools for Eclipse-based IDEs. We started out more than a decade ago to implement refactoring support for Eclipse CDT, such as AST-rewriting [4], heuristics for keeping the transformed code as close to its original representation, such as keeping comments around[7], and also worked on source-to-source transformation of sequential C++ to parallel C++ including generating C++ source code targeting FPGAs in the EU-FP7 REPARA project [5] [3]. The result of our work on better supporting C++ modernization is currently made available through the free-to-use IDE Cevelop[2]

2 CUDA SYNTAX TO BE TRANSFORMED

The underlying concepts of CUDA as well as OpenCL/SYCL are not inherently different. That makes transformation feasible. However, manual transformation can be very tedious. Here we give a brief overview of key elements of CUDA syntax that will be transformed. Unfortunately, at the time of this writing, no formal specification of CUDA syntax is available in contrast to standard C++[6], so what is described here is derived from the CUDA programming guide[1] that presents the NVCC CUDA dialect.

2.1 Marking CUDA Kernels

Kernel functions in CUDA are marked with the following specifiers:

- **`__host__`** function is executable on the host CPU (redundant, unless combined with `__device__`
- **`__global__`** kernel function that is executable on the GPU device and can be called from the host CPU with the special call syntax
- **`__device__`** kernel function executable and callable only on the device, unless combined with `__host__`

These identifiers are implemented as macros, which makes detecting them in the parsed AST already a challenge. Similar specifiers are used for memory space designation (`__device__ __constant__ __shared__`).

2.2 Invoking Kernels

For calling a kernel, there is a special syntax. This syntax consists of <<< to start the launch-parameter list, and >>> to close it (Listing 1).

```
kernelname<<<grid_dimensions,
    block_dimensions, bytes_of_shared_mem,
    stream>>>(args);
```

Listing 1: Special CUDA kernel invocation syntax

The first mandatory argument, `grid_dimensions`, is of type `int`, `uint3`, or `dim3`, and defines how many blocks are contained in the grid in each dimension (x, y, z).

The second argument, `block_dimensions`, is also mandatory, and of type `int`, `uint3`, or `dim3`. It is used to define how many threads are there per dimension in a block.

The number of bytes of shared memory allocated for each block in the grid are passed as an optional argument `size_t` (`bytes_of_shared_mem`).

The optional argument `stream` tells the CUDA on which stream this kernel should be run. This value must be of type `cudaStream_t` and defaults to the default-stream if omitted.

The concept of CUDA streams is not handled yet, but it can be transformed to SYCL queues.

2.3 Special Indicdes

In a CUDA kernel, each running thread has a set of built-in variables allowing to calculate the current block's position in the grid (`blockIdx`) and a thread's position in the block (`threadIdx`). Both dimensions are given by the special CUDA arguments of a kernel call. The member selectors `.x`, `.y`, `.z` allow indexing relative to each thread running a kernel.

3 TRANSFORMING CUDA KERNELS TO SYCL KERNELS

The information on CUDA kernel functions, memory management operations (omitted above), and kernel implementations needs to be detected in CUDA source code and transformed to corresponding SYCL mechanisms in C++. Most of the remaining plain C++ code can be taken literally.

3.1 Adjusting kernel function signatures

A first step in the transformation is to find CUDA kernel function definitions and declarations, i.e., by looking for those that have the attribute global attached, because the `__global__` macro is

expanded to (__attribute__((global))) in the case of the GCC compiler as in Listing 2. Using the Microsoft Visual Studio Compiler toolchain the macro would be expanded to __declspec(__global__).

```
__global__ void matrixMultiplicationKernel(
    float *A,
    float *B,
    float *C,
    int & N);
```

