Parsing CUDA® for Transformation to SYCL™ in an IDE

Cevelop – Your C++ deserves It

IFS INSTITUTE FOR SOFTWARE

Tobias Stauber

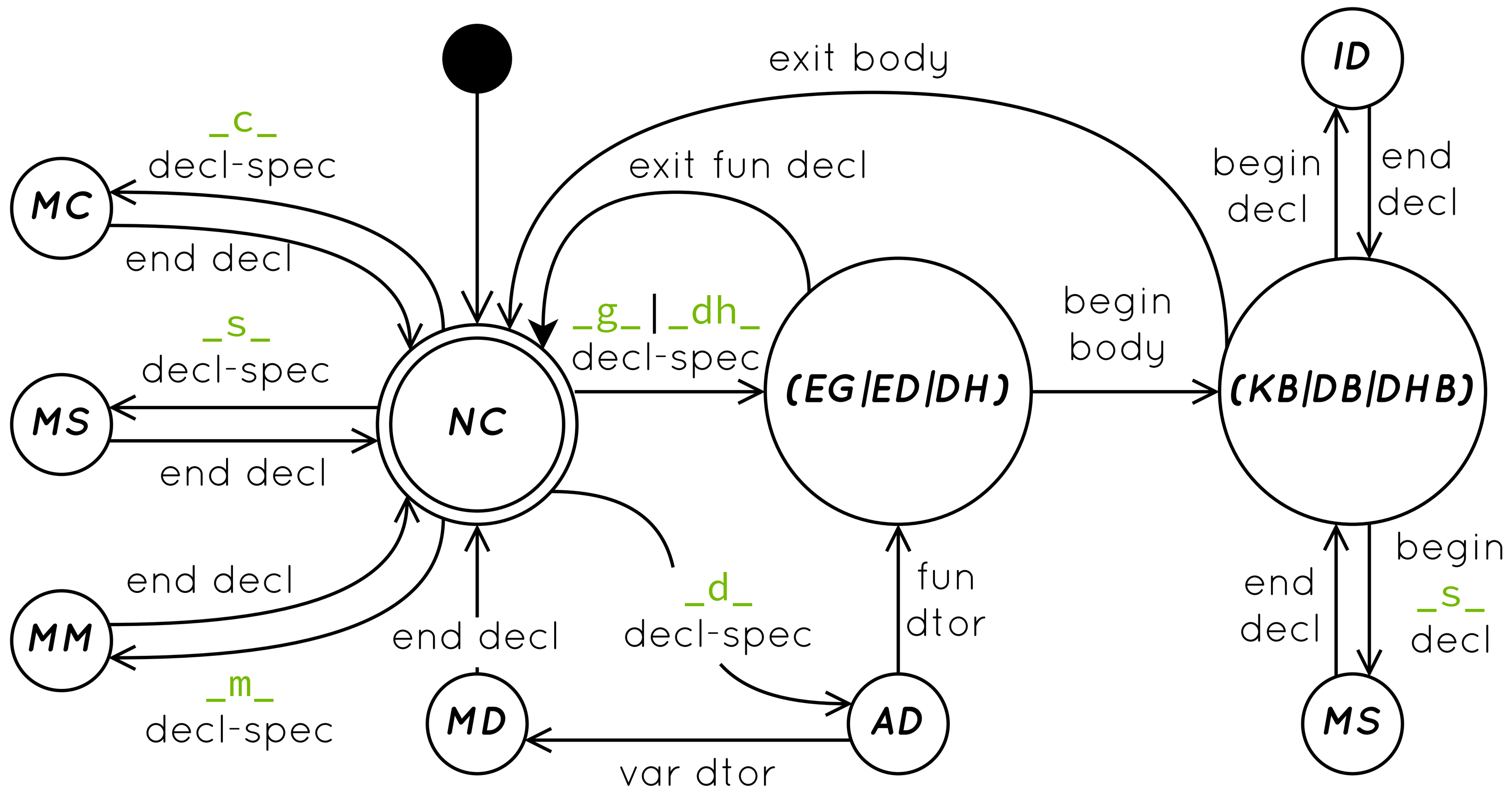
AST Augmentation for CUDA to SYCL Transformation

In order to facilitate further support for the transformation from CUDA to SYCL, the parsed AST is enhanced with information about its variables' Memory Space (*MSS*) and its functions' Execution Space (*ESS*). Thus making it possible to quickly evaluate in what scope a name can be resolved. For this purpose an additional state machine was embedded into the CUDA parser. By means of this state machine, targets which are of heightened interest for the transformation, such as device functions or field references to CUDA built-in variables, are recognized as such during parsing.

