# Array Parameters of SYCL Kernels

## Introduction

This document describes the changes to support passing arrays to SYCL kernels and special treatment of Accessor arrays. The following cases are handled:

1. arrays of standard-layout type as top-level arguments
2. arrays of Accessors as top-level arguments
3. arrays of accessors within structs that are top-level arguments

The first few sections describe the current design. The last three sections describe the design to support 1. to 3. above. The implementation of this design is confined to three functions in the file SemaSYCL.cpp.

## A SYCL Kernel

The SYCL constructs single\_task, parallel\_for, and parallel\_for\_work\_group each take a function object or a lambda function as one of their arguments. The code within the function object or lambda function is executed on the device. Code generation for SYCL is based on the internal representation of the function/lambda object.

## SYCL Kernel Code Generation

Consider a source code example that captures an int, a struct and an accessor by value:

constexpr size\_t c\_num\_items = 10;

range<1> num\_items{c\_num\_items}; // range<1>(num\_items)

int main()

{

int output[c\_num\_items];

queue myQueue;

int i = 55;

struct S {

int m;

} s = { 66 };

auto outBuf = buffer<int, 1>(&output[0], num\_items);

myQueue.submit([&](handler &cgh) {

auto outAcc = outBuf.get\_access<access::mode::write>(cgh);

cgh.parallel\_for<class Worker>(num\_items, [=](cl::sycl::id<1> index) {

outAcc[index] = i + s.m;

});

});

return 0;

}

The input to the code generation routines is a function object that represents the kernel. In pseudo-code:

struct Capture {

Accessor outAcc;

int i;

struct S s;

() {

outAcc[index] = i + s.m;

}

}

On the CPU a call to such a lambda function would look like this:

()(struct Capture\* this);

When offloading the kernel to a device, the lambda function cannot be directly called with a capture object address. Instead, the code generated for the device is in the form of a “kernel caller” and a “kernel callee”. The *callee* looks very similar to a lambda function on the CPU. The *caller* requires special code generation. It is designed to receive the lambda capture object in pieces, assemble the pieces into the original lambda capture object and then call the callee:

spir\_kernel void caller(

int AccDim, // arg1 of Accessor init function

range<1> AccR1, // arg2 of Accessor init function

range<1> AccR2, // arg3 of Accessor init function

id<1> I, // arg4 of Accessor init function

int i,

struct S s

)

{

// Local capture object

struct Capture local;

// Reassemble capture object from parts

local.i = i;

local.s = s;

// Call accessor’s init function

Accessor::init(&local.outAcc, AccDim, AccR1, AccR2, I);

// Call the kernel body

callee(&local, id<1> wi);

}

spir\_func void callee(struct Capture\* this, id<1> wi)

{

}

As may be observed from the example above, standard-layout lambda capture components are passed by value to the device as separate parameters. This includes scalars, pointers, and standard-layout structs. Certain SYCL struct types that are not standard-layout, such as Accessors and Samplers, are treated specially. The arguments to their init functions are passed as separate parameters and used within the kernel caller function to initialize Accessors/Samplers on the device by calling their init functions using the received arguments.

There is one other aspect of code generation. An “integration header” is generated to be used during host compilation. This header file contains entries for each kernel. Among the items it defines is a table of sizes and offsets of the kernel parameters. For the source example above the integration header contains the following snippet:

// array representing signatures of all kernels defined in the

// corresponding source

static constexpr

const kernel\_param\_desc\_t kernel\_signatures[] = {

//--- \_ZTSZZ4mainENKUlRN2cl4sycl7handlerEE19->18clES2\_E6Worker

{ kernel\_param\_kind\_t::kind\_accessor, 4062, 0 },

{ kernel\_param\_kind\_t::kind\_std\_layout, 4, 32 },

{ kernel\_param\_kind\_t::kind\_std\_layout, 4, 36 },

};

Each entry in the kernel\_signatures table contains three values: 1) an encoding of the type of capture object member, 2) a field that encodes additional properties, and 3) an offset within a block of memory where the value of that kernel argument is placed.