Listing 2: Declaring a CUDA kernel

In the C++ AST of Eclipse CDT macros are expanded, while also the original source code with the unexpanded macro is referred by it. Macros are one of the aspects that makes AST-based code transformations tricky in C++. However, the AST nodes representing the CUDA kernel specifier get removed. Then the CUDA parameter declarations need to be extended to include the SYCL-specific nd_item dimension parameter as their first parameter. The dimension template argument of nd_item is introduced by transforming the function declaration into a template function with an integer template parameter. As a remaining step the pointer parameters of a typical CUDA kernel, need to be mapped to SYCL accessors (Listing 4) or SYCL global pointers(Listing 3). The latter is used in the transformation, because it allows a more direct mapping of the kernel function body. However, in the future a SYCL-specific refactoring from global pointer parameters to accessors could be provided using Cevelop's refactoring infrastructure. Such a refactoring could be beneficial also for existing or manually transformed SYCL code.

```
using namespace cl::sycl;

template<int dimensions>
void matrixMultiplicationKernel(
nd_item<dimensions> item,
global_ptr<float> A,
global_ptr<float> B,
global_ptr<float> C,
global_ptr<int> N);
```

Listing 3: SYCL declaration with global pointers

```
//template aliases provided automatically
template<cl::sycl::access::mode mode,
  int dim>
using Accessor = cl::sycl::accessor<
  float, dim, mode,
  cl::sycl::access::target::global_buffer>;
template<int dim>
using ReadAccessor = Accessor<
  cl::sycl::access::mode::read,dim>;
template<int dim>
using WriteAccessor = Accessor<
  cl::sycl::access::mode::write, dim>;

template<int dim>
void matrixMultiplicationKernel(
  nd_item<dim> item,
  ReadAccessor<dim> A,
  ReadAccessor<dim> B,
  WriteAccessor<dim> C,
  int N);
```

Listing 4: SYCL declaration with accessors

3.2 Transforming kernel function bodies

After the kernel signature has been adjusted to SYCL, the actual kernel code needs to be transformed. One major substitution to take place, is to translate the CUDA-specific index variables (threadIdx, blockIdx) and dimension variables (blockDim, gridDim) to corresponding accesses via the SYCL nd_item parameter. Each CUDA index and dimension variable provides three member accessors (x, y, z) that map to SYCL dimension indices 0, 1, 2 respectively. For the rest of the mapping see Table **??** where DIM denotes member and its corresponding index respectively.

CUDA variable	SYCL nd_item call
threadIdx.DIM	item.get_local_id(DIM)
blockIdx.DIM	item.get_group(DIM)
blockDim.DIM	item.get_local_range(DIM)
gridDim.DIM	item.get_local_id(DIM)

Table 1: Mapping CUDA variables to SYCL nd_item member functions

Taking the original CUDA implementation from Listing 5 will result in the following transformed SYCL kernel function in Listing 6. Note that, due to a SYCL compiler warning, array index operator uses on pointers was translated to explicit pointer arithmetic. In the future this cludge might no longer be required to produce valid code.

```
__global__ void matrixMultiplicationKernel(
    float *A, float *B, float *C, int N) {
    int ROW = blockIdx.y * blockDim.y +
        threadIdx.y;
    int COL = blockIdx.x * blockDim.x +
        threadIdx.x;

    float tmpSum = 0;
    if (ROW < N && COL < N) {
        for (int i = 0; i < N; i++) {
            tmpSum += A[ROW * N + i] *
                      B[i * N + COL];
        }
    }
    C[ROW * N + COL] = tmpSum;
}
```

Listing 5: CUDA matrix multiplication kernel

```
template<int dim>
void matrixMultiplicationKernel(
  nd_item<dim> item,
  global_ptr<float> A,
  global_ptr<float> B,
  global_ptr<float> C,
  global_ptr<int> N)
{
    int ROW = item.get_group(1) *
              item.get_local_range(1)
              + item.get_local_id(1);
    int COL = item.get_group(0) *
              item.get_local_range(0)
              + item.get_local_id(0);

    float tmpSum = 0;
    if (ROW < N && COL < N) {
        for (int i = 0; i < N; i++) {
            tmpSum += *(A + ROW * N + i) *
                      *(B + i * N + COL);
        }
    }
    *(C + ROW * N + COL) = tmpSum;
}
```

Listing 6: Transformed SYCL kernel function

4 TRANSFORMING CUDA KERNEL CALL SITE

A typical call site of a CUDA kernel consists of the following parts:

- pointer definitions for memory blocks to be allocated
- preparation of memory through cudaMalloc calls
- initializing kernel input data
- preparing kerne grid dimensions depending on data size and layout, if not fixed
- the CUDA kernel call (see Listing 1)
- synchronizing with the device
- obtaining the results
- freeing the memory

The example program in Listing 10 shows this and compares a CPU matrix multiplication result with the GPU results.

