WebCL Memory Protection

Source-to-source instrumentation

Vincit Ltd: Mikael Lepistö, Rami Ylimäki

About us

- **Vincit**: Small and agile software house (~70 people) filled with pretty good people

- This protection method was found during a Nokia Research Center project where we investigated different options for memory protection for OpenCL in LLVM IR level

Web is pretty dangerous

- Running untrusted code without sandbox may...
- crash your GPU and/or workstation
- steal your thoughts

Protecting Memory

- The memory allocated in the program has to be initialized (prevent access to old data from other programs)
- Untrusted code must not access invalid memory

Fixing initializers

- OpenCL does not allow to initialize local memory
- Private variables might be uninitialized
- Pretty easy to handle (more about it later)

Memory Accesses

- Kernels have pretty limited ways to access memory:

- table[index]
- struct_ptr->field
- vector_type_ptr->x
- *(any_address)

- Some builtins may take pointer arguments

To protect memory we have to

- Keep track of memory allocations (also in runtime if platform supports dynamic allocation)
- Trace valid ranges for reads and writes
- Validate them efficiently while compiling or at run-time

Is it that hard?

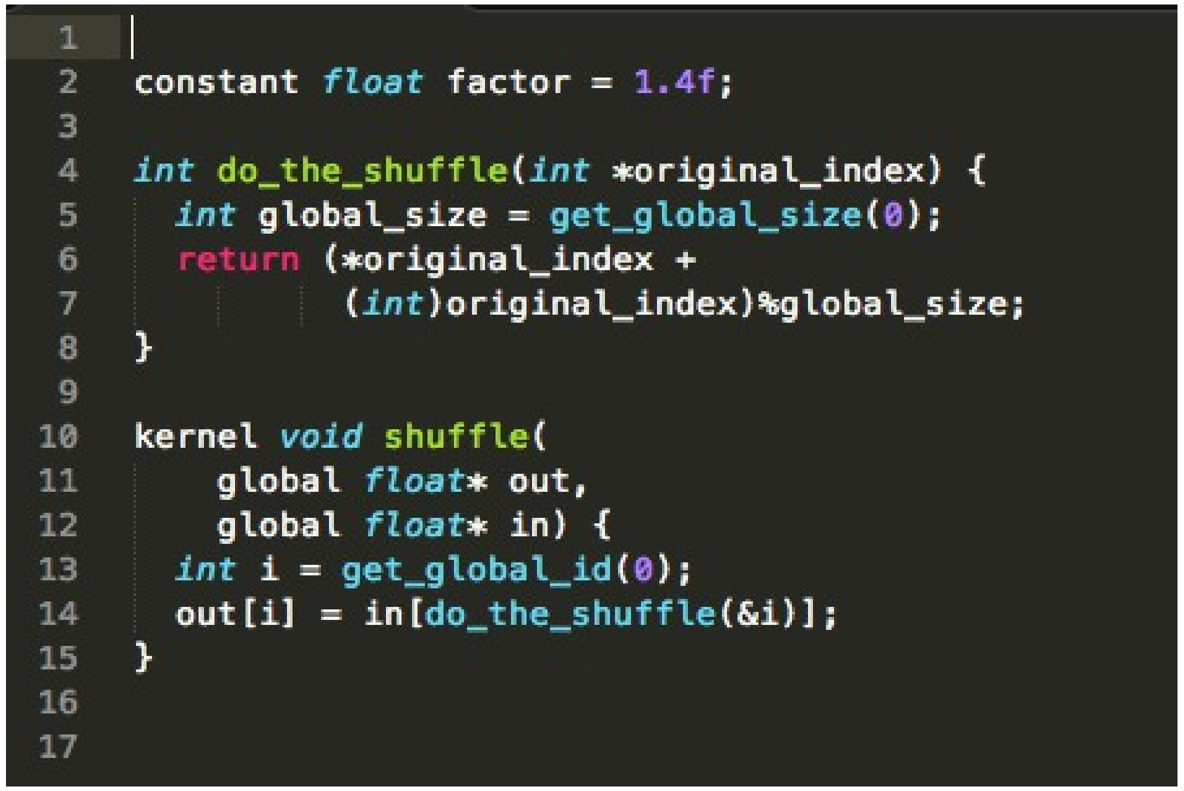
- Tracing valid ranges for pointers at compile time requires good analysis
- Extensive compile time analysis not available while working at AST (Abstract Syntax Tree) level
- Run time tracking may cause 3x more reads/writes for pointer assignments and 3x memory overhead for e.g. pointer arrays
- There are cases where tracking is impossible even run-time e.g. int → ptr casting.

How OpenCL makes the task easier?

We know our memory

- OpenCL does not allow recursion so we know amount of private memory at compile time
- No dynamic allocation in kernel side
- When entering the kernel we know also start addresses of all memory available for the program
- Different memories have different address spaces

Memory accessible for kernel



```

constant float factor = 1.4f;

int do_the_shuffle(int *original_index) {
    int global_size = get_global_size(0);
    return (*original_index +
                (int)original_index)%global_size;
}

kernel void shuffle(
        global float* out,
        global float* in) {
    int i = get_global_id(0);
    out[i] = in[do_the_shuffle(&i)];
}


```

- Constant address space allocation:

  float factor

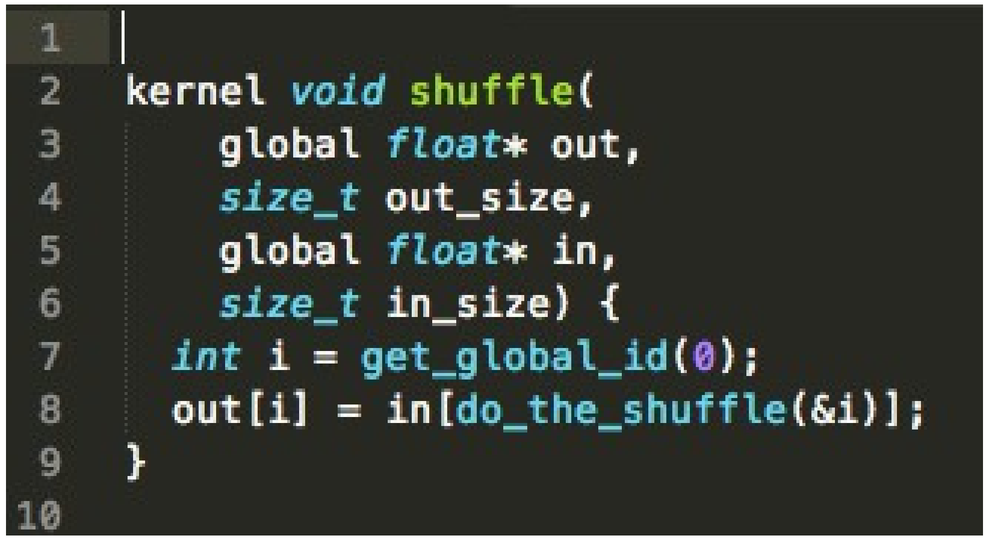
- Private address space allocations:

  int global_size and int i

- Global address space allocations:

  Data allocated for *in and *out pointers.