***The previous sections described how kernel arguments are handled today. The next three sections describe support for arrays.***

## Extension 1: Kernel Arguments that are Standard-Layout Arrays

The motivation for this extension is this: On the CPU, a lambda function is allowed to access an element of an array defined outside the lambda. The implementation captures the entire array by value. A user would naturally expect this to work in SYCL as well. However, the current implementation does not allow arrays referenced within kernels, that are implicitly captured by value.

This feature is added by a proposed change to SemaSYCL.cpp. As described earlier, each captured variable is passed by value to the kernel caller. Simply allowing the existing scheme to let arrays through would result in a function parameter of array type. This is not supported in C++. Therefore, the array needing capture is wrapped in a struct for the purposes of passing to the device. Once received on the device within its wrapper, the array is copied into the local capture object. All references to the array within the kernel body are directed to the non-wrapped array which is a member of the local capture object.

Source code fragment:

int array[100];

auto outBuf = buffer<int, 1>(&output[0], num\_items);

myQueue.submit([&](handler &cgh) {

auto outAcc = outBuf.get\_access<access::mode::write>(cgh);

cgh.parallel\_for<class Worker>(num\_items, [=](cl::sycl::id<1> index) {

outAcc[index] = array[index.get(0)];

});

});

Integration header produced:

static constexpr

const kernel\_param\_desc\_t kernel\_signatures[] = {

//--- \_ZTSZZ4mainENKUlRN2cl4sycl7handlerEE16->18clES2\_E6Worker

{ kernel\_param\_kind\_t::kind\_accessor, 4062, 0 },

{ kernel\_param\_kind\_t::kind\_std\_layout, 400, 32 },

};

The changes to device code made to support this extension, in pseudo-code:

struct Capture {

Accessor outAcc;

int array[100];

() {

// Body

}

}

struct wrapper {

int array[100];

};

spir\_kernel void caller(

int AccDim, // arg1 of Accessor init function

range<1> AccR1, // arg2 of Accessor init function

range<1> AccR2, // arg3 of Accessor init function

id<1> I, // arg4 of Accessor init function

struct wrapper w\_s // Pass the array wrapped in a struct

)

{

// Local capture object

struct Capture local;

// Reassemble capture object from parts

// Initialize array using existing clang Initialization mechanisms

local.array = w\_s;

// Call accessor’s init function

Accessor::init(&local.outAcc, AccDim, AccR1, AccR2, I);

callee(&local, id<1> wi);

}

The sharp-eyed reviewer of SemaSYCL.cpp will notice that the array is actually double-wrapped in structs. This was done simply to preserve the interface to an existing function within the code which processes each kernel caller parameter as a capture object *member*. By wrapping the array twice, the wrapped array appears as a member of a struct and meets the requirements of the existing code. This could be changed but would lead to modifications in parts of the code unrelated to this extension.

## Extension 2: Kernel Arguments that are Arrays of Accessors

Arrays of accessors are supported in a manner similar to that of a plain Accessor. For each accessor array element, the four values required to call its init function are passed as separate arguments to the kernel. Reassembly within the kernel caller is serialized by accessor array element.

Source code fragment:

myQueue.submit([&](handler &cgh) {

using Accessor =

accessor<int, 1, access::mode::read, access::target::global\_buffer>;

Accessor inAcc[2] = {in\_buffer1.get\_access<access::mode::read>(cgh),

in\_buffer2.get\_access<access::mode::read>(cgh)};

auto outAcc = out\_buffer.get\_access<access::mode::write>(cgh);

cgh.parallel\_for<class Worker>(num\_items, [=](cl::sycl::id<1> index) {

outAcc[index] = inAcc[0][index] + inAcc[1][index];

});

});

Integration header:

static constexpr

const kernel\_param\_desc\_t kernel\_signatures[] = {

//--- \_ZTSZZ4mainENKUlRN2cl4sycl7handlerEE20->18clES2\_E6Worker

{ kernel\_param\_kind\_t::kind\_accessor, 4062, 0 },

{ kernel\_param\_kind\_t::kind\_accessor, 4062, 32 },

{ kernel\_param\_kind\_t::kind\_accessor, 4062, 64 },

};

Device code generated in pseudo-code form:

struct Capture {

Accessor outAcc;

Accessor inAcc[2];