All these parts have to be adapted to the concepts and syntax of SYCL. Fortunately, some of the parts can be eliminated in total, such as the explicit freeing of memory, because SYCL employs the C++ scope-based resource management idiom with automatic clean-up when leaving a scope.

4.1 SYCL memory buffer set up

As one can see in Listing 10 a typical CUDA program needs to call allocation and deallocation functions symmetrically to provide memory for device use. This is not considered a relevant style in C++, where the RAII pattern(resource-acquisition is initialization)–also called scope-bound resource management (SBRM) provides a cleaner and less error-prone mechanism. So the definition of pointers and `cudaMalloc` and `cudaFree` calls are replaced by SYCL buffers, that automatically manage memory allocation, transfer to and from, and synchronization with the computing device.

As an example, for the usage of one of the input matrices (A) the lines declaring the pointer, allocating device memory, synchronizing results as well as freeing the memory again as the excerpt from Listing 10 in Listing 7, get replaced by the corresponding code that is synthesized from the CUDA code in Listing 8. Note, that in contrast to cudaMalloc call that allocates bytes, the SYCL buffer variable definition automatically takes the size of the element type into account. The refactoring detects if the expression can just drop the `sizeof` expression from a multiplication. In case of a more complex, or simpler size computation the division by `sizeof(elementtype)` is explicitly introduced.

```
/* Declare device memory pointers */
float *d_A;
/* Allocate CUDA memory */
cudaMallocManaged(&d_A, SIZE * sizeof(
    float));
/* Synchronize device and host memory */
cudaDeviceSynchronize();
/* Free the CUDA memory */
cudaFree(d_A);
```

Listing 7: Setting up and cleaning up CUDA input data.

```
/* Replacement statement with simplified
    size expression*/
cl::sycl::buffer<float> d_A(SIZE);
```

Listing 8: SYCL buffer declaration replaces all lines in Listing 7

4.2 SYCL memory accessors from CUDA pointer accesses

The example code in Listing 10 contains nested loops initializing the input matrices. For simplicity, this loop does two dimensional index transformation manually in the allocated area using the pointers. Since SYCL accesses all buffer memory through SYCL accessors for such an access corresponding accessor objects have to be created. It is important that these accessor objects are defined in a as local as possible scope, because their lifetime is used to synchronize between host and kernel access to the underlying buffer. Because the latter is quite expensive, it is also important that the accessors to a SYCL buffer are not defined within close loops. Therefore, the transformation will introduce a scope surrounding the initialization loops and defines two accessor variables in that newly introduced scope. The accessor variables' names are taken from the buffer name and prefixed with "`acc_`". This is a situation where AST-based transformations shine, because the AST subtree consisting of the usages is put into the newly introduced compound statement. You can also see the comment-retainment heuristic in action from [7], because the comment associated with the outer for loop is also attached in front of the new compound statement.

Furthermore, the index accesses through the original pointer variables, e.g., `d_A`, need to be adjusted to use the newly introduced

accessor variable acc_d_A. The transformed code for the loops filling matrices A and B from Listing 10 is shown in Listing 9.

```
/* Fill values into A and B */
{
    auto acc_d_B = d_B.get_access<cl::sycl::
        access::mode::read_write>();
    auto acc_d_A = d_A.get_access<cl::sycl::
        access::mode::read_write>();
    /* Fill values into A and B */
    for (int i { 0 }; i < N; i++) {
        for (int j { 0 }; j < N; j++) {
            acc_d_B[N * i + j] = cos(j);
            acc_d_A[j + N * i] = sin(i);
        }
    }
}
```

Listing 9: Introducing scope for SYCL accessors

The underlying scope-introduction algorithm employs slicing and scope matching to find or introduce a minimal scope for all CUDA-pointer based accesses. This avoids blocking SYCL buffer access from the kernel, because an accessor is still alive. At the end of its lifetime towards the end of the scope, a SYCL accessor releases its lock on the memory region managed by its associated SYCL buffer. From the example in Listing 10 a similar transformation would happen for the section commented with "Compare the results".