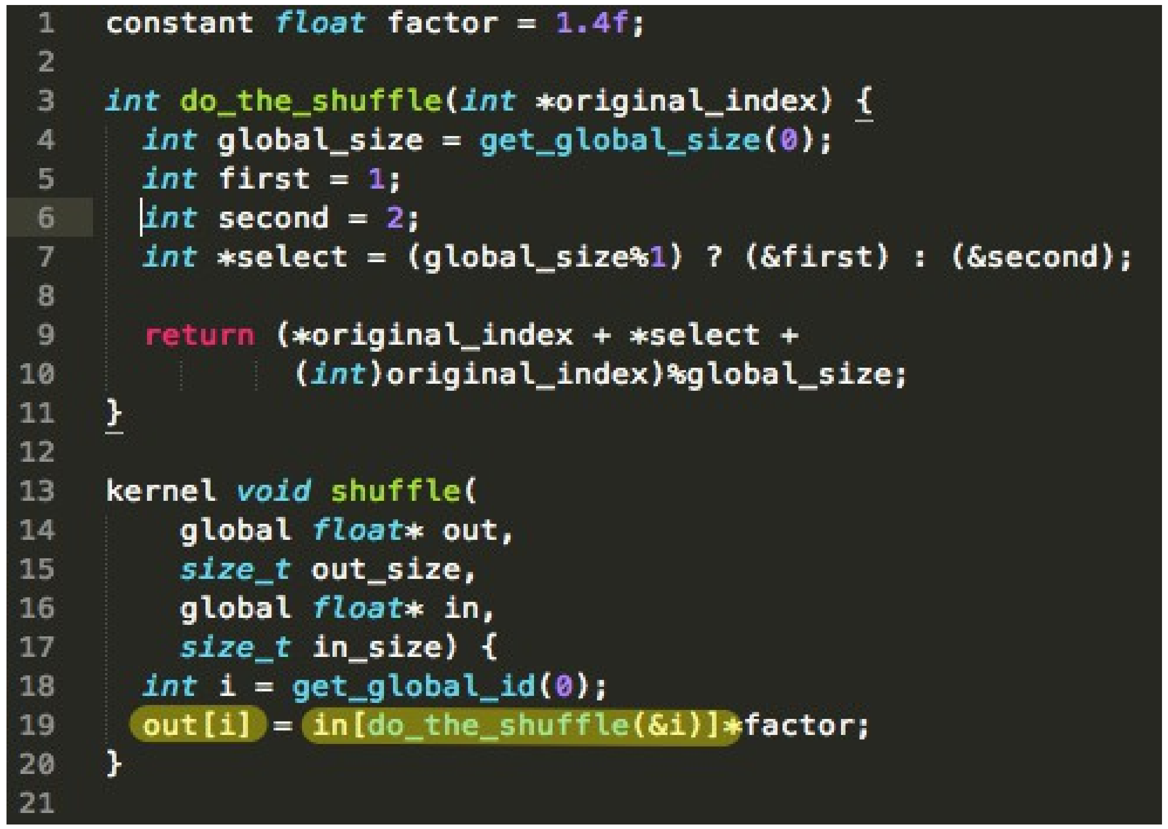
How WebCL further helps us out?



```

kernel void shuffle(
     global float* out,
     size_t out_size,
     global float* in,
     size_t in_size) {
   int i = get_global_id(0);
   out[i] = in[do_the_shuffle(&i)];
}

```

- It tells us how many elements are allocated in each kernel pointer arguments.
- in_size and out_size tells us how much there is accessible memory passed to kernel
- Effectively we know start and end addresses of all the memory which is meant to be accessed by program.

Global memory checks



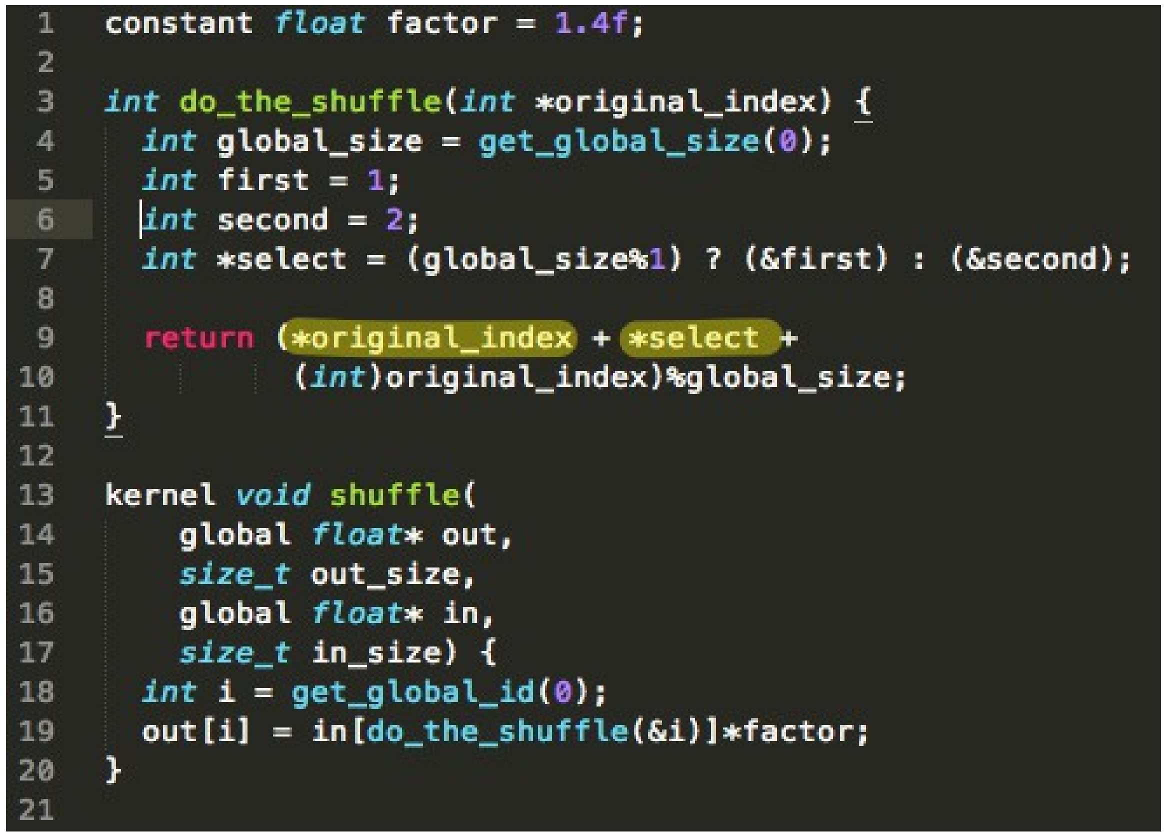
```
constant float factor = 1.4f;

int do_the_shuffle(int *original_index) {
  int global_size = get_global_size(0);
  int first = 1;
  int second = 2;
  int *select = (global_size%1) ? (&first) : (&second);

  return (*original_index + *select +
          (int)original_index)%global_size;
}

kernel void shuffle(
    global float* out,
    size_t out_size,
    global float* in,
    size_t in_size) {
  int i = get_global_id(0);
  out[i] = in[do_the_shuffle(&i)]*factor;
}
```

- There are 4 memory accesses: *original_index, *select, out[...] and in[...]