() {

// Body

}

}

spir\_kernel void caller(

int outAccDim, // args of OutAcc

range<1> outAccR1,

range<1> outAccR2,

id<1> outI,

int inAccDim\_0, // args of inAcc[0]

range<1> inAccR1\_0,

range<1> inAccR2\_0,

id<1> inI\_0,

int inAccDim\_1, // args of inAcc[1]

range<1> inAccR1\_1,

range<1> inAccR2\_1,

id<1> inI\_1,

)

{

// Local capture object

struct Capture local;

// Reassemble capture object from parts

// Call outAcc accessor’s init function

Accessor::init(&local.outAcc, outAccDim, outAccR1, outAccR2, outI);

// Call inAcc[0] accessor’s init function

Accessor::init(&local.inAcc[0], inAccDim\_0, inAccR1\_0, inAccR2\_0, inI\_0);

// Call inAcc[1] accessor’s init function

Accessor::init(&local.inAcc[1], inAccDim\_1, inAccR1\_1, inAccR2\_1, inI\_1);

callee(&local, id<1> wi);

}

## Extension 3: Accessor Arrays within Structs

*Individual* Accessors within structs were already supported. Struct parameters of kernels that are structs are traversed member by member, recursively, to enumerate member structs that are one of the SYCL special types: Accessors and Samplers. For each special struct encountered in the scan, arguments of their init functions are added as separate arguments to the kernel.

However, *arrays* of accessors within structs were not supported. The extension to arrays of Accessors/Samplers within structs is straightforward. Each element of such arrays is treated as an individual object, and the arguments of its init function are added to the kernel arguments in sequence. Within the kernel caller function, the lambda object is reassembled in a manner similar to other instances of Accessor arrays.

Source code fragment:

myQueue.submit([&](handler &cgh) {

using Accessor =

accessor<int, 1, access::mode::read, access::target::global\_buffer>;

struct S {

int m;

Accessor inAcc[2];

} s = { 55,

{in\_buffer1.get\_access<access::mode::read>(cgh),

in\_buffer2.get\_access<access::mode::read>(cgh)}

};

auto outAcc = out\_buffer.get\_access<access::mode::write>(cgh);

cgh.parallel\_for<class Worker>(num\_items, [=](cl::sycl::id<1> index) {

outAcc[index] = s.m + s.inAcc[0][index] + s.inAcc[1][index];

});

});

Integration header:

static constexpr

const kernel\_param\_desc\_t kernel\_signatures[] = {

//--- \_ZTSZZ4mainENKUlRN2cl4sycl7handlerEE20->18clES2\_E6Worker

{ kernel\_param\_kind\_t::kind\_accessor, 4062, 0 },

{ kernel\_param\_kind\_t::kind\_std\_layout, 72, 32 },

{ kernel\_param\_kind\_t::kind\_accessor, 4062, 40 },

{ kernel\_param\_kind\_t::kind\_accessor, 4062, 72 },

};

Device code generated in pseudo-code form:

struct Capture {

Accessor outAcc;

struct S s;

() {

// Body

}

}

spir\_kernel void caller(

int outAccDim, // args of OutAcc

range<1> outAccR1,

range<1> outAccR2,

id<1> outI,

struct S s, // the struct S

int inAccDim\_0, // args of s.inAcc[0]

range<1> inAccR1\_0,

range<1> inAccR2\_0,

id<1> inI\_0,

int inAccDim\_1, // args of s.inAcc[1]

range<1> inAccR1\_1,

range<1> inAccR2\_1,

id<1> inI\_1,

)

{

// Local capture object

struct Capture local;

// Reassemble capture object from parts

// Copy struct argument contents to local copy

// Accessor array will be initialized by calling init functions

local.s = s;

// Call outAcc accessor’s init function

Accessor::init(

&local.outAcc, outAccDim, outAccR1, outAccR2, outI);

// Call s.inAcc[0] accessor’s init function

Accessor::init(

&local.s.inAcc[0], inAccDim\_0, inAccR1\_0, inAccR2\_0, inI\_0);

// Call s.inAcc[1] accessor’s init function

Accessor::init(

&local.s.inAcc[1], inAccDim\_1, inAccR1\_1, inAccR2\_1, inI\_1);

callee(&local, id<1> wi);

}