4.3 Transforming a CUDA kernel call to a SYCL compute-queue submission

In contrast to the relatively simple CUDA kernel call syntax, activating a kernel in SYCL is a bit more elaborated, because it requires introducing a compute queue the kernel is submitted with. The special arguments to a CUDA kernel call specifying the underlying grid and block dimensions that are computed

5 TOOLING ARCHITECTURE

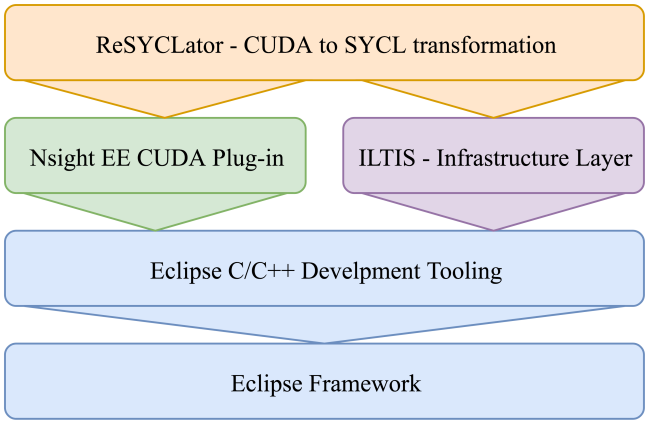

Figure 2: Architectural overview initial prototype.

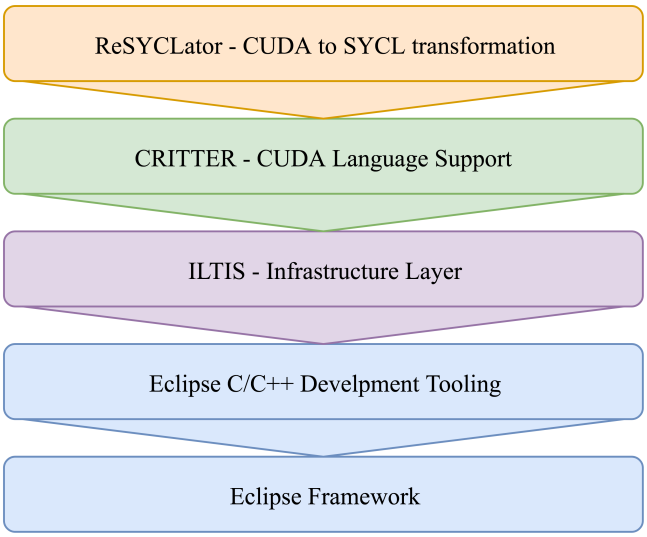

Figure 3: Architectural overview after own CUDA parser implementation.

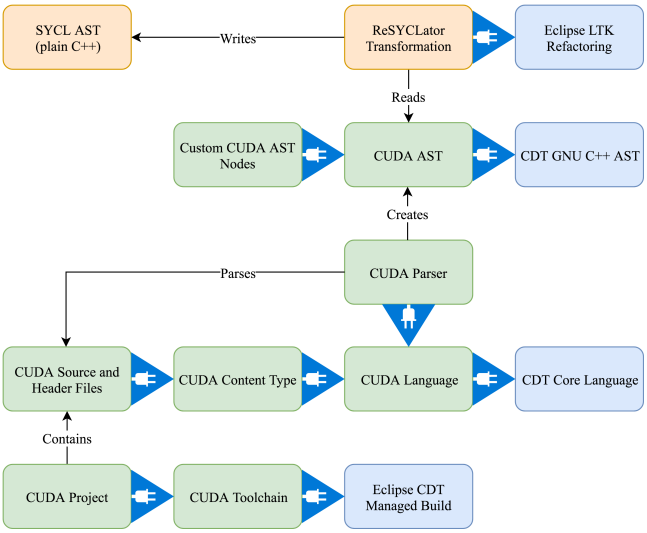

Figure 4: Plug-in dependencies of CUDA C++ parser SYCL transformation. Triangles mark plug-in extensions.

