vincit

WebCL Validator and Memory Protection

Mikael Lepistö, Vincit Ltd
2014-01-29

---

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

Features

Implementation

Usage

vincit

What was committed to deliver

Phase 1 (response to the RFQ Khronos sent):

✓ Local and private memory initialisations
✓ Verify restrictions from WebCL draft
✓ Unit test cases for every feature
✓ Command line client to do instrumentation
✓ Support for safe builtin function like get_global_id()

Phase 2a (there were some budget left from phase 1):

✓ Builtin support for image builtins and vstore/vload
✓ Expose kernel arguments as JSON is start comment of instrumented code

Phase 2b (continuation project) – Critical Features by end of January 2014:

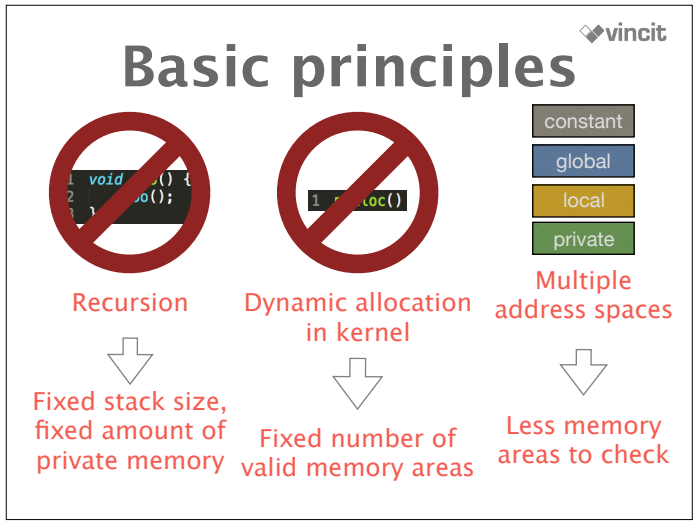
✓ Complete builtin support
✓ Library API for integrating validator
✓ Update original LLVM/Clang 3.2 to 3.4

---

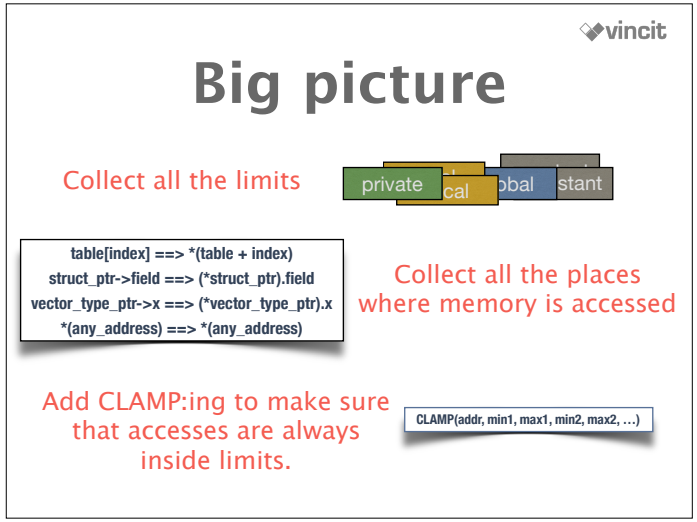
vincit

Some Features

- Parse and validate WebCL C against spec
- Extensive error messages
- LLVM/Clang 3.4 source to source transformation
- C API + static library
- Optimised validation time performance
- Extensive test bench
- Adds size arguments for buffers passed to kernel
- Collect and exposes information about kernel argument types
- Forces private and local memory initialisation
- Run time checks to make all memory accesses safe
- Automatic generation for safe builtins, to protect from potentially dangerous builtin invokes
- Check sampler_t usage to be correct at instrumentation time
- Complete suport for WebCL builtins
- Collection of statically allocated variables of address spaces to reduce number of valid memory areas
- Reorganisation of type declarations to make them available for address space structures
- Renaming of moved variables to prevent collision of variables of different scopes by the same name
- Find biggest memory access and memory alignment requirements for each address space
- Finds and adds code to select valid area for each address space null pointer
- Adds arguments to all helper functions to be able to pass memory limit structure to be available in helper functions
- Restricted goto statement
- Instrumentation time generation for program specific memory clamping macro
- Restricted image support only for image2d_t
- Restricted identifier names to 256 characters



- 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.



- 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 ..
}
```

private

local

global

constant

- 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[10];
   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

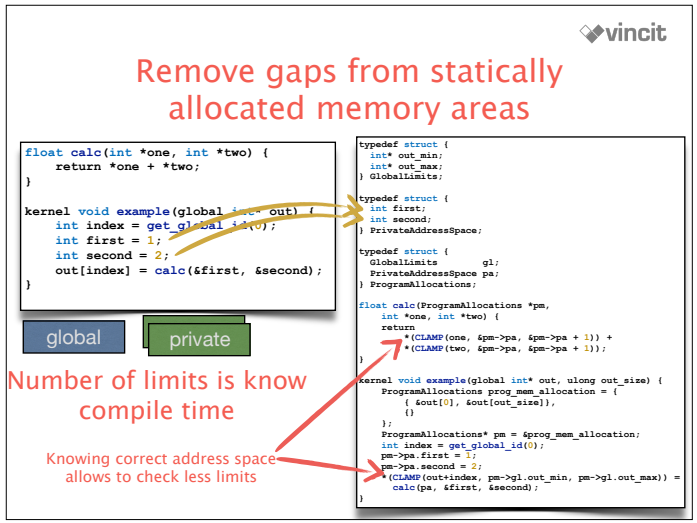
Builtins

```
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.



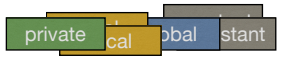
- Compiler collects all the memory accesses and makes them to continuous area.
- Bookkeeping structure is initialised in the start of kernel (prog_mem_allocation variable).
- Bookkeeping structure is passed to helpers (ProgramAllocations *pm).
- Global memory access is normalised to *() format before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

- 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.

vincit

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, ...)
```

---

vincit

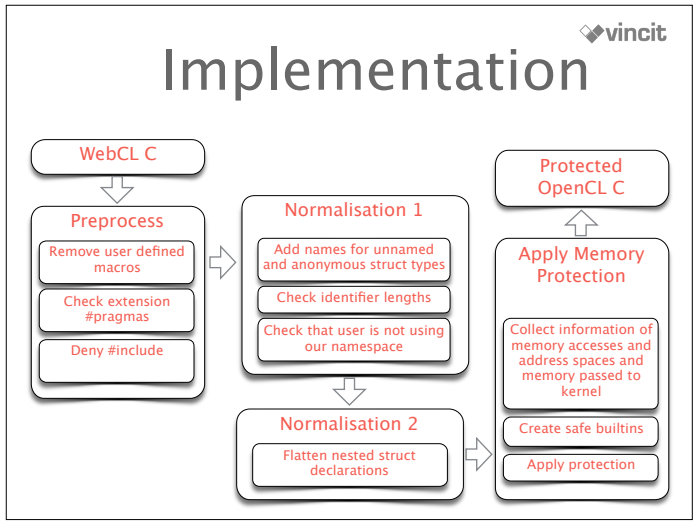
Stack smashing

- 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.

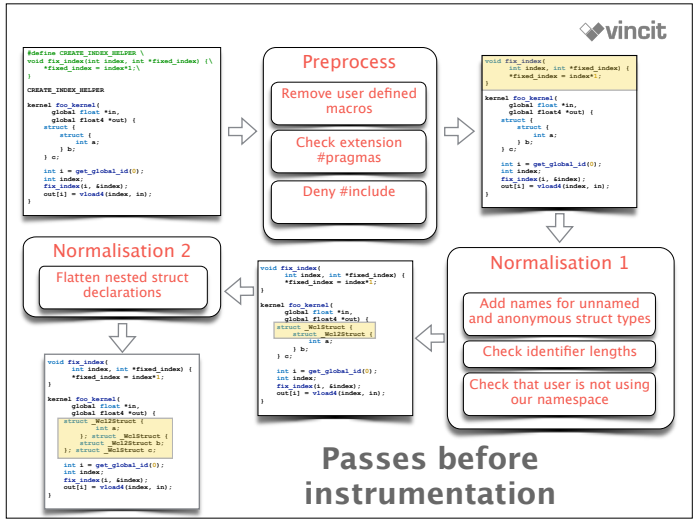
Stack starting from kernel exec

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	
…	

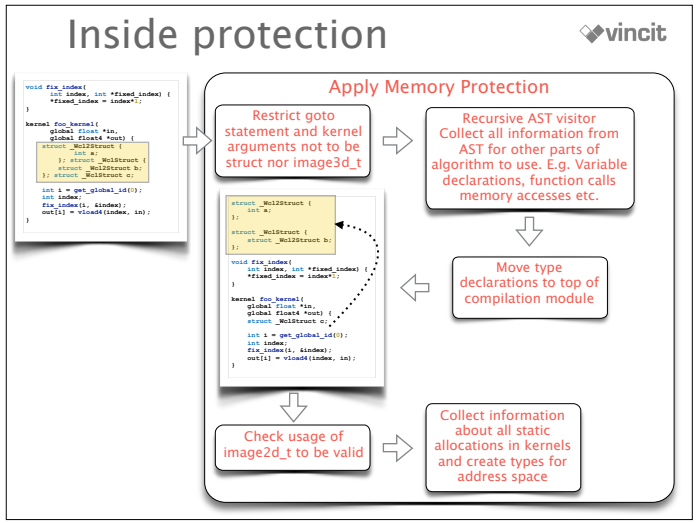
- Green area is where the variables are places, which has potentially access through pointer. This can be written and read by user code.
- Rest cannot be written / read by user code (all accesses of private address space are clamped to green area)
- All builtins are protected so e.g. if we would have 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.



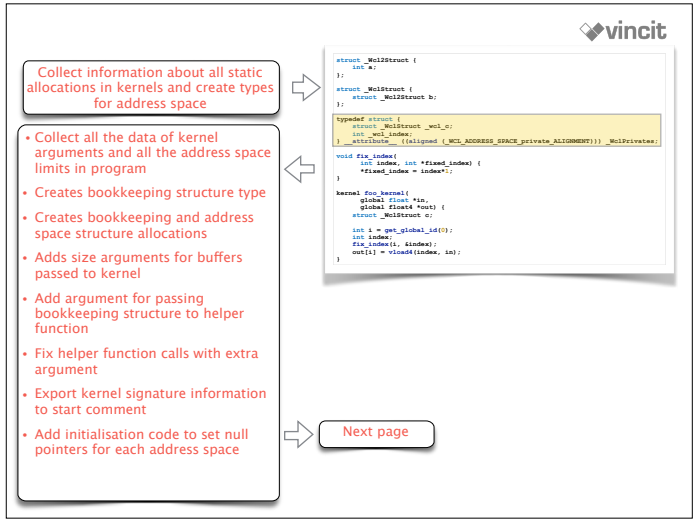
- In high level validation and protection is implemented in 4 passes.
- Preprocessor, Normalisation 1, Normalisation 2, Protector.
- Source is re-written and parsed again between each pass.



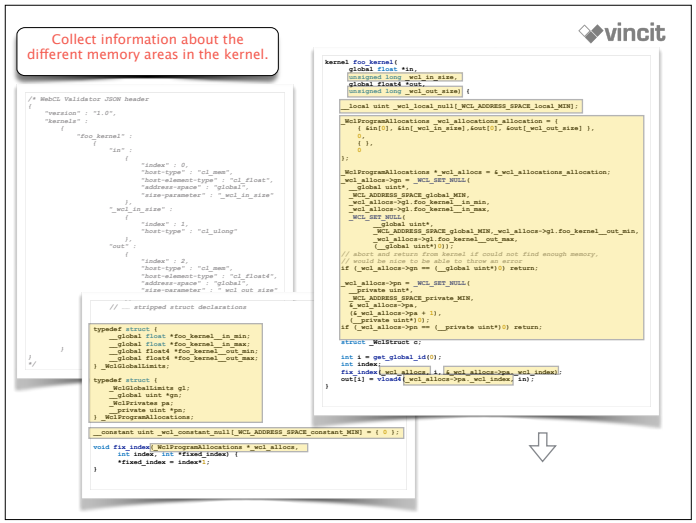
- Yellow parts are the changes done in code.
- Normalisation passes are important to make final protection pass more simple.



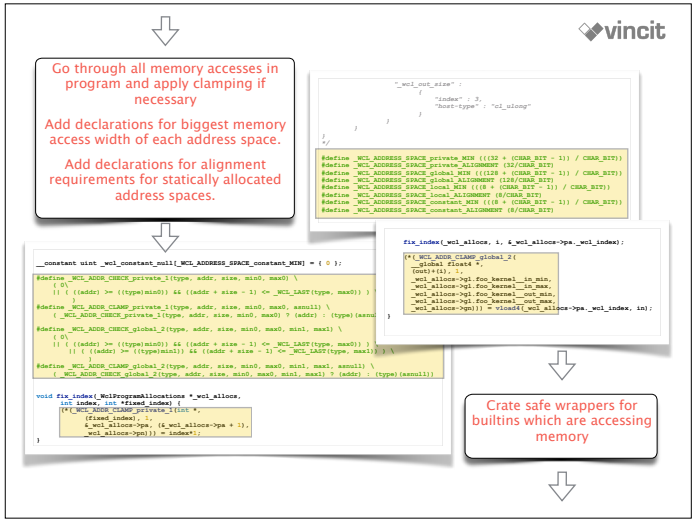
- Inside memory protector pass source code is not re-written / re-parsed.
- Declarations must be in early in compilation module to be able to use all the types in address space structures.



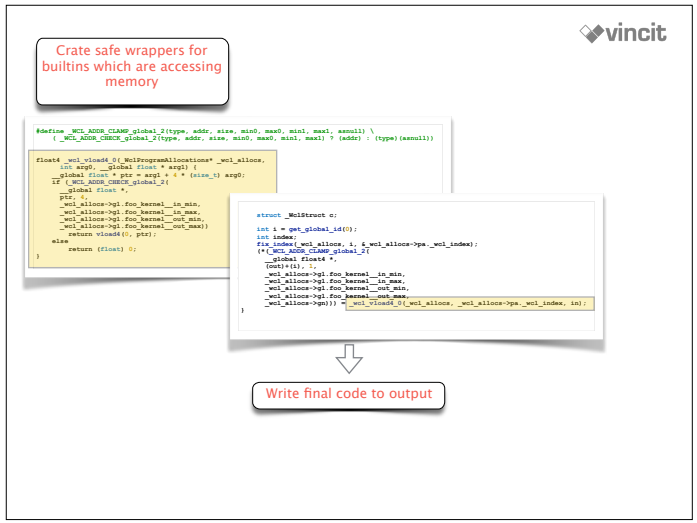
- Information about address spaces is collected.
- Type declarations for address spaces are created.



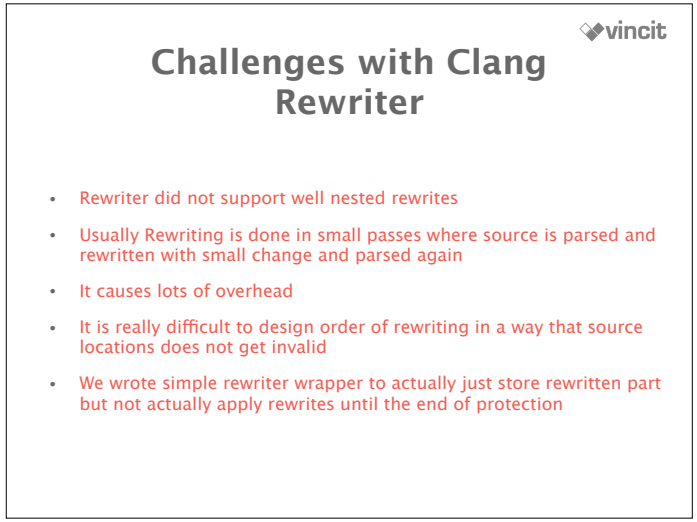
- Data about kernel arguments is collected.
- Limit bookkeeping structure type declarations are created.
- Constant address space is allocated.
- Bookkeeping struct argument is added to helper functions.
- Size arguments are added to kernel argument list.
- Bookkeeping structure is allocated and initialised with memory limits.
- Code to find big enough memory areas to set null pointers for every address space is added to kernel initialisation.
- References to variables that are relocated to address space structure are fixed to refer address space struct fields (i.e. _wcl_alloca->pa._wcl_index).



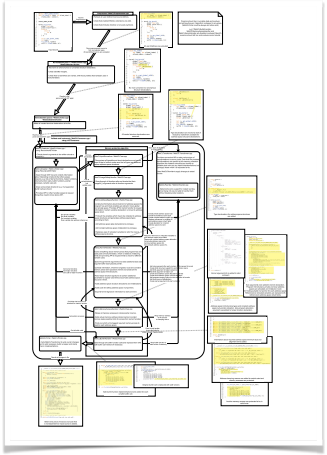
- Clamping macros are created.
- Memory protection calls is added to every memory access.
- Biggest memory access size of every address space is resolved.
- Alignment requirements for address space allocations are resolved.



- Find all builtin calls and create safe versions of them if necessary.
- Fix calls to builtins to use safe versions.
- Write the final code to file / string.



- With clang-3.2 rewriter didn't support complex rewriting.
- We added own wrapper for clang rewriter to allow some nested rewrites.
- When our rewriter applies changes it first internally flattens nested rewrites, so that we don't have to do any overlapping rewrites with clang.



vincit

Detailed flow chart is in github doc folder:

- different program modules
- Usage of Clang classes
- responsibilities of different passes
- communication between modules

https://github.com/KhronosGroup/webcl-validator/blob/master/doc/InstrumentationFlowChart.pdf

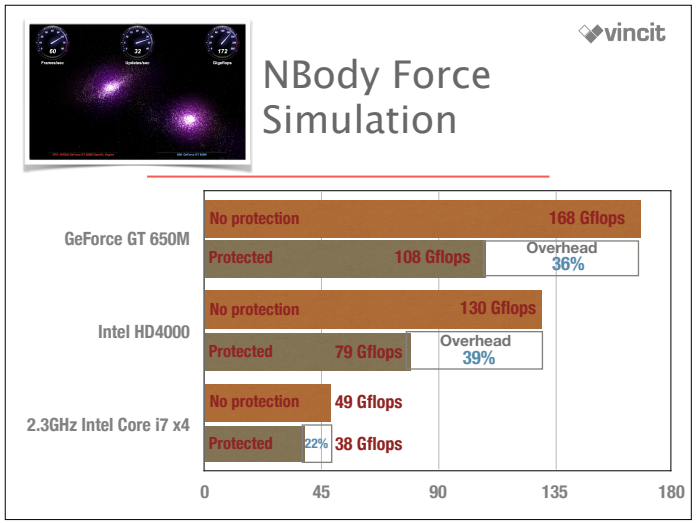
vincit

Demo 1

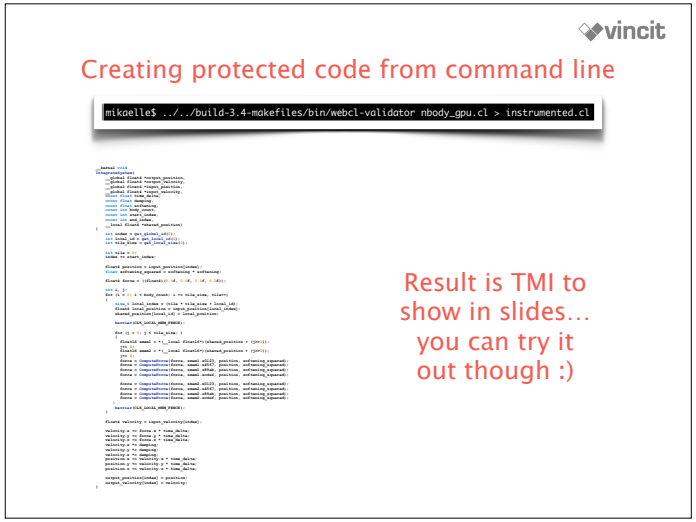
```
barrier(CLK_LOCAL_MEM_FENCE);
for (j = 0; j < tile_size; )
{
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);
    force = ComputeForce(force, shared_position[j++], position, softening_squared);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

- 8 time unrolled loop that has one float4 load on every call to helper function.



- 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.



- WebCL validator can be executed from command line
- We offer also library API, which command line client is also using internally.

vincit

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).

Demo 1 Optimised

vincit

```
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);
```

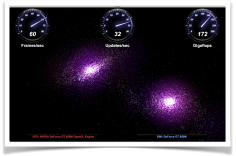
float16 access x2

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

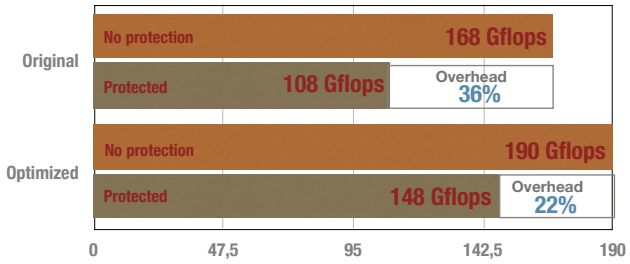
---

vincit

Overhead reduced by 39%



Optimized Number of Memory Accesses



- 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

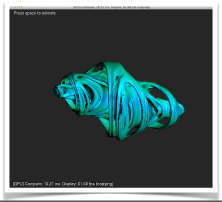
```
__kernel void
QJuliaKernel(
    __global uchar4 *result,
    const float4 mu,
    const float4 diffuse,
    const float epsilon)
{
    int tx = get_global_id(0);
    int ty = get_global_id(1);
    int index = ty * WIDTH + tx;
    float4 coord = (float4)((float)tx, (float)ty, 0.0f, 0.0f);
    float4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

- Apple's QJulia example

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

What was committed to deliver

Phase 1:
✓ Local and private memory initialisations
✓ Verify restrictions from WebCL draft
✓ Unit test cases for every feature
✓ Command line client to do instrumentation
✓ Support for safe builtin function like get_global_id()

Phase 2a:
✓ Builtin support for image builtins and vstore/vload
✓ Expose kernel arguments as JSON is start comment of instrumented code

Phase 2b – Critical Features by end of January 2014:
✓ Complete builtin support
✓ Library API for integrating validator
✓ Update original LLVM/Clang 3.2 to 3.4

Phase 2b – Rest of the budged
✘ Re-evalueate the best use for the remaining budget (was earlier allocated for some optimisations)

- Rest of the budged was decided to be used to find reduced test cases that causes our current test bench to fail with some vendor drivers.
- Also some additional checks to allowed extensions still need to be done. E.g. disable double type, if fp64 extension is not enabled.

---

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

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

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

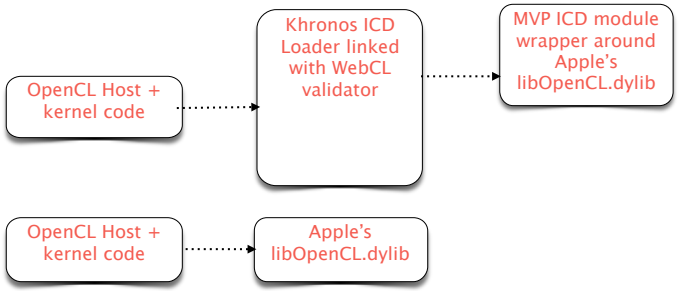
Supported platforms

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

Demo: Automatic instrumentation and fixing kernel arguments



- For integrating validator one could put validation calls for example to ICD Loader, which can afterwards call vendor driver.
- In next example the same test code (OpenCL Host and Kernel code) was first linked with vendor driver.
- Then the same case was linked with ICD loader, which automatically added protection.
- Demo showed that when code was linked with protection, program was not allowed to access invalid memory anymore.
- Protection applied automatically by modified ICD Loader. No changes in OpenCL host nor kernel code.

---

vincit

#include "clv/clv.h"

Link with:

liblibclv.a
libLLVMNVPTXCodeGen.a
libLLVMNVPTXDesc.a
libLLVMMCParser.a
libLLVMNVPTXInfo.a
libLLVMObjCARCOpts.a
libLLVMInstCombine.a
libLLVMipa.a
libLLVMTarget.a
libLLVMMC.a
libLLVMSupport.a
libclangASTMatchers.a
libclangFrontend.a
libLLVMOption.a
libLLVMBitReader.a
libclangParse.a
libLLVMObjCARCOpts.a
libLLVMInstCombine.a
libLLVMipa.a
libLLVMTarget.a
libLLVMAsmParser.a
libclangAnalysis.a
libclangLex.a
libLLVMMC.a
-lcurses

libLLVMAsmPrinter.a
libLLVMSelectionDAG.a
libLLVMNVPTXAsmPrinter.a
libLLVMCodeGen.a
libLLVMScalarOpts.a
libLLVMTransformUtils.a
libLLVMAnalysis.a
libLLVMCore.a
libLLVMObject.a
libclangTooling.a
libclangRewriteFrontend.a
libclangDriver.a
libclangSerialization.a
libclangRewriteCore.a
libclangSema.a
libLLVMScalarOpts.a
libLLVMTransformUtils.a
libLLVMAnalysis.a
libclangEdit.a
libLLVMCore.a
libclangAST.a
libclangBasic.a
libLLVMObject.a
-lcurses

- Currently only static linking is supported.
- One needs to link validator library and all the llvm libraries to which it depends.

vincit

Thank you

Mikael Lepistö
mikael.lepisto@vincit.fi
Vincit Ltd

- Special thanks to WebCL working group!