- Without any static analysis we know that out[...] and in[...] has two possible ranges: (&out[0], &out[out_size]) or (&in[0], &in[in_size])

- Making few compares for each global memory access is not really big problem for the performance, since those accesses are already pretty slow

Private address space checks



```
constant float factor = 1.4f;

int do_the_shuffle(int *original_index) {
  int global_size = get_global_size(0);
  int first = 1;
  int second = 2;
  int *select = (global_size%1) ? (&first) : (&second);

  return (*original_index + *select +
            (int)original_index)%global_size;
}

kernel void shuffle(
    global float* out,
    size_t out_size,
    global float* in,
    size_t in_size) {
  int i = get_global_id(0);
  out[i] = in[do_the_shuffle(&i)]*factor;
}

```

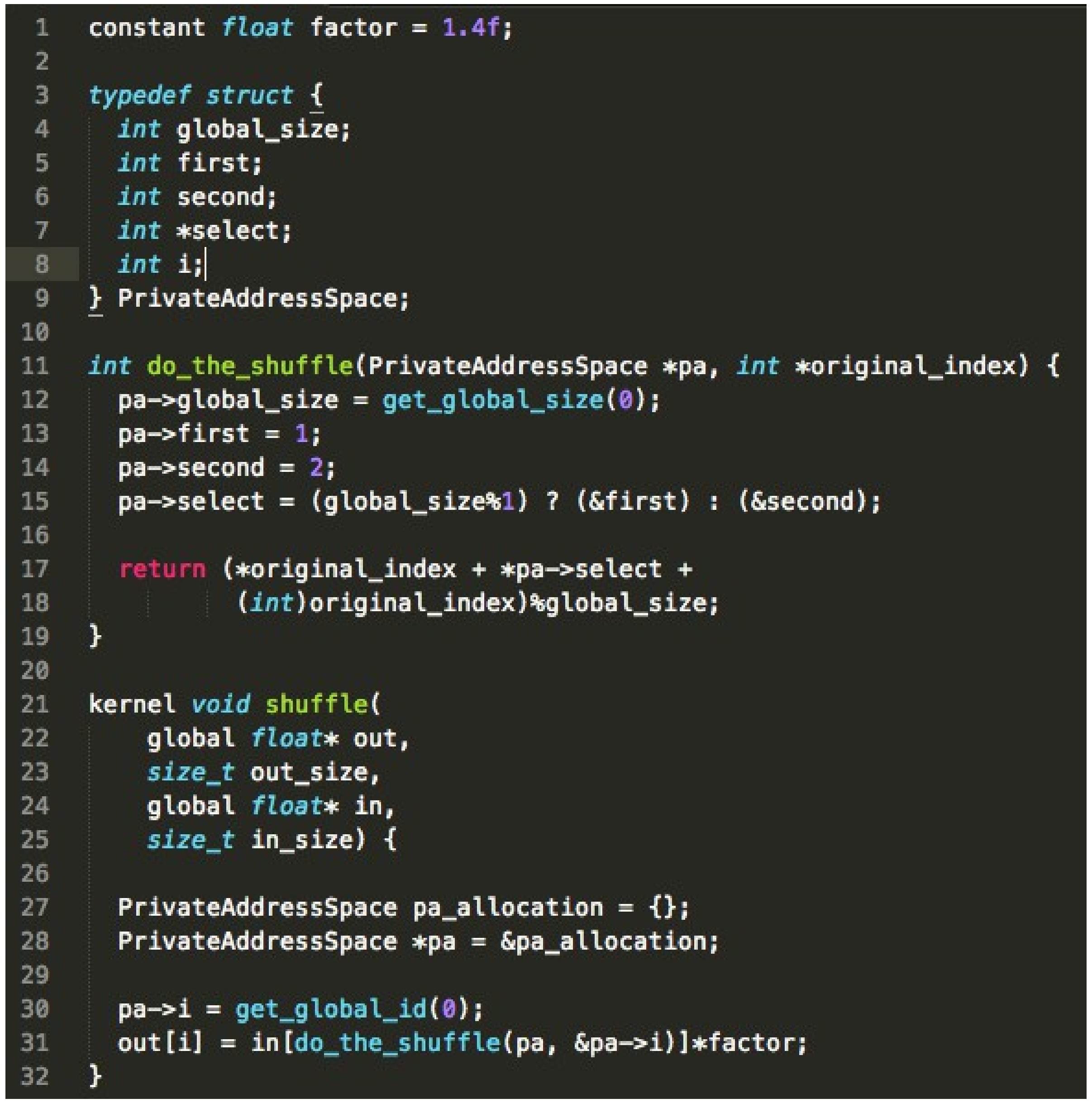
- Private memory should be really fast so all the extra overhead may have a big hit to performance

- In this example in do_the_shuffle() scope there might be 4 valid fragments to check:

  global_size, first, second or select

- For real programs the problem would be a lot worse

Reducing fragments

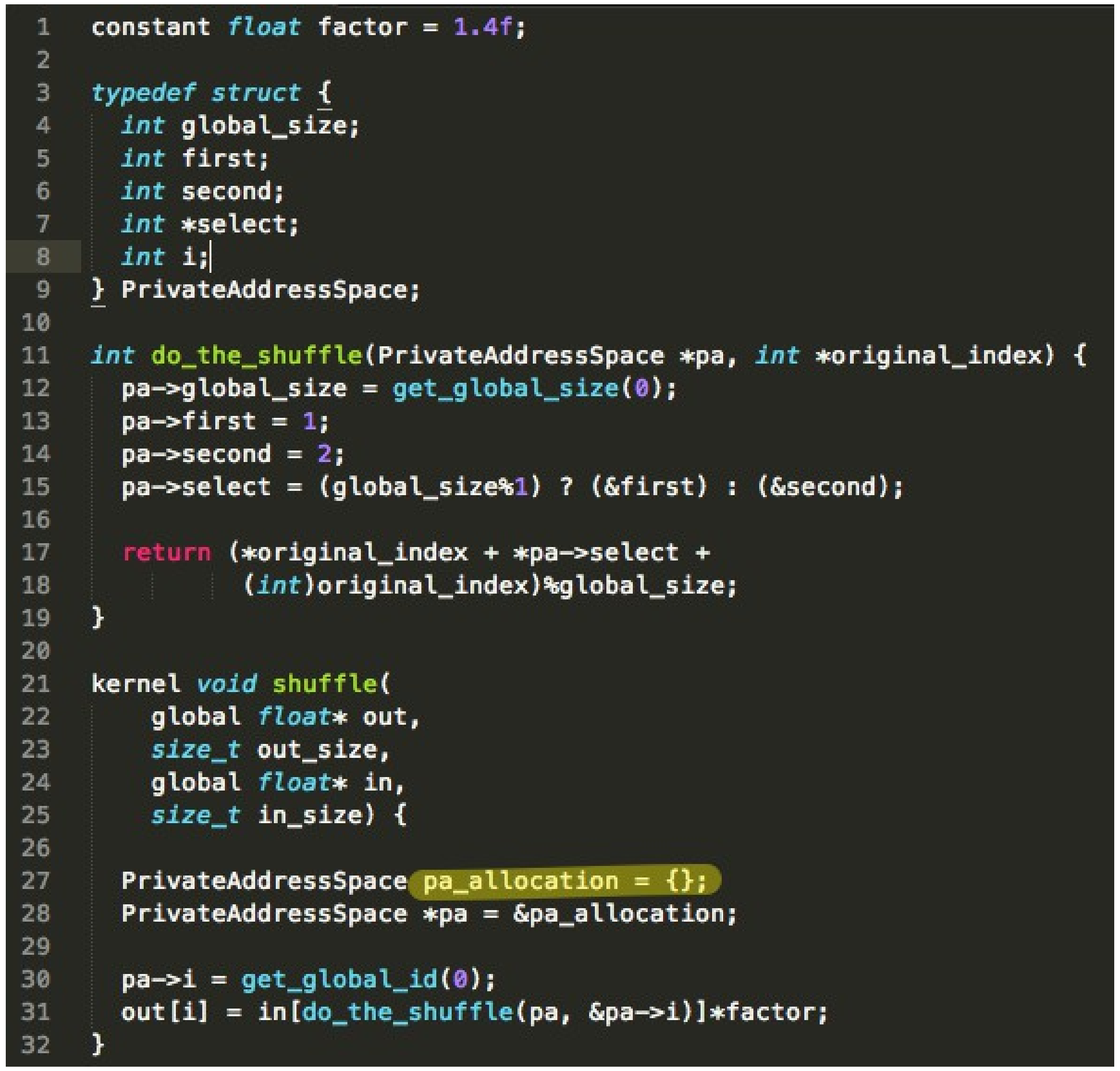


```
constant float factor = 1.4f;

typedef struct {
  int global_size;
  int first;
  int second;
  int *select;
  int i;
} PrivateAddressSpace;

int do_the_shuffle(PrivateAddressSpace *pa, int *original_index) {
  pa->global_size = get_global_size(0);
  pa->first = 1;
  pa->second = 2;
  pa->select = (global_size%1) ? (&first) : (&second);

  return (*original_index + *pa->select +
          (int)original_index)%global_size;
}

kernel void shuffle(
    global float* out,
    size_t out_size,
    global float* in,
    size_t in_size) {

  PrivateAddressSpace pa_allocation = {};
  PrivateAddressSpace *pa = &pa_allocation;

  pa->i = get_global_id(0);
  out[i] = in[do_the_shuffle(pa, &pa->i)]*factor;
}
```

- Collect all the private variables to one struct
- Now we know that all the private address space accesses must be inside memory area

  (&pa[0], &pa[1])
- Only one range check per access!

Handling initialization



```
constant float factor = 1.4f;

typedef struct {
  int global_size;
  int first;
  int second;
  int *select;
  int i;
} PrivateAddressSpace;

int do_the_shuffle(PrivateAddressSpace *pa, int *original_index) {
  pa->global_size = get_global_size(0);
  pa->first = 1;
  pa->second = 2;
  pa->select = (global_size%1) ? (&first) : (&second);

  return (*original_index + *pa->select +
          (int)original_index)%global_size;
}

kernel void shuffle(
    global float* out,
    size_t out_size,
    global float* in,
    size_t in_size) {

  PrivateAddressSpace pa_allocation = {};
  PrivateAddressSpace *pa = &pa_allocation;

  pa->i = get_global_id(0);
  out[i] = in[do_the_shuffle(pa, &pa->i)]*factor;
}
```

- Private memory zero-init comes as a side effect when we allocate address space structure
- C99 standard specifies that we can zero-initialize nested structures with {} [1]
- There is also test case added to make sure that drivers does initialization correctly
- For local memory initialization is done parallel so that each local work item initializes just part of the memory. If memory initialization extension is enabled from OpenCL local memory is not initialized at all

[1] http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf chapter 6.7.8 paragraphs 10 and 21

Memory Accesses

- All can be presented in format: *(address)[.<field>]

- table[index] ==> *(table + index)
- struct_ptr->field ==> *(struct_ptr).field
- vector_type_ptr->x ==> *(vector_type_ptr).x
- *(any_address) ==> *(any_address)

- We can change *(address) ==> *(SAFE(addr, min, max)) which will never access invalid memory

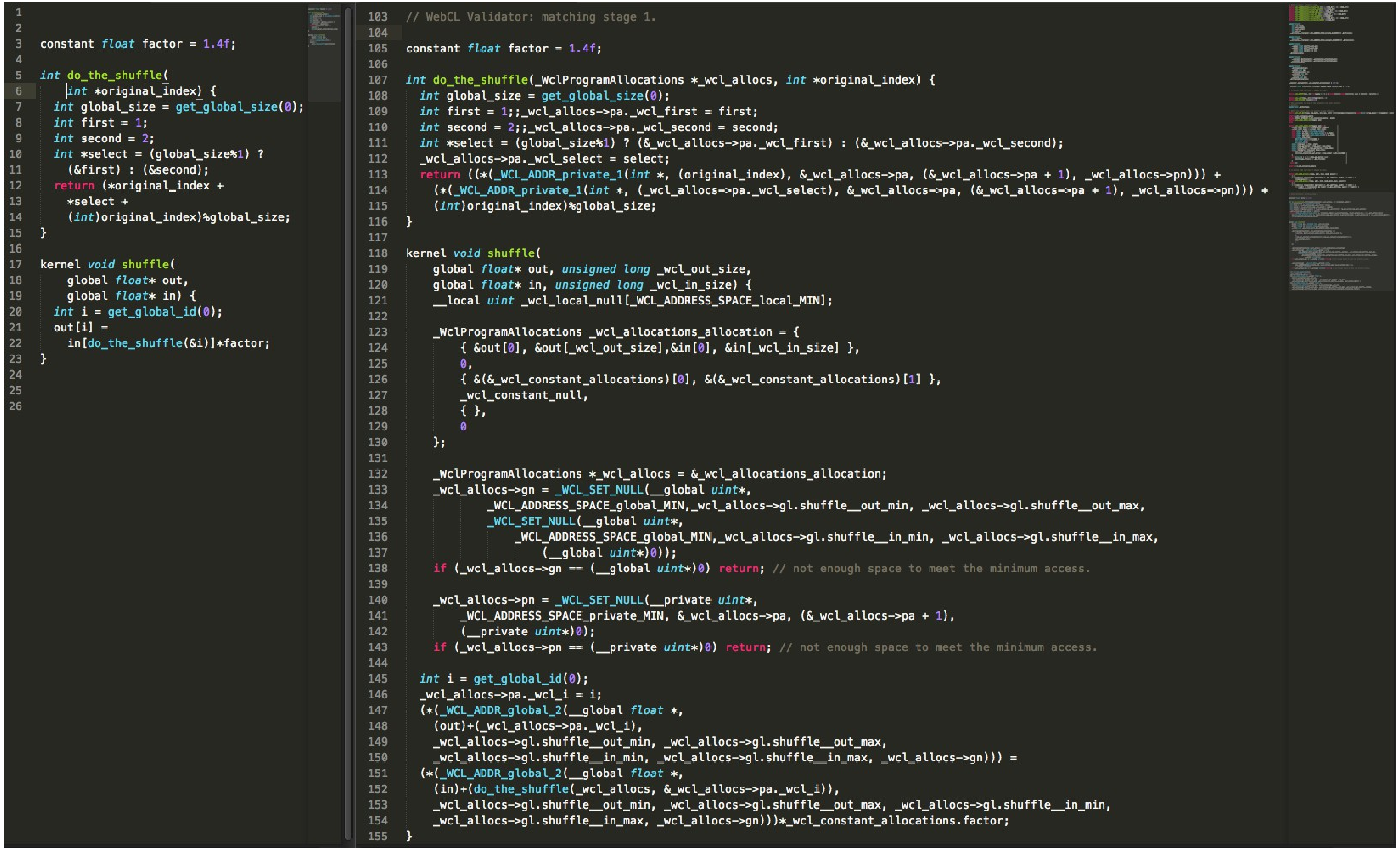
Reality is not that beautiful

- Multiple ranges
- Null pointer in case of invalid access
- Knowing access type
- Moving type declarations to before address space structures (and handling declaration scopes)
- Function arguments passed to other function as address
- Declarations in strange places (e.g. for statement)
- Much more...

Not Another Software Company

vincit

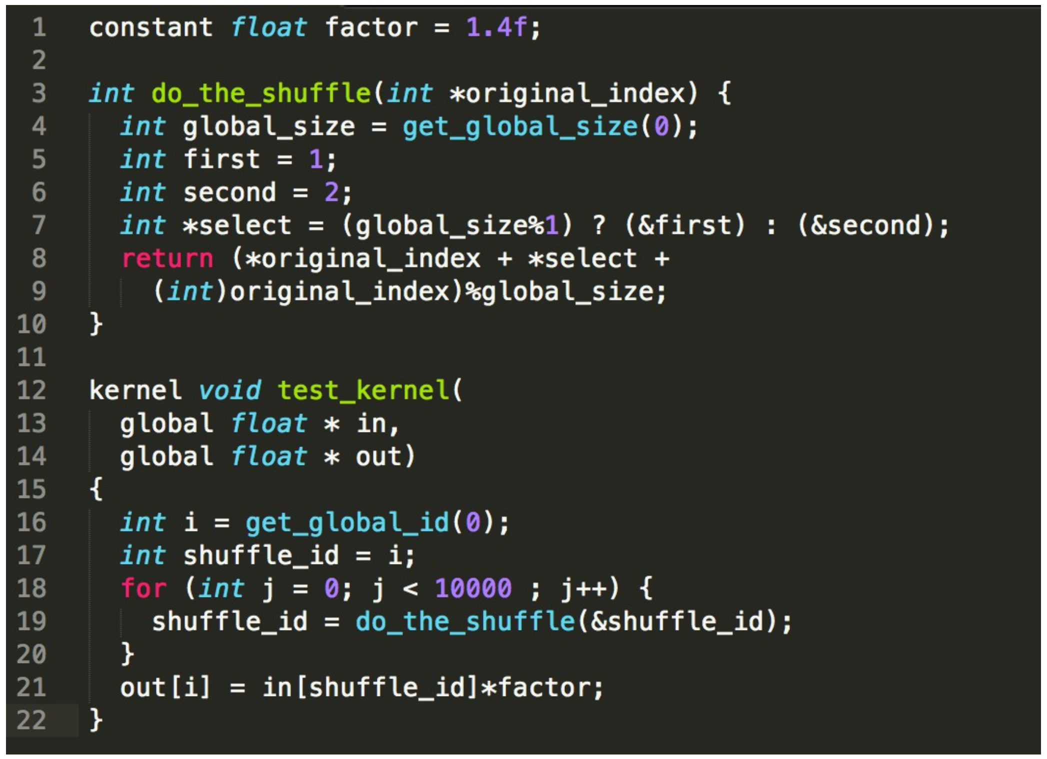
However it works!



```
constant float factor = 1.4f;

int do_the_shuffle(
    int *original_index) {
  int global_size = get_global_size(0);
  int first = 1;
  int second = 2;
  int *select = (global_size%1) ?
    (&first) : (&second);
  return (*original_index +
    *select +
    (int)original_index)%global_size;
}

kernel void shuffle(
    global float* out,
    global float* in) {
  int i = get_global_id(0);
  out[i] =
    in[do_the_shuffle(&i)]*factor;
}
```

```
// WebCL Validator: matching stage 1.

constant float factor = 1.4f;

int do_the_shuffle(_WclProgramAllocations *_wcl_allocs, int *original_index) {
  int global_size = get_global_size(0);
  int first = 1;;_wcl_allocs->pa._wcl_first = first;
  int second = 2;;_wcl_allocs->pa._wcl_second = second;
  int *select = (global_size%1) ? (&_wcl_allocs->pa._wcl_first) : (&_wcl_allocs->pa._wcl_second);
  _wcl_allocs->pa._wcl_select = select;
  return ((*(_WCL_ADDR_private_1(int *, (original_index), &_wcl_allocs->pa, (&_wcl_allocs->pa + 1), _wcl_allocs->pn))) +
    (*(_WCL_ADDR_private_1(int *, (_wcl_allocs->pa._wcl_select), &_wcl_allocs->pa, (&_wcl_allocs->pa + 1), _wcl_allocs->pn))) +
    (int)original_index)%global_size;
}

kernel void shuffle(
    global float* out, unsigned long _wcl_out_size,
    global float* in, unsigned long _wcl_in_size) {
    __local uint _wcl_local_null[_WCL_ADDRESS_SPACE_local_MIN];

    _WclProgramAllocations _wcl_allocations_allocation = {
        { &out[0], &out[_wcl_out_size],&in[0], &in[_wcl_in_size] },
        0,
        { &(&_wcl_constant_allocations)[0], &(&_wcl_constant_allocations)[1] },
        _wcl_constant_null,
        { },
        0
    };

    _WclProgramAllocations *_wcl_allocs = &_wcl_allocations_allocation;
    _wcl_allocs->gn = _WCL_SET_NULL(__global uint*,
            _WCL_ADDRESS_SPACE_global_MIN,_wcl_allocs->gl.shuffle__out_min, _wcl_allocs->gl.shuffle__out_max,
            _WCL_SET_NULL(__global uint*,
                _WCL_ADDRESS_SPACE_global_MIN,_wcl_allocs->gl.shuffle__in_min, _wcl_allocs->gl.shuffle__in_max,
                    (__global uint*)0));
    if (_wcl_allocs->gn == (__global uint*)0) return; // not enough space to meet the minimum access.

    _wcl_allocs->pn = _WCL_SET_NULL(__private uint*,
        _WCL_ADDRESS_SPACE_private_MIN, &_wcl_allocs->pa, (&_wcl_allocs->pa + 1),
        (__private uint*)0);
    if (_wcl_allocs->pn == (__private uint*)0) return; // not enough space to meet the minimum access.

  int i = get_global_id(0);
  _wcl_allocs->pa._wcl_i = i;
  (*(_WCL_ADDR_global_2(__global float *,
    (out)+(_wcl_allocs->pa._wcl_i),
    _wcl_allocs->gl.shuffle__out_min, _wcl_allocs->gl.shuffle__out_max,
    _wcl_allocs->gl.shuffle__in_min, _wcl_allocs->gl.shuffle__in_max, _wcl_allocs->gn))) =
  (*(_WCL_ADDR_global_2(__global float *,
    (in)+(do_the_shuffle(_wcl_allocs, &_wcl_allocs->pa._wcl_i)),
    _wcl_allocs->gl.shuffle__out_min, _wcl_allocs->gl.shuffle__out_max, _wcl_allocs->gl.shuffle__in_min,
    _wcl_allocs->gl.shuffle__in_max, _wcl_allocs->gn)))*_wcl_constant_allocations.factor;
}
```

Trying it out

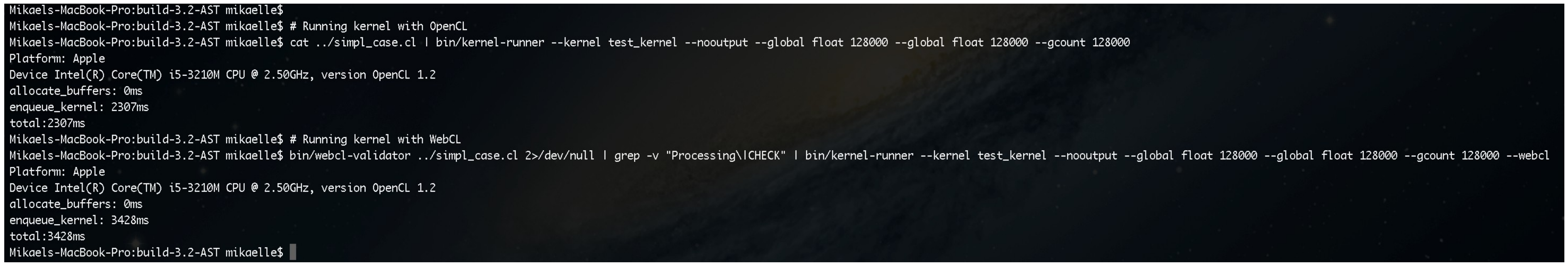


```
constant float factor = 1.4f;

int do_the_shuffle(int *original_index) {
  int global_size = get_global_size(0);
  int first = 1;
  int second = 2;
  int *select = (global_size%1) ? (&first) : (&second);
  return (*original_index + *select +
    (int)original_index)%global_size;
}

kernel void test_kernel(
  global float * in,
  global float * out)
{
  int i = get_global_id(0);
  int shuffle_id = i;
  for (int j = 0; j < 10000 ; j++) {
    shuffle_id = do_the_shuffle(&shuffle_id);
  }
  out[i] = in[shuffle_id]*factor;
}
```

- Added inner loop to be able to measure execution times
- bin/kernel-runner just creates buffers to give kernels and if –webcl switch is given it also adds size argument after each buffer to comply webcl calling interface
- Running over with global work size: 128k
- On Apple: Device Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz, version OpenCL 1.2

  Original code: 2307ms / Protected code: 3428ms
- On NVIDIA: GeForce GTX 650, version OpenCL 1.1 CUDA

  Original code: 137ms / Protected code: 265ms



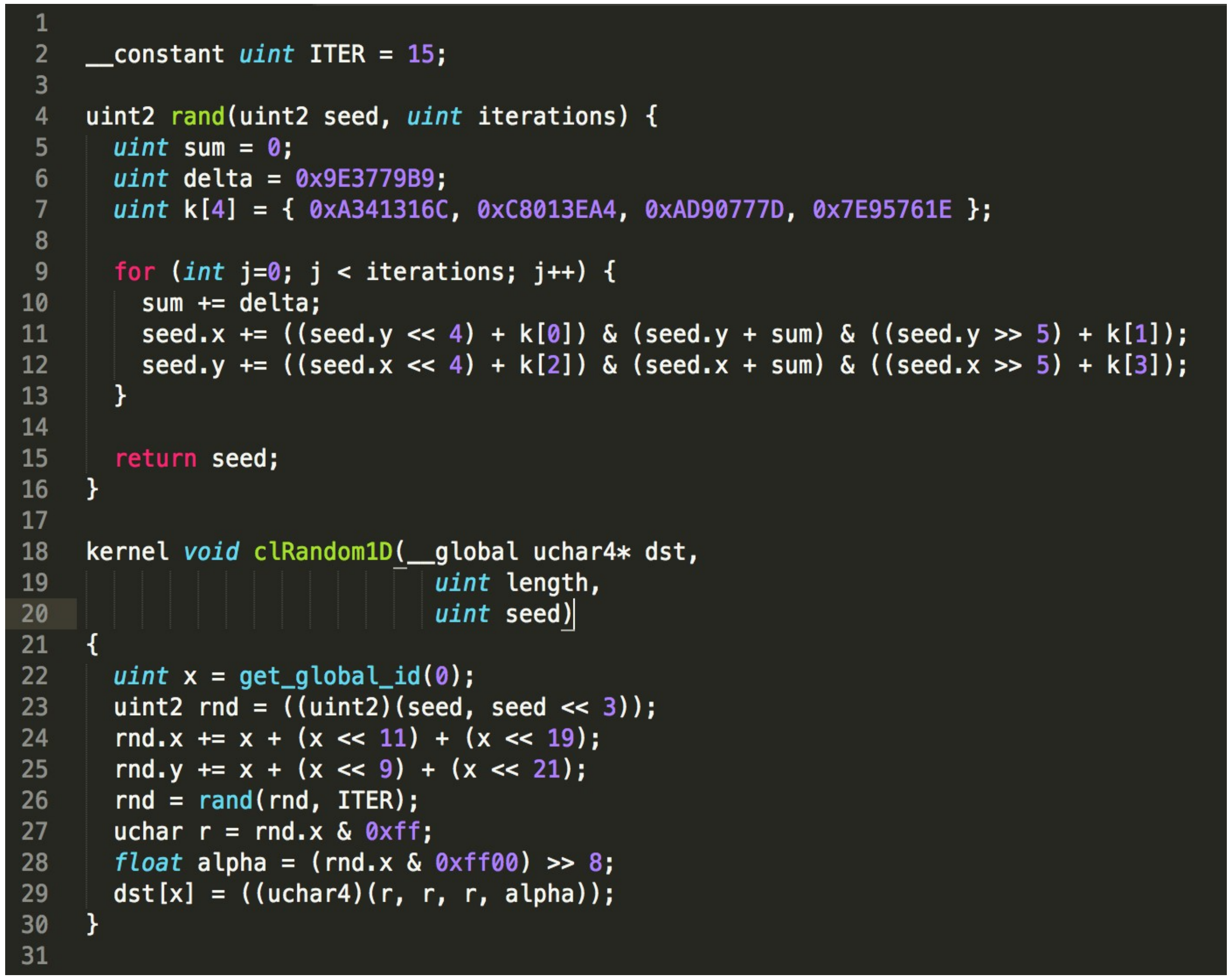
```
Mikaels-MacBook-Pro:build-3.2-AST mikaelle$
Mikaels-MacBook-Pro:build-3.2-AST mikaelle$ # Running kernel with OpenCL
Mikaels-MacBook-Pro:build-3.2-AST mikaelle$ cat ../simpl_case.cl | bin/kernel-runner --kernel test_kernel --nooutput --global float 128000 --global float 128000 --gcount 128000
Platform: Apple
Device Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz, version OpenCL 1.2
allocate_buffers: 0ms
enqueue_kernel: 2307ms
total:2307ms
Mikaels-MacBook-Pro:build-3.2-AST mikaelle$ # Running kernel with WebCL
Mikaels-MacBook-Pro:build-3.2-AST mikaelle$ bin/webcl-validator ../simpl_case.cl 2>/dev/null | grep -v "Processing\|CHECK" | bin/kernel-runner --kernel test_kernel --nooutput --global float 128000 --global float 128000 --gcount 128000 --webcl
Platform: Apple
Device Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz, version OpenCL 1.2
allocate_buffers: 0ms
enqueue_kernel: 3428ms
total:3428ms
Mikaels-MacBook-Pro:build-3.2-AST mikaelle$
```

Not Another Software Company

vincit

Random Number Generator case

(kindly tested by Tomi Aarnio)



```

__constant uint ITER = 15;

uint2 rand(uint2 seed, uint iterations) {
  uint sum = 0;
  uint delta = 0x9E3779B9;
  uint k[4] = { 0xA341316C, 0xC8013EA4, 0xAD90777D, 0x7E95761E };

  for (int j=0; j < iterations; j++) {
    sum += delta;
    seed.x += ((seed.y << 4) + k[0]) & (seed.y + sum) & ((seed.y >> 5) + k[1]);
    seed.y += ((seed.x << 4) + k[2]) & (seed.x + sum) & ((seed.x >> 5) + k[3]);
  }

  return seed;
}

kernel void clRandom1D(__global uchar4* dst,
                        uint length,
                        uint seed)
{
  uint x = get_global_id(0);
  uint2 rnd = ((uint2)(seed, seed << 3));
  rnd.x += x + (x << 11) + (x << 19);
  rnd.y += x + (x << 9) + (x << 21);
  rnd = rand(rnd, ITER);
  uchar r = rnd.x & 0xff;
  float alpha = (rnd.x & 0xff00) >> 8;
  dst[x] = ((uchar4)(r, r, r, alpha));
}

```

• http://webcl.nokiaresearch.com/rng/	
• AMD CPU:	338 ms vs. 514 ms
• Intel CPU:	108 ms vs. 340 ms
• NVIDIA GPU:	28 ms vs. 80 ms

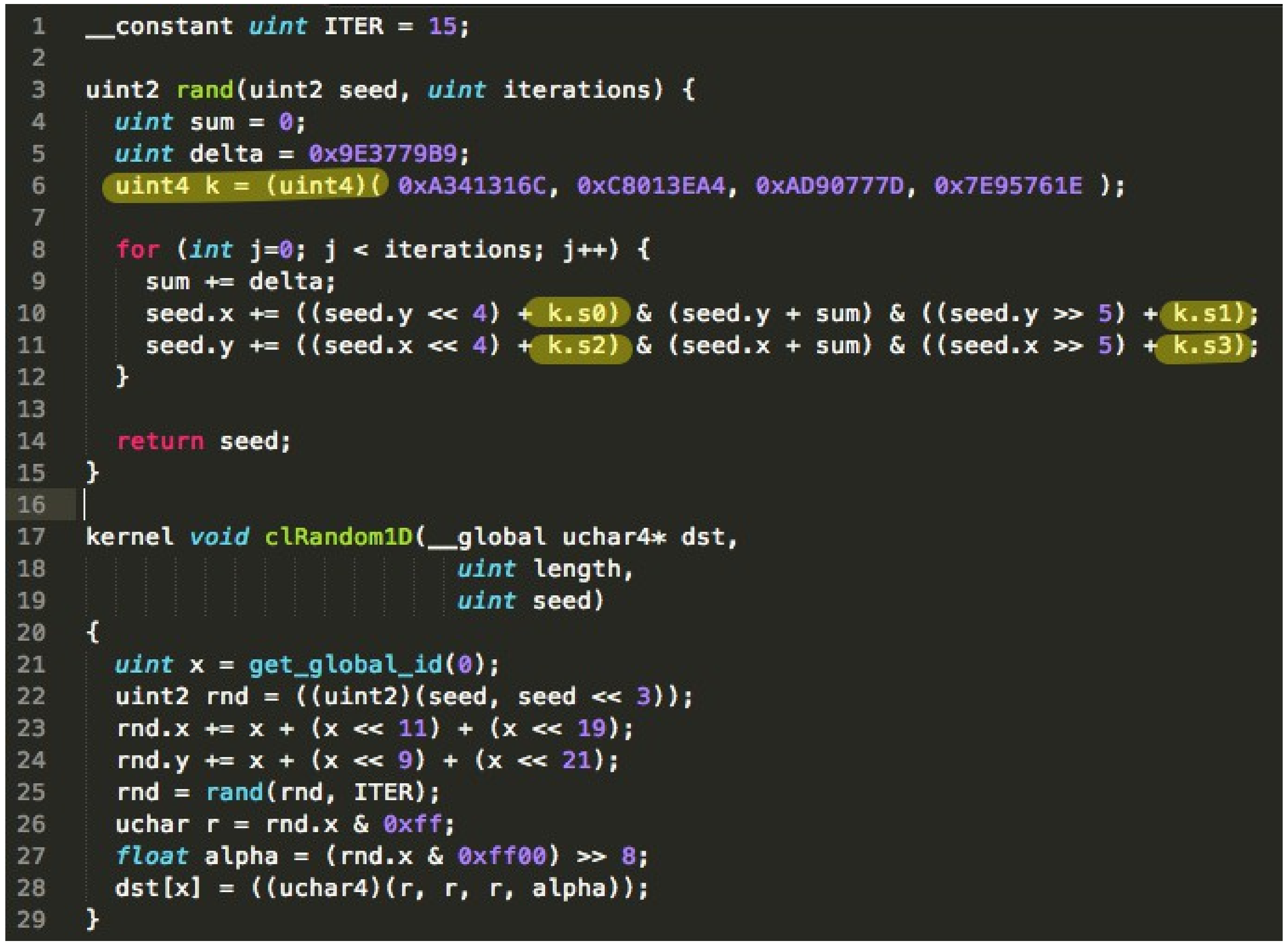
- Overhead 1.5x – 3x

- One case for better static analysis

- For now we can change k[4] to uint4 and get better results

Random Number Generator case 2

(kindly tested by Tomi Aarnio)



```
__constant uint ITER = 15;

uint2 rand(uint2 seed, uint iterations) {
  uint sum = 0;
  uint delta = 0x9E3779B9;
  uint4 k = (uint4)( 0xA341316C, 0xC8013EA4, 0xAD90777D, 0x7E95761E );

  for (int j=0; j < iterations; j++) {
    sum += delta;
    seed.x += ((seed.y << 4) + k.s0) & (seed.y + sum) & ((seed.y >> 5) + k.s1);
    seed.y += ((seed.x << 4) + k.s2) & (seed.x + sum) & ((seed.x >> 5) + k.s3);
  }

  return seed;
}

kernel void clRandom1D(__global uchar4* dst,
                       uint length,
                       uint seed)
{
  uint x = get_global_id(0);
  uint2 rnd = ((uint2)(seed, seed << 3));
  rnd.x += x + (x << 11) + (x << 19);
  rnd.y += x + (x << 9) + (x << 21);
  rnd = rand(rnd, ITER);
  uchar r = rnd.x & 0xff;
  float alpha = (rnd.x & 0xff00) >> 8;
  dst[x] = ((uchar4)(r, r, r, alpha));
}
```

Xeon X5570, OpenCL 1.1 AMD-APP (831.4):

Array version: 324ms vs. 515ms

Vector version: 284ms vs. 347ms

Xeon X5570, Intel OpenCL 1.2 (Build 63463):

Array version : 99ms vs. 323ms

Vector version: 77ms vs. 87ms

NVIDIA GTX 650 Ti Boost:

Array version : 30ms vs. 89ms

Vector version: 30ms vs. 30ms

- Good static analysis and guiding user about slow code constructs can make big difference

Implementation

- Implemented as Clang 3.2 tool separated in few different passes

  ~ 6k lines of well commented code

  ~ 2.5k lines of OpenCL test cases

- Stripped binary size on OSX 8MB
- Compressed executable binary 2.7MB (compressed with http://upx.sourceforge.net/) decompress overhead ~70ms
- Tested on Linux, Windows XP, Windows 7 and OSX

- Added support to Clang for OpenCL extension pragma callbacks
- Back-ported OpenCL type size fix from Clang trunk to our Clang 3.2 branch
- If everything goes well official Clang 3.4 will have all the features required by validator

Patches applied to Clang 3.2

- Added support to Clang for OpenCL extension pragma callbacks (sent patches for upstreaming them to Clang)
- Back-ported OpenCL type size fix from Clang trunk to our Clang 3.2 branch
- If everything goes well official Clang 3.4 will have all the features required by validator

Some driver differences / issues

Some compiler differences:

- AMD and Intel support "__local int array[]" as function parameter. POCL supports only "__local int *array". OpenCL requires only "__local int *array".
- Intel and POCL allow liberal access to vector fields: "v.x", "v[0]", "int i = 0; v[i]" and "(&v)->x". AMD supports only "v.x". OpenCL requires only "v.x".

Some compiler crashes:

- AMD crashes on "__constant struct { int v; __constant int *p; } c = { 0, &c.v };".
- AMD crashes on "__kernel void dummy(int d) { barrier(CLK_LOCAL_MEM_FENCE); }".
- Apple i5 and Intel driver compiler crashes with one test case

Some run-time crashes:

- Apple i5 driver allow running only kernels with local work group size 1

More information about issues in:

https://github.com/KhronosGroup/webcl-validator/blob/master/ISSUES.txt

Null pointers

- Algorithm requires that each address space with memory accesses has null pointer where invalid access will be directed
- Widest memory access of each address space is know at compile time
- When kernel starts we need to find out place which we can use as a null reference for the cases where user tries to access invalid memory

Builtins

- Some builtins access memory
- We know the behavior of all builtins by the specs
- We could replace the original builtin calls with ones that check all the limits before calling driver implementations
- Currently validation part allows builtin calls that are safe without any modifications like get_global_id()

Future

- Optimize limit checking order to make finding correct limit on the first try more probable
- Emulate stack to minimize overhead of private memory
- Research what effect putting private variables to struct has in register allocation level
- Make handling multiple kernels in one compilation module more efficient
- Integrate transformation to some driver
- Host side memory protection (e.g. zero init when getting buffers etc.)

Next steps

- Port to support Clang 3.3
- Implement support for all builtins, current support is really limited (might be needed for performance evaluation)
- Analysis of the major causes of performance degrading
- Extensive test set for the current implementation of typical OpenCL programs

  e.g. Parboil, Rodinia ViennaCL, LuxMark
- Performance results with all major driver implementations
- Evaluate how much those cases / general performance could be improved

Next step proposal

- Implement "*Safe OpenCL driver*" which is basically ICD Loader, which in addition to wrapping OpenCL API calls also does instrumentation
- It would validate given kernel arguments and add required size arguments
- Could select to use vendor's own memory protection instead of general source to source solution if some extensions are supported
- Khronos ICD implementation did seem pretty straight forward to start with

- Different WebCL browser implementations could directly use the "*Safe OpenCL driver*" instead of writing their own implementation
- Would also serve for other uses e.g. for shared environments / virtualization and not only for WebCL

- Implementation which looks like normal driver and allows any underlying vendor driver to be used would allow easy use of all current publicly available test / benchmarking suites and allow easy testing against all major vendor drivers

Rough estimates for some of the tasks

- Effort to get safe driver ready on Linux: 5 – 8 man months
  - Integrating instrumentation to Khronos ICD (kernel arg validation etc.) 1-2 months
  - Setting up test benches and ironing out problems found from different parts of pipeline (mostly finding things that are not working with current kernel instrumentation) 2 to many months
  - Adding good builtin support and good static analysis (to be able order limits checks better and to remove checks from constant indexing) 2-3 months
- Other tasks
  - Optimizing binary size hard way. Hack into Clang code base and build system and add flags which allow ignoring specific C++ parts of front-end 1 to many months.
  - Optimizing binary size even smaller (check if we can remove some libraries or find if there is tools, which can do global optimization and dead code elimination for statically linked binary e.g. llvm-gold linker) 2 weeks
  - Windows and OSX support for Khronos ICD (change build system to cmake, provide homebrew package etc. and add support to include OSX default driver in addition to ICD plugins) 1 - 2 months
  - Clang 3.3 and trunk porting 1 week
- Vincit is committed to continue investing on competence in compilers and OpenCL

vincit

Try it!

https://github.com/KhronosGroup/webcl-validator