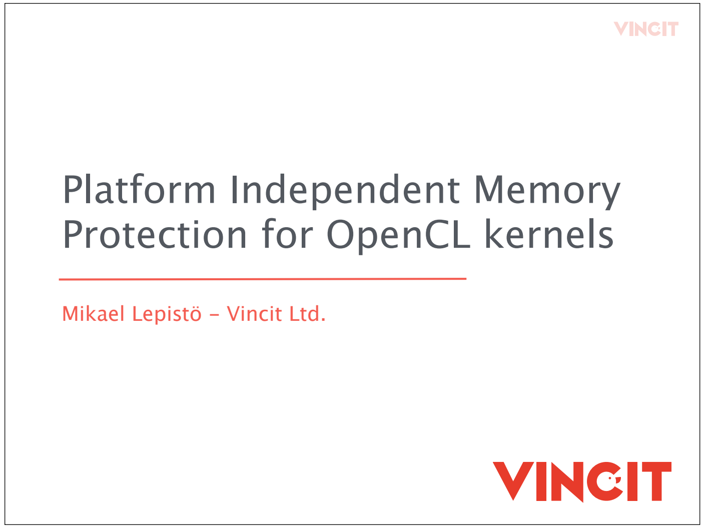
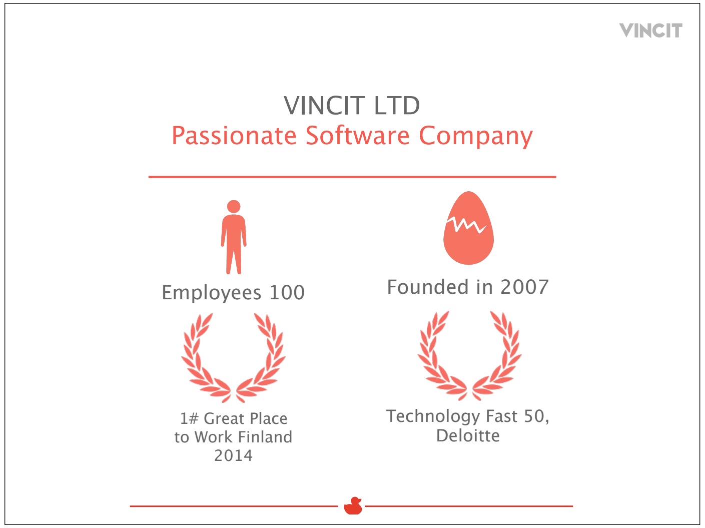


Platform Independent Memory Protection for OpenCL kernels

Mikael Lepistö – Vincit Ltd.

- We implemented memory protection for WebCL kernels
- Same kind of approach can be implemented for OpenCL as well
- It guarantees that when you are running kernel on CPU/GPU, it will not be able to access memory of any other "process" that is running on the same hardware.
- Makes sure that statically allocated and local memory is wiped before given to kernel.



VINCIT LTD

Passionate Software Company

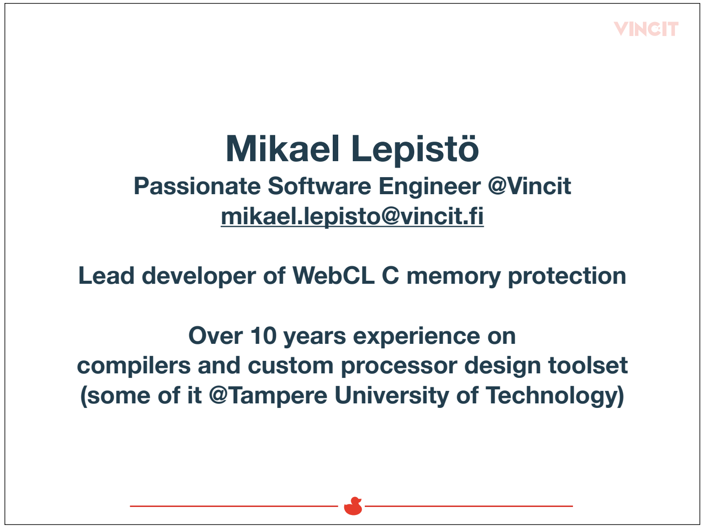
Employees 100

Founded in 2007

1# Great Place
to Work Finland
2014

Technology Fast 50,
Deloitte

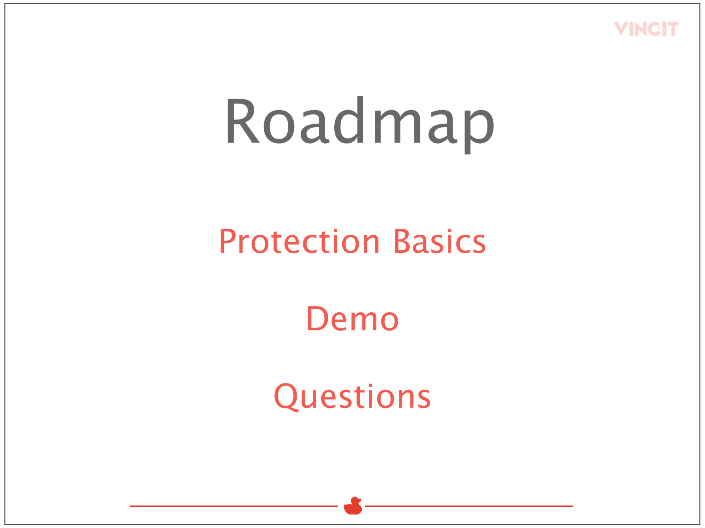
- Finnish software vendor with wide area of expertise (embedded, compilers, machine vision, mobile, web and more..)
- Puts lot of effort to keep customers and employees happy
- #1 Great Place to Work in Finland 2014
- Among the fastest growing software vendors in Finland
- you can trust the Black Duck



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

Lead developer of WebCL C memory protection

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



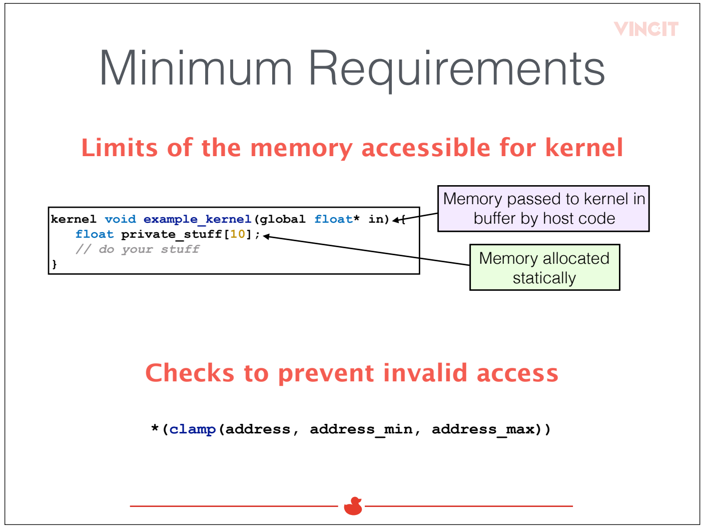
Roadmap

Protection Basics

Demo

Questions

- How we ended up with current approach
- Typical "Hey, our numbers of performance overhead are really good" - demo



Minimum Requirements

Limits of the memory accessible for kernel

```
kernel void example_kernel(global float* in) {
    float private_stuff[10];
    // do your stuff
}
```

Memory passed to kernel in buffer by host code

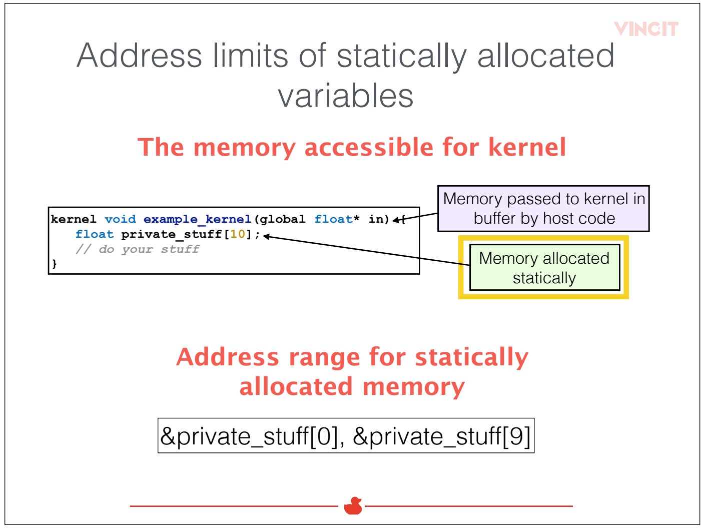
Memory allocated statically

Checks to prevent invalid access

```
*(clamp(address, address_min, address_max))
```

- We have to figure out which are the valid memory areas that kernel is allowed to access
- If we know ranges of valid addresses, checking is easy to do e.g. with clamp

---



Address limits of statically allocated variables

The memory accessible for kernel

```
kernel void example_kernel(global float* in) {
    float private_stuff[10];
    // do your stuff
}
```

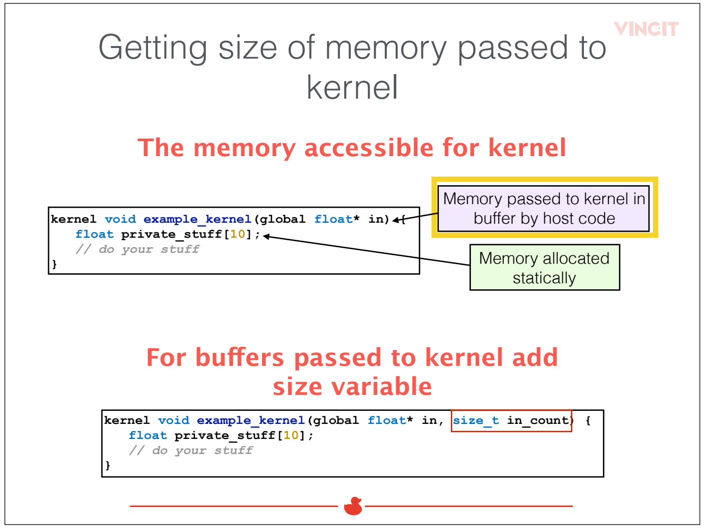
Memory passed to kernel in buffer by host code

Memory allocated statically

Address range for statically allocated memory

&private_stuff[0], &private_stuff[9]

- For static allocations we can easily find out the start and end addresses



Getting size of memory passed to kernel

The memory accessible for kernel

```
kernel void example_kernel(global float* in) {
   float private_stuff[10];
   // do your stuff
}
```

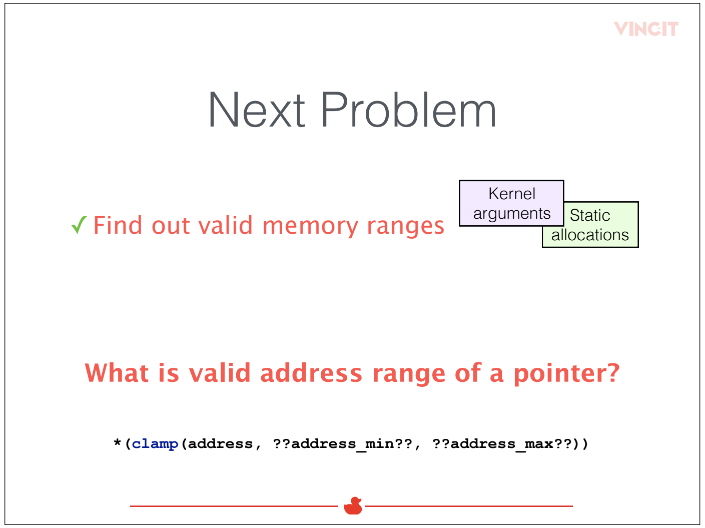
Memory passed to kernel in buffer by host code

Memory allocated statically

For buffers passed to kernel add size variable

```
kernel void example_kernel(global float* in, size_t in_count) {
   float private_stuff[10];
   // do your stuff
}
```

- For memory buffers passed to kernel we have to add an additional size argument
- Argument tells how many items there are allocated in the end of passed pointer
- Size argument is set by trusted party. E.g. by host code which is applying protection to kernel.



Next Problem

✓ Find out valid memory ranges