CUDA Parser States for Gathering Information About MSS & ESS

- *NC* = Normal Context (__host__ or none)
- *MC* = Constant Memory (__constant__)
- *MS* = Shared Memory (__shared__)
- *MM* = Managed Memoyr (__managed__)
- *MD* = Device Memory (__device__)
- *AD* = Ambiguous __device__ tag
- *EG* = Gloabl Execution Space (__global__)
- *ED* = Device Execution Space (__device__)
- *DH* = Device and Host Execution Space (__host__ and __device__)
- *KB* = In Body of a Kernel Function
- *DB* = In Body of a Device Function
- *DHB* = In Body of a DH Function
- *ID* = Implicit Device Memory

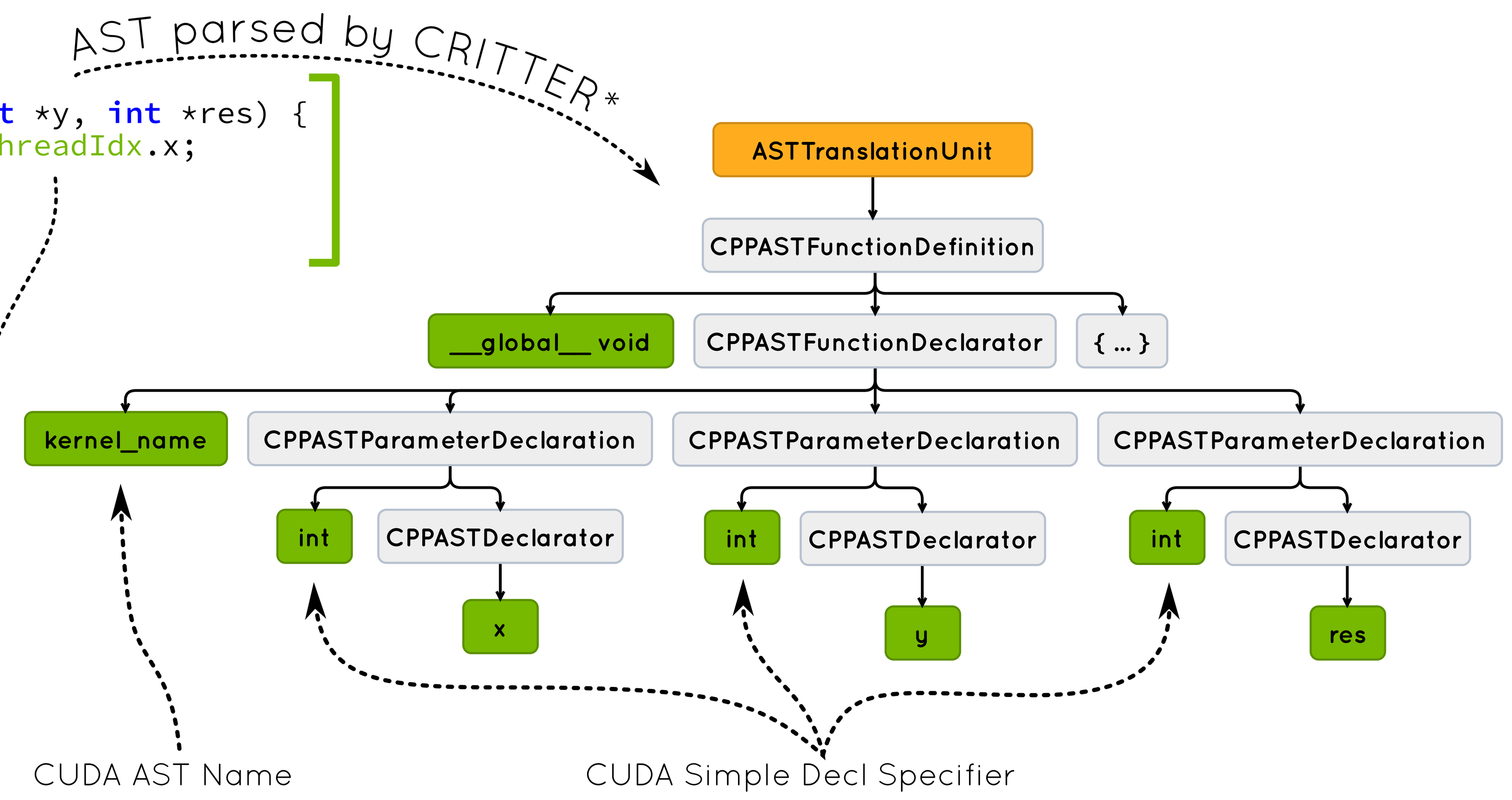
decl-spec = C++ Declaration Specifier
decl = C++ Declaration
dtor = C++ Declarator



