vincit

Memory Protection Scheme for OpenCL

Mikael Lepistö, Vincit Ltd
2014-01-28

- We implemented kernel side memory protection for WebCL.
- How OpenCL people may benefit from this?

---

vincit

Vincit

Professionals for embedded systems, linux drivers, compiler development, mobile development, web services and more.

TECHNOLOGY FAST 50 FINLAND 2013
Deloitte Finland

#37 BEST WORKPLACE IN EUROPE & #5 BEST IN FINLAND 2013
Great Place To Work

COMPANY OF THE YEAR 2013
OP-Pohjola & Kauppalehti

GREAT PLACE TO WORK BEST 2013

Deloitte. Technology Fast500

- Close to 10 years old software vendor
- Does not hire people for specific projects, but people who love what they do
- Fastest growing software vendor in Finland 2013 (Deloitte)

vincit

Mikael Lepistö

Passionate Software Engineer @Vincit
mikael.lepisto@vincit.fi

Lead developer of WebCL C memory protection

Around 10 years experience on compilers and custom processor design toolset (@Tampere University of Technology)

Contributer in:
LLVM/Clang
Portable OpenCL Driver

---

vincit

Today

Background

Demo/how does it work?

What's next? Why should I care?

- We start by briefly going through where this project comes from.
- Next the implementation and performance is explained.
- In the last part we discuss why and how OpenCL community could benefit from kernel side memory protection.

vincit

2012 Nokia Research Center

2013 KHRONOS GROUP CONNECTING SOFTWARE TO SILICON WebCL

2014 WebCL ?

- 2012: We did project for NRC steered by Tomi Aarnio to explore and implement kernel side memory protection for WebCL.
- Project was Open Source.
- Found method that was efficient and simple and benefits from OpenCL C restrictions.
- 2013: WebCL working group sent out RFQ to implement WebCL C validation and protection. We participated and WebCL working group did like our proposal. Work started April and the first version meeting requirements of RFQ was ready on August.
- Continuation project to make feature complete version was approved by Khronos.
- 2014 January: Feature complete protection was finished.

---

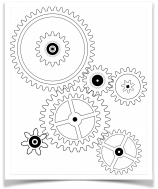
vincit

Two years of

- This is product of roughly 2 years of development.
- All the developers who has been part of the project has been praising how interesting project has been.
- I can't argue them, it has been really exciting!

vincit

How does it work?



Demo

Tips to make it run faster

Tools for integrators

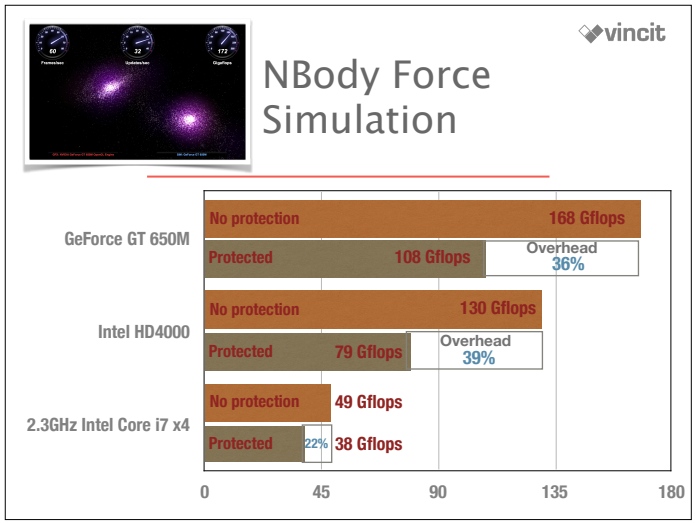
- First we check typical demo case of the protection and explain basic principles how protection works.
- Then we go through possibilities how to reduce overhead caused by protection.
- Finally tools that project offer are discussed.