6 OUTLOOK AND FUTURE WORK

passing accessors instead of global_ptr to kernels

multiple dimensions handling instead of one dimensional explicit mapping.

ACKNOWLEDGMENTS

To our Codeplay friends who inspired work on CUDA to SYCL transformation and their support during Tobias' master project work.

REFERENCES

[1] 2018. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] 2019. https://cevelop.com

[3] Silvano Brugnoni, Thomas Corbat, Peter Sommerlad, Toni Suter, Jens Korinth, David de la Chevallerie, and Andreas Koch. 2016. Automated Generation of Reconfigurable Systems-on-Chip by Interactive Code Transformations for High-Level Synthesis. In *FSP 2016; Third International Workshop on FPGAs for Software Programmers; Proceedings of*. VDE, 1–11.

[4] Emanuel Graf, Guido Zgraggen, and Peter Sommerlad. 2007. Refactoring support for the C++ development tooling. *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07* (2007). https://doi.org/10.1145/1297846.1297885

[5] G. Gyimesi, D. Bán, I. Siket, R. Ferenc, S. Brugnoni, T. Corbat, P. Sommerlad, and T. Suter. 2016. Enforcing Techniques and Transformation of C/C++ Source Code to Heterogeneous Hardware. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. 1173–1180. https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0180

[6] ISO. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). 1605 pages. https://www.iso.org/standard/68564.html

[7] Peter Sommerlad, Guido Zgraggen, Thomas Corbat, and Lukas Felber. 2008. Retaining comments when refactoring code. *Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA Companion '08* (2008). https://doi.org/10.1145/1449814.1449817

A LONGER CODE EXAMPLES

```
int main() {
   size_t N = 16;
   /* Matrix dimension */
   size_t SIZE = N * N;

   /* Declare device memory pointers */
   float *d_A;
   float *d_B;
   float *d_C;

   /* Allocate CUDA memory */
   cudaMallocManaged(&d_A, SIZE * sizeof(
       float));
   cudaMallocManaged(&d_B, SIZE * sizeof(
       float));
   cudaMallocManaged(&d_C, SIZE * sizeof(
       float));

   /* Fill values into A and B */
   for (int i { 0 }; i < N; i++) {
      for (int j { 0 }; j < N; j++) {
         d_B[N * i + j] = cos(j);
         d_A[j + N * i] = sin(i);
      }
   }

   /* Define grid and block dimensions */
   dim3 block_dim;
   dim3 grid_dim;

   if (N * N > 512) {
      block_dim = {512, 512};
      grid_dim = {(N + 512 - 1) / 512, (N +
          512 - 1) / 512};
   } else {
      block_dim = {N, N};
      grid_dim = {1, 1};
   }

   /* Invoke kernel */
   matrixMultiplicationKernel<<<grid_dim,
       block_dim>>>(d_A, d_B, d_C, N);

   /* Synchronize device and host memory */
   cudaDeviceSynchronize();

   float *cpu_C;
   cpu_C = new float[SIZE];

   /* Run matrix multiplication on the CPU
       for reference */
   float sum;
   for (int row { 0 }; row < N; row++) {
      for (int col { 0 }; col < N; col++) {
         sum = 0.f;
         for (int n { 0 }; n < N; n++) {
            sum += d_A[row * N + n] * d_B[n
                * N + col];
         }
         cpu_C[row * N + col] = sum;
      }
   }

   double err { 0 };
   /* Compare the results */
   for (int ROW { 0 }; ROW < N; ROW++) {
      for (int COL { 0 }; COL < N; COL++) {
         err += cpu_C[ROW * N + COL] - d_C[
             ROW * N + COL];
      }
   }

   std::cout << "Error: " << err << std::
       endl;

   /* Free the CUDA memory */
   cudaFree(d_A);
   cudaFree(d_B);
   cudaFree(d_C);

   return 0;
}
```

Listing 10: main() calling CUDA matrix multiplication kernel