Example for an Automated Transformation of a Kernel

```
__global__ void kernel_name(int *x, int *y, int *res) {
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  res[id] = x[id] * y[id];
}
```

Original CUDA code (before transformation)




```
#include <CL/sycl.hpp>

template<int dim>
void kernel_name(nd_item<dim> item, global_ptr<int> x,
                 global_ptr<int> y, global_ptr<int> res) {
  int id = item.get_group(0) * item.get_local_range(0) +
           item.get_local_id(0);
  res[id] = x[id] * y[id];
}
```

Resulting SYCL code (after transformation)

From CUDA to SYCL

Memory Management

```
float *array;
cudaMallocManaged(&array, 1024 * sizeof(float));
for (int i = 0; i < 1024; i++) {
  array[i] = 1.0f;
}
cudaFree(array);
```

```
using namespace cl::sycl;
buffer<float> array(1024);
{
  auto acc_array = array.get_access<access::mode::read_write>();
  for (int i = 0; i < 1024; i++) {
    acc_array[i] = 1.0f;
  }
}
```

The CUDA to SYCL transformation keeps track of the CUDA memory management calls and tries to deduce which pointers' memory is allocated by the CUDA runtime.

Kernel Call

```
kernel_name<<<1,1024>>>(x, y, res);
```

The CUDA parser creates a new AST node for the kernel-call expression. This node can be automatically transformed into the equivalent SYCL construct shown on the right. (The "using-directives" are only used for clarification)

```
{
  using namespace cl::sycl;
  gpu_selector selector { };
  device selectedDevice { selector };
  queue compute_queue { selectedDevice };
  compute_queue.submit(
    [&](handler& cgh) {
      cgh.parallel_for<class kernel_name_functor>(nd_range<1> {
          range<1> {1024}, range<1> {1024}}, [=](nd_item<> item) {
        kernel_name(item, x, y, res);
      }
    );
  });
}
```

Future Work

Transformation of Shared Memory

Currently, shared memory can not be converted. In the transformation's next iteration support for shared and constant memory will be added. CUDA shared memory will be mapped to local memory in SYCL. For this, an accessor will be passed to the kernel function. In the same step support for transforming variables declared in the host-code but using device memory can be added.

Support for Error Handling

In most of the analyzed CUDA code, macros are used for evaluating the error codes returned by the calls to the CUDA runtime API. As the SYCL specification declares exceptions that are to be thrown if something went wrong, the CUDA error handling actions should be implemented for the corresponding SYCL exceptions. This could be done by wrapping the code, into which the macro expands, in a "try-catch" statement.

Support for Non-Managed CUDA Memory

While the CUDA managed memory can be transformed directly into a SYCL buffer, the manually handled memory uses two pointers, one for the host copy, and one for the device's copy. Those have to be merged into a single SYCL buffer. Thereby, the calls to the memory movement functions should be analyzed, and the results used to deduce which accessor-mode is used best for the corresponding SYCL accessor.