Appendix 2: WebCL to OpenCL source-to-source instrumentation to prevent invalid memory access

Problem description

Basically there are three different cases to consider when adding checks to a memory access:

1. We know at compile time that the access will be inside memory allocated to the program.
2. We know at compile time the limits which the access must respect
3. We don't even know the limits at compile time (this causes most overhead to support)

Case 1 covers situations where, for example, we have a have constant length loop that does indexing.

Case 2 is a case where indexing depends on input values fed to the program (e.g. kernel arguments or get_global_id is used in indexing / pointer arithmetic.)

Case 3 is when we have an indirect pointer where the address referred to by the indirect pointer is modified in the program in a non-trivial way.

Case 3 causes the biggest memory and processing overhead because it requires that pointer limits are passed at runtime in the program.

The following example illustrates case 3 type of a problem:

```
float function(int user_input) {
  float table1[5] = {1,2,3,4,5};
  float table2[2] = {6,7};

  // table of pointers to make things more complicated, each pointer in table actually require
  // 3 addresses to be stored to because we need to reserve space for storing limits
  float* indirect_table[2];

  // In instrumented code in addition to the original assignment also pointer limits needs to be stored
     // indirect_table[0]->cur = table; indirect_table[0]->min = &table1[0]; indirect_table[0]->max = &table1[4];
     // indirect_table[1]->cur = table; indirect_table[1]->min = &table2[0]; indirect_table[1]->max = &table2[1];
  indirect_table[0] = table1;
  indirect_table[1] = table2;

  // limits for clamping are fetched runtime from pointer before reading the value to return
```

```
  return indirect_table[user_input][4];
}
```

Proposed solution

Bad solution: Trivial smart pointer approach

Normal pointers in the program are replaced with smart pointers that contain, in addition to the actual pointer value, also the limits of the area being pointed to (for example the limits of a statically allocated array).

Pros:

- instrumented code stays readable
- driver optimizations might be able to reduce memory consumption by removing unnecessary temporary values
- precise information of limits of each memory access

Cons:

- 3x memory consumption when storing pointers
- performance hit due to more memory accesses
- GPU has very limited local memory which can cause problems of fitting existing algorithms to WebCL
- especially having a table of pointers is very expensive

If we could add a restriction that indirect pointers and pointer arrays are not supported basically all negative aspects could be prevented at the cost of removing features from the language.

Solution: Collect statically allocated memory to one contiguous area

This approach is based on an already ongoing WebCL code instrumentation project, which is done at LLVM IR level: https://github.com/toaarnio/llvm-ir-memprotect.

If we collect all static memory of each address space to be contained in one big global memory area, we can always clamp memory accesses to the same limits &(address_space.first) and &(address_space.last)-1 (last limit needs some pointer casting depending on sizeof(type) which is being accessed -- better explanation is in code instrumentation example).

One program may contain a few different memory areas:

1. Global memory area(s). OpenCL does not allow kernel to statically allocate memory from global address space. All global memory areas available for the program are passed to kernel in kernel arguments, each global memory argument defines one separate area.

In the worst case if there is non-trivial indirect pointer arithmetic in the program done with global memory pointers we need to do 1 clamping per each global area argument passed to the kernel. However this

overhead can be recognized at compile time and circumvented quite easily by the user by making a small a change to the kernel code (using local / private temporary memory instead of directly doing complicated arithmetics with global memory pointers).

2. Constant memory area is read-only and the kernel can allocate it statically. It is possible that constant memory parameter can be passed to the kernel as a parameter, so there might exist various memory areas which could require checking.

However, not being allowed to write to constant memory guarantees that we always know at compile time the limits the pointers must respect so it should be enough to do max 1 limit check per access.

3. Local memory is always uninitialized memory and it can be given as an argument to the kernel or it can be allocated statically by the kernel. We can always unify this memory to be one contiguous area whose limits are fast to check (actual values of limits can be computed compile time which makes clamping even faster).

4. Private memory is easy to initialize to zero and it is trivial to pool all private memory to be one contiguous area.

Four points above implicates that we can effectively resolve limits of every pointer in compile time, except for the global memory pointers whose instrumentation in the worst case will be a bit more inefficient.

To prevent this overhead the compiler can instruct user to copy the needed values from global memory to local or private memory whose address space limits are known at compile time. This is already commonly used OpenCL optimization technique due to better local memory performance on GPU.

Trivial example of source-to-source transformation

In this example we know the limits at compile time but we don't know if the limits will be violated. The example demonstrates the idea of collecting memory to one contiguous area (for each address space) whose limits are further used to do actual clamping.

```
float4 helper_function(local float4* helper_in) {
	float4 temp = *helper_in;
	return temp*temp;
}

webcl_kernel void kernel_function(global float4* input, global float4* output) {
	size_t id = get_global_id();
	local float4 temp = input[id];
	output[id] = helper_function(&temp);
}
```

Instrumentation unleashed

Because OpenCL does not support recursion we always know statically how much stack space is needed, which makes it possible to promote all auto variables to global structure fields.

```
typedef private struct {
        float4 helper_function_temp;
        size_t kernel_function_id;
        char    last_address;
} PrivateAddressSpaceType;
PrivateAddressSpaceType private_mem = {(float4)(0), 0};

typedef local struct {
        float4 kernel_function_temp;
        char    last_address;
} LocalAddressSpaceType;

// no initialiser allowed, needs to be zeroed when kernel is ran or by vendor driver
// or we might be able to trace if value is read before writing.
// One option is to generate zero initialisation routine to start of kernel function which
// initializes parts of local memory
LocalAddressSpaceType local_mem;

float4 helper_function(local float4* helper_in) {
        // cast last address to type that we are accessing
         // and find the last valid position in memory where we can access that type:
        // *( ((local float4*)last_local_address) - 1)
        private_mem.helper_function_temp =
            *(max(&local_mem, min(((local float4*)&(local_mem.last_address))-1), helper_in));
        return private_mem.helper_function_temp*private_mem.helper_function_temp;
}

// webcl driver will tell kernel element count of passed memory object
// (instrumentation can convert kernel to correct format, which adds
// driver generated parameter after each pointer passed to kernel)
webcl_kernel void kernel_function(
    global float4* input, size_t input_count,
    global float4* output, size_t output_count) {
        private_mem.kernel_function_id = get_global_id();
        local_mem.kernel_function_temp = input[id%input_count];
        output[id%output_count] = helper_function(&local_mem.kernel_function_temp);
}
```

Pros:

- minimal memory overhead
- no additional overhead when loading / storing pointers
- supports all types of memory accessing, one can even cast a constant to a pointer and the instrumentation can clamp it to limits
- pretty easy to implement even in comparison to trivial smart pointer approach, does not require touching helper function signatures

Cons:

- could make some later compiler optimisations harder to implement or might need an additional "remove unused structure elements" pass
- less strict checking; system allows to access memory at invalid addresses as long as those addresses are within the area the kernel has allocated

Additional notes about RFQ:

- There are also builtin functions which takes pointer arguments, these might be unsafe and require additional handling
- Memory zeroing in source code level might not guarantee that memory is really zeroed. E.g. if struct { uchar, float* } is aligned to have 3 bytes of padding after first element and those padding bytes in worse case will contain leaked data. Might not be real problem if all vendor implementations initialize padding to 0. Alternatively, at source code level we could for example allocate the structure first as a static array initialized with zeros and then cast it to the actual structure type and do the initialization in run-time.