vincit

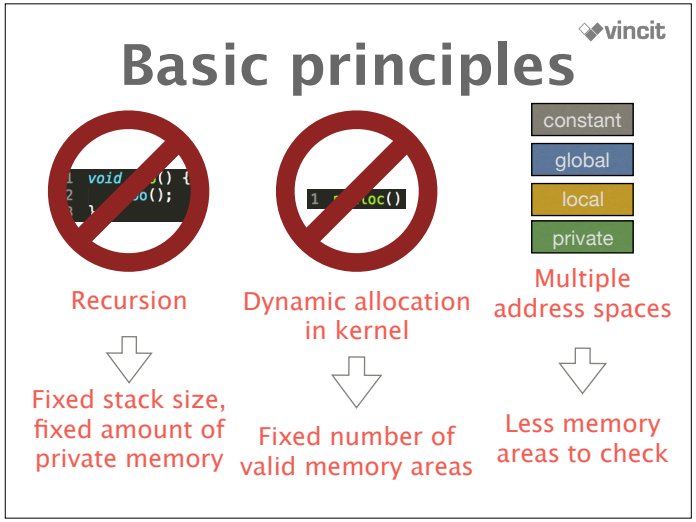
Demo 1

```
barrier(CLK_LOCAL_MEM_FENCE);
for (j = 0; j < tile_size; )
{
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);

    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
}
barrier(CLK_LOCAL_MEM_FENCE);
```

- Typical OpenCL kernel code, which has one local memory read for each computation helper function call.
- 8 times unrolled inner loop.



- Typical case of overhead caused by instrumentation.
- Basic implementation of the protection method.
- No special optimisations implemented.
- Not even the most simple static analysis is done to eliminate unnecessary checks / nor ordering of checks so that most probably area would be checked first.

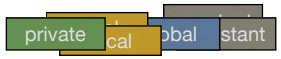


- If malloc/free are supported as pool allocation, it won't affect to method.
- If there is real dynamic allocation support, also that part of bookkeeping of memory areas must be implemented dynamically.
- No need to trace valid memory area for each pointer, but just collect all the valid limits when entering the kernel.

vincit

Big picture

Collect all the limits




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

Collect all the places where memory is accessed

Add CLAMP:ing to make sure that accesses are always inside limits.

```
CLAMP(addr, min1, max1, min2, max2, ...)
```

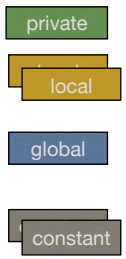
- When program enters the kernel, all valid memory areas are stored to bookkeeping structure.
- All the memory accesses are normalised to *(address) form, to make same type of clamping to work for everything.
- Protected memory access would look like *(CLAMP(address, valid_range_min, valid_range_max)).
- CLAMP(address, valid_range_min, valid_range_max) returns given address if in range or address to address space specific NULL which has valid alignment and size to fulfil safely any of the programs memory accesses.

---

vincit

How memory is allocated for different address spaces

```
constant int magic = 10;

kernel void example_kernel(
  global float* out,
  constan float* factors,
  local float* scratch) {

  local float static local memory[10];
  int index = get_global_id(0);
  // TODO: do something useful here ..
}
```



- Global memory: Can be only passed in kernel arguments.
- Constant memory: Can be passed in arguments or allocated statically.
- Private memory: Only static allocation possible.
- Local memory: Static allocation and may be passed to kernel.
- These are ALL of the valid memory areas, where program should access.

vincit

Add bookkeeping to kernel start

```
constant int magic = 10;

kernel void example_kernel(
   global float* out,
   constan float* factors,
   local float* scratch) {

   local float static_local_memory[10];
   int index = get_global_id(0);
   // TODO: do something useful here ..
}
```

Add arg sizes

```
constant int magic = 10;

kernel void example_kernel(
   global float* out, ulong out_size,
   constan float* factors, ulong factors_size,
   local float* scratch, ulong scratch_size) {

   local float static_local_memory[1
   int index = get_global_id(0);
   // TODO: do something useful here ..
}
```

Memory area bookkeeping

AS	start	end
constant	&magic	&magic + 1
constant	&factors[0]	&factors[size]
global	&out[0]	&out[size]
local	&scratch[0]	&scratch[size]
local	&sec…ratch[0]	&sec…ratch[size]
private	&index	&index + 1

- Now when we enter to kernel we know ALL the memory areas that program should be allowed to access.
- When kernel is started, instrumentation collects all the limits of separate memory areas to bookkeeping structure.
- To be able to know end of memory areas passed in kernel arguments, also size arguments must exist for buffers.
- We add second argument for each memory buffer that is passed, which tells element count.
- Size arguments cannot be given by end user, but WebCL host API implementation must set value correctly.

---

vincit

Memory accesses

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

*(address) ==> *(CLAMP(addr, min1, max1, min2, max2, …))

Instrumentation does not cause any restrictions to pointer arithmetics, casting etc.

- There might be need to do multiple memory limit checks if there are many limits in bookkeeping for the same address space.
- To reduce number of memory areas, we collect all the static allocations to structs, one per address space to minimise number of limit checks.
- Functionality of builtins is well defined, so if they are potentially dangerous, they are wrapped to safe versions, which first checks if call is safe before executing actual builtin.

vincit

Builtins

```
vload2(offset, f);
```

```
float2 _wcl_vload2_0(
_WclProgramAllocations* _wcl_allocs, size_t arg0, const __global float * arg1)
{
    const __global float * ptr = arg1 + 2 * (size_t) arg0;
    if (_wcl_addr_check_global_2_const___u_uglobal__float__Ptr(ptr, 2,
(const __global float *)_wcl_allocs->gl.builtin_wrappers__output_min,
(const __global float *)_wcl_allocs->gl.builtin_wrappers__output_max,
(const __global float *)_wcl_allocs->gl.builtin_wrappers__f_min, (const __global float
*)_wcl_allocs->gl.builtin_wrappers__f_max))
        return vload2(0, ptr);
    else
        return (float) 0;
}
```

```
_wcl_vload2_0(_wcl_allocs, _wcl_locals._wcl_offset, f);
```

- For builtin calls safe builtin implementations are generated
- Safe builtin version makes sure that no invalid memory will be accessed before calling the real builtin.

---

vincit

Remove gaps from statically allocated memory areas

```
float calc(int *one, int *two) {
    return *one + *two;
}

kernel void example(global int* out) {
    int index = get_global_id(0);
    int first = 1;
    int second = 2;
    out[index] = calc(&first, &second);
}
```

global | private

Number of limits is know compile time

Knowing correct address space allows to check less limits

```
typedef struct {
  int* out_min;
  int* out_max;
} GlobalLimits;

typedef struct {
  int first;
  int second;
} PrivateAddressSpace;

typedef struct {
  GlobalLimits        gl;
  PrivateAddressSpace pa;
} ProgramAllocations;

float calc(ProgramAllocations *pm,
    int *one, int *two) {
    return
        *(CLAMP(one, &pm->pa, &pm->pa + 1)) +
        *(CLAMP(two, &pm->pa, &pm->pa + 1));
}

kernel void example(global int* out, ulong out_size) {
    ProgramAllocations prog_mem_allocation = {
        { &out[0], &out[out_size]},
        {}
    };
    ProgramAllocations* pm = &prog_mem_allocation;
    int index = get_global_id(0);
    pm->pa.first = 1;
    pm->pa.second = 2;
    *(CLAMP(out+index, pm->gl.out_min, pm->gl.out_max)) =
      calc(pa, &first, &second);
}
```

- Compiler collects all the memory accesses and makes them to continuous area.
- Bookkeeping structure is initialised in the start of kernel (prog_mem_allocation variable).
- Bookkeeping structure is passed to helpers (ProgramAllocations *pm).
- Global memory access is normalised to *() format before clamping (was out[index]).
- Index variable is not collected to private address space, because there is no address reference to it.

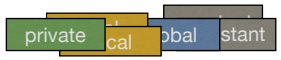
Actually it is not exactly so straight forward

```
// STRIPPED OUT SOME DEFINES + START COMMENT JSON DATA

typedef struct {
    int _wcl_first;
    int _wcl_second;
} __attribute__ ((aligned (WCL_ADDRESS_SPACE_private_ALIGNMENT))) _WclPrivates;

typedef struct {
    __global int *example__out_min;
    __global int *example__out_max;
} _WclGlobalints;

typedef struct {
    _WclGlobalints gi;
    __global uint *ga;
    _WclPrivates pa;
    __private uint *pa;
} _WclProgramAllocations;

__constant uint _wcl_constant_null[WCL_ADDRESS_SPACE_constant_MIN] = { 0 };

// STRIPPED OUT 50 LINES OF MACROS HERE

bool _wcl_addr_check_private_1_int__Ptr(int *addr, unsigned size, int * min0, int * max0) {
    return ( (!( ((addr) >= (min0)) && ((addr + size - 1) <= _WCL_LAST(int *, max0)) ));
}
int * wcl_addr_clamp_private_1_int__Ptr(int *addr, unsigned size, int * min0, int * max0, int * assull) {
    return _wcl_addr_check_private_1_int__Ptr(addr, size, min0, max0) ? addr : assull;
}

bool _wcl_addr_check_global_1__u_uglobal__int__Ptr(
    __global int *addr, unsigned size, __global int * min0, __global int * max0) {
    return (
        (!( ((addr) >= (min0)) && ((addr + size - 1) <= _WCL_LAST(__global int *, max0)) ));
}
__global int * wcl_addr_clamp_global_1__u_uglobal__int__Ptr(
    __global int *addr, unsigned size, __global int * min0, __global int * max0, __global int * assull) {
    return _wcl_addr_check_global_1__u_uglobal__int__Ptr(addr, size, min0, max0) ? addr : assull;
}


float calc(_WclProgramAllocations *_wcl_allocs, int *one, int *two) {
    return (*(_wcl_addr_clamp_private_1_int__Ptr((one), 1,
        (int *)(*_wcl_allocs->pa, (int *)((*_wcl_allocs->pa + 1), (int *)_wcl_allocs->pa))) +
        (*(_wcl_addr_clamp_private_1_int__Ptr((two), 1,
            (int *)(*_wcl_allocs->pa, (int *)((*_wcl_allocs->pa + 1), (int *)_wcl_allocs->pa))));
}


kernel void example(global int* out, ulong _wcl_out_size) {
    __local uint _wcl_local_null[WCL_ADDRESS_SPACE_local_MIN];
    _WclProgramAllocations _wcl_allocations_allocation = {
        { &out[0], &out[_wcl_out_size] },
        {
        },
        { },
    };
    _WclProgramAllocations *_wcl_allocs = &_wcl_allocations_allocation;
    _wcl_allocs->ga = WCL_SET_NULL(__global uint*,
        WCL_ADDRESS_SPACE_global_MIN, _wcl_allocs->gi.example__out_min, _wcl_allocs->gi.example__out_max,
        _wcl_global_null);
    if (_wcl_allocs->ga == (__global uint*)0) return;
    _wcl_allocs->pa = WCL_SET_NULL(__private uint*,
        WCL_ADDRESS_SPACE_private_MIN, &_wcl_allocs->pa, (&_wcl_allocs->pa + 1), (__private uint*)0);
    if (_wcl_allocs->pa == (__private uint*)0) return;

    int index = get_global_id(0);
    int first = (*_wcl_allocs->pa)._wcl_first = first;
    int second = (*_wcl_allocs->pa)._wcl_second = second;
    (*(_wcl_addr_clamp_global_1__u_uglobal__int__Ptr((out)+(index), 1,
        (__global int *)_wcl_allocs->gi.example__out_min,
        (__global int *)_wcl_allocs->gi.example__out_max,
        (__global int *)_wcl_allocs->ga))) = calc(_wcl_allocs, &_wcl_allocs->pa._wcl_first,
                                                  &_wcl_allocs->pa._wcl_second);
}
```

- The same code instrumented by WebCL memory protector.
- 50-100 lines of macro declarations and comments were stripped from output.
- More information of real instrumentation phases will follows in next section.

Recap

Collect all the limits



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

Collect all the places where memory is accessed

Add CLAMP:ing to make sure that accesses are always inside limits.

CLAMP(addr, min1, max1, min2, max2, …)

- We know all the memory areas allocated for the kernel when entering kernel.
- No need to track pointers to be able to know valid limits of memory (compared to fat-pointers)
- No extra memory needed for storing valid limits of every pointer (compared to fat-pointers).
- No restriction to address calculations or pointer casting.

Stack smashing

- All memory writes, also those which are done by builtins are protected.
- Cannot write to area that is outside of min,max limits of address space.
- E.g. return address is stored outside of that area.

Stack starting from kernel exec

global min addr	Accessed only by instrumeted code (_WclProgramAllocations)
global max addr	
…	
global null pointer	
local min addr	
local max addr	
…	
local null pointer	
private as var 1	Stack variables that can be accessed through address ref.
private as var 2	
private as var 3	
…	
other private var 1	Can be accessed only as values there is never address reference to these.
other private var 2	
…	
return ptr	
…	

- As a side effect protects also from stack smashing.
- Green area is where the variables are places, which has potentially access through pointer. This can be written and read by user code.
- Rest cannot be written / read by user code (all accesses of private address space are clamped to green area)
- All builtins are protected so e.g. if we would have classic buffer overflow by strcpy, our safe implementation would prevent it from writing over address space limits.
- Because only green area can be accessed by user code or builtins, return address cannot be overridden.

How the overhead could be reduced?

- Less than 50% overhead is already really good starting point
- What coder can do to make it faster?

vincit

Coding conventions

Always read big block from memory

Don't use arrays in private address space

Constant length of loops and as simple indexing operations as possible

- Most importantly access memory in as big blocks as possible -> Less run time checks.
- These same guidelines should help you out with any OpenCL code.
- Constant length loops with simple indexing can make it possible to move checks from inside of loop to before loop (currently there is no this kind of optimisations).

---

vincit

Demo 1 Optimised

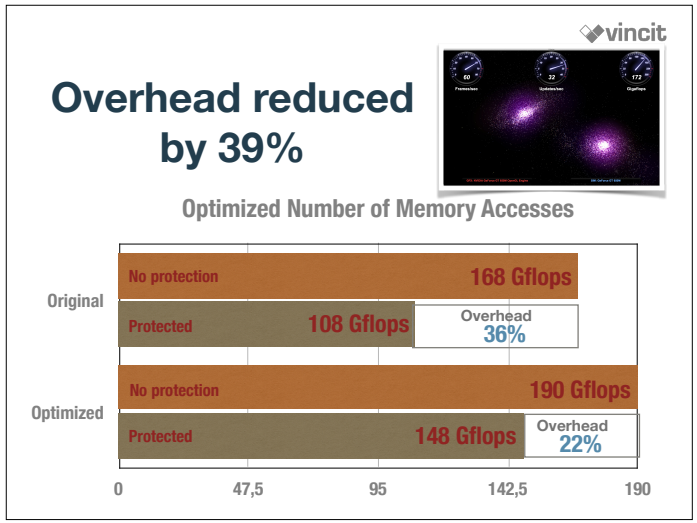
```
barrier(CLK_LOCAL_MEM_FENCE);
for (j = 0; j < tile_size; )
{
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);

    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
```

float4 access x8

```
barrier(CLK_LOCAL_MEM_FENCE);
for (j = 0; j < tile_size; )
{
    float16 smem1 = *(__local float16*)(shared_position + (j>>2));
    j+= 4;
    float16 smem2 = *(__local float16*)(shared_position + (j>>2));
    j+= 4;

    force = ComputeForce(force, smem1.s0123, position, softening_squared);
    force = ComputeForce(force, smem1.s4567, position, softening_squared);
    force = ComputeForce(force, smem1.s89ab, position, softening_squared);
    force = ComputeForce(force, smem1.scdef, position, softening_squared);

    force = ComputeForce(force, smem2.s0123, position, softening_squared);
    force = ComputeForce(force, smem2.s4567, position, softening_squared);
    force = ComputeForce(force, smem2.s89ab, position, softening_squared);
    force = ComputeForce(force, smem2.scdef, position, softening_squared);
}
barrier(CLK_LOCAL_MEM_FENCE);
```

float16 access x2

- Changed core loop of kernel
- Before: 8 times reading float4 from local memory
- After: 2 times read float16 from local memory
- The same amount of memory accessed, but in bigger blocks.



- Few lines in kernel core loop was changed.
- Original code runs 12% faster.
- Overhead was reduced by 39% (from 36% -> 22%).
- In the end overhead caused by protection was 22%

---

vincit

Demo 2

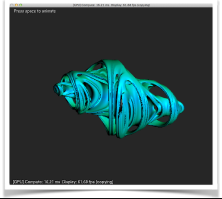
```
__kernel void
QJuliaKernel(
    __global uchar4 *result,
    const float4 mu,
    const float4 diffuse,
    const float epsilon)
{
    int tx = get_global_id(0);
    int ty = get_global_id(1);
    int index = ty * WIDTH + tx;
    float4 coord = (float4)((float)tx, (float)ty, 0.0f, 0.0f);
    float4 color = QJulia(coord, mu, diffuse, epsilon, ITERATIONS, SHADOWS, WIDTH, HEIGHT);
    uchar4 output = convert_uchar4_sat_rte(color * 255.0f);
    result[index] = output;
}
```

- Fractal, basically no input data just write to output.

vincit

QJulia

	Original	Protected	Overhead
Intel HD4000	61 fps	61 fps	0%



- Virtually no overhead.
- Most of the initialisation code is optimised away by vendor driver.
- Just one global memory write was instrumented by protection.
- With coding conventions and better static analysis also more complicated cases would be possible to validate instrumentation time. We could e.g. resolve buffer size requirements as a function of local and global size and check those before enqueuing kernel.

---

vincit

How can I try it out?

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

- Open source
- Progress has been and is all the time visible
- Sources, issues, instructions all in https://github.com/KhronosGroup/webcl-validator

vincit

Code from Khronos github

C API + static library.

Command line client

- Code in Khronos Github.
- C API + static library, extensive error reporting APIs.
- Command line client. Good for trying out how instrumented code looks like.
- Some tools for testing kernels that are used by test bench (kernel-runner is used to run kernels without need to write separate host code, opencl-validator compiles kernel code with all available drivers).

---

vincit

Supported platforms

Microsoft® Visual Studio®

- Build system is CMake based part of llvm/clang building.
- Testbench uses llvm's test framework that is python + shell script based.
- Currently we test this on Visual Studio, Mac and Linux.
- Build instructions are in github repository main page (README.md).
- Best test results are got with Intel CPU driver or Portable OpenCL driver (pocl).
- With other drivers usually some tests will fail e.g. due to not supported work group size etc.

vincit

Why should we care?

WebCL Host code is in JavaScript which is safe

C++ Host code is not safe anyways

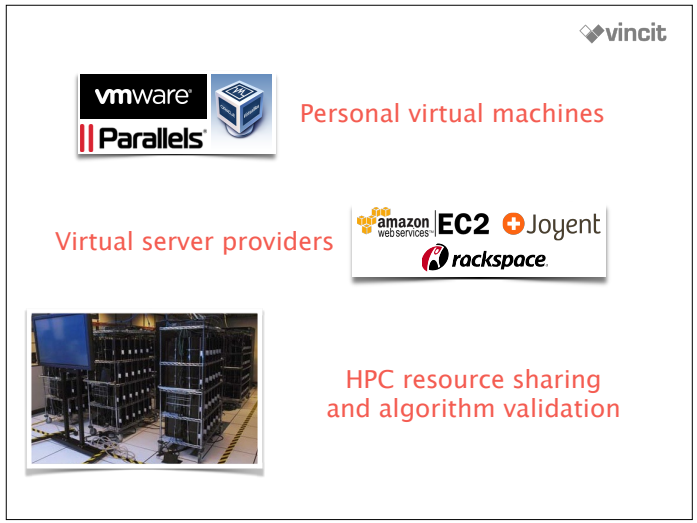
- Protection is currently done for WebCL.
- With OpenCL Host code is still usually in C++ which is not safe anyways.
- What use it would be to have protected OpenCL kernel code?

---

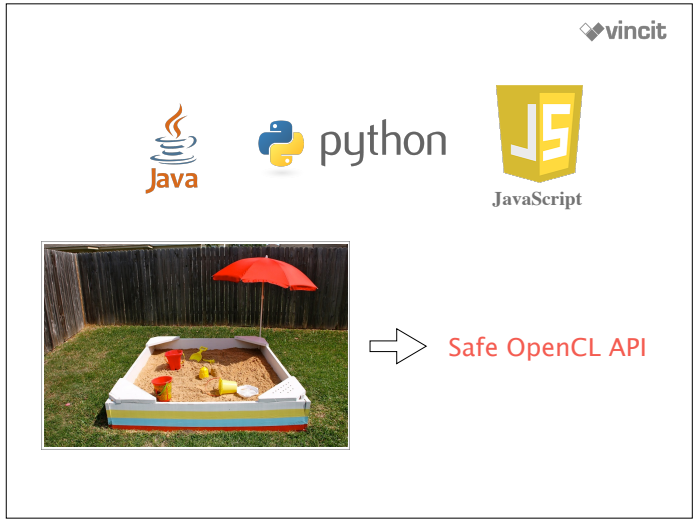
vincit

Few examples

- Next there are some use cases where protecting OpenCL could be beneficial.



- **Virtual Machines**: Normally I'm using Windows and Linux in virtual machine. If you are running C++ Host code in virtual machine it actually is safe. However now I cannot run OpenCL on GPU from there and if I want OpenCL support I need to boot natively to other operating system.
- **Virtual Server Providers**: Since virtualisation they have able to share CPU / memory / IO resources better between multiple server instances, but if you would like to provide cloud server with GPGPU support, you would need to have dedicated hardware for each client. Maybe there some special hardware which allows to separate processes, but it means that you cannot use the most cost effective consumer products of any vendor.
- **Cloud computing**: HPC clusters could benefit from this also by being able to share hardware resources better without need to worry about other processes break / spy on others that are running on the same cluster.



- In addition to kernel protection, **Safe OpenCL Host API** is needed to make sure that kernels are having correct type of arguments and memory buffers etc. by adding more sanity checks how OpenCL API is used.
- It could also apply protection to kennel code automatically and add extra kernel size arguments.
- Ideally Safe API would be perfectly compatible with normal OpenCL host API.
- Now everyone of these needs to make sure that OpenCL Host API is used safely and blacklist dangerous implementations etc.
- Safe OpenCL API could be also used by sandboxed "safe" languages.
- The same problem exist basically with every sandbox calling OpenCL. Like virtual machines.

vincit

Share insight

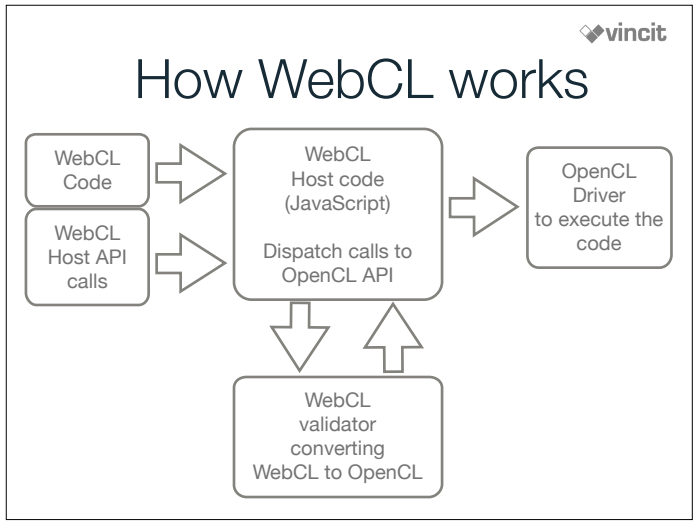
- Vendors and OpenCL users should have lot more end user feedback where protection could be beneficial.
- One use case raised was to use protection to make sure that **3rd party plugins** are not behaving badly.
- Validator / protection could be also used by developers to check if code has potential problems to help development.
- If protection overhead is small enough or even compile time SW vendor could use it to enforce safe coding conventions.
- End user could use Safe driver to be system default in case if he is having problems with some 3rd party OpenCL code crashing.

---

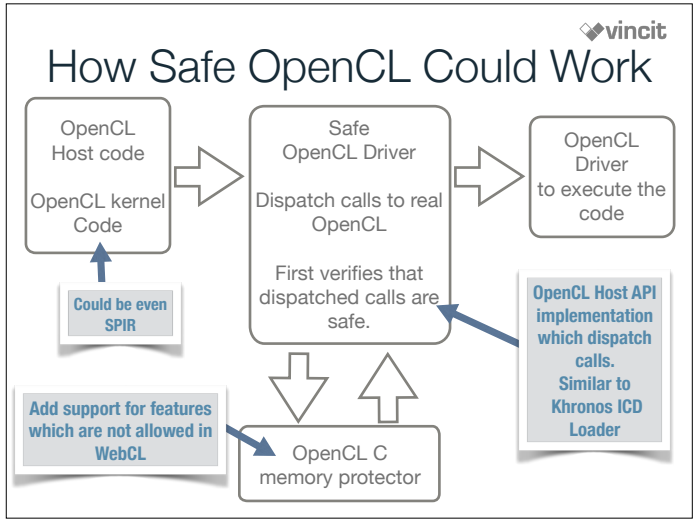
vincit

What we need to implement Safe OpenCL Driver?

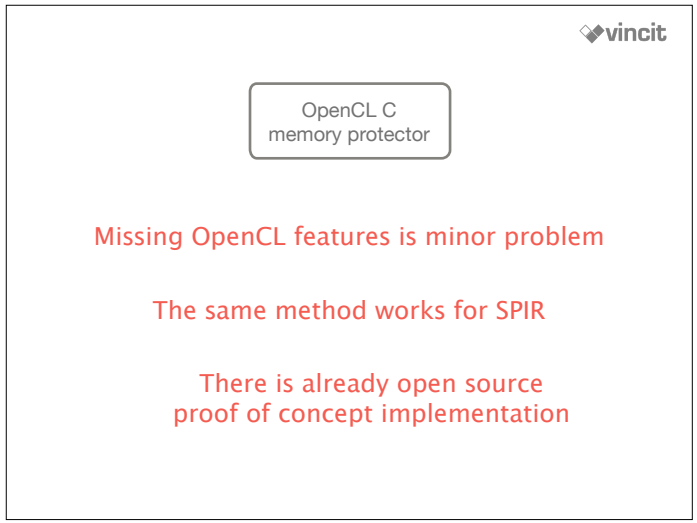
- Currently kernel protector is missing some features supported by OpenCL (WebCL is OpenCL subset)
- Safe host API is not implemented at all.



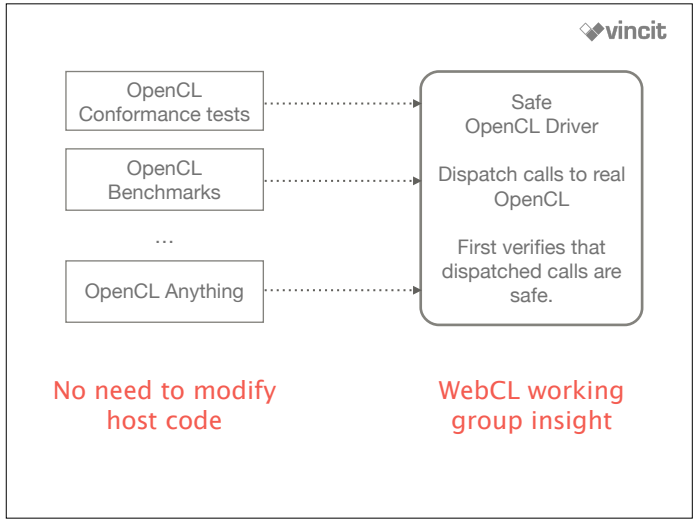
- With WebCL protection works like in diagram.
- WebCL host API code runs validations and applies protection.
- Host code adds extra kernel size arguments.
- Runs protected code with vendor driver.



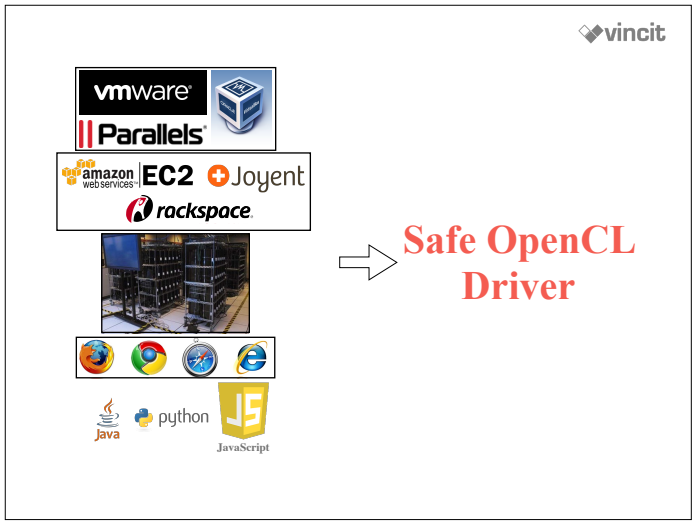
- Missing OpenCL features: Rest of OpenCL C 1.1 features, then 1.2 and pretty soon also OpenCL C 2 features.
- SPIR: Currently instrumentation is done in source level. Would be better to apply it to SPIR soon and use Khronos OpenCL to SPIR frontend to support OpenCL from source.
- OpenCL Host code: Might need to check buffer alignments, types, access qualifiers, zero initialisations etc. Adds extra kernel arguments to pass buffer sizes to protected kernels.



- Missing features: Struct as a kernel arguments, #include directive and kernel to kernel calls.
- First finish 1.1 then apply 1.2 changes and then 2.0
- The same protection method can be applied to SPIR. In fact we already have proof of concept from NRC project (did apply protection to NVPTX and SPIR).
- We should provide front-end to produce SPIR (llvm works already really nice). There was some clang 3.2 based Khronos project for it.



- WebCL working group people has some insight about making API safe.
- Invisible for end user. Looks just like normal OpenCL driver.
- No need to change Host nor kennel code -> easy to run existing benchmarks and conformance tests to make implementation more robust.
- Driver could be implemented to Khronos ICD Loader by adding additional bookkeeping and checking how API calls are done before dispatching them to vendor implementation.



- Could be enabled as extension or could be separate safe driver.
- Could be linked statically programs, e.g. in OSX where ICD loader is not available.

- WebCL proved that it can be done.
- Even the most basic implementation without any static analysis to reduce number of run-time checks was fast.
- We would love to keep working on this to bring it to OpenCL.

vincit

Thank you

Mikael Lepistö
mikael.lepisto@vincit.fi
Vincit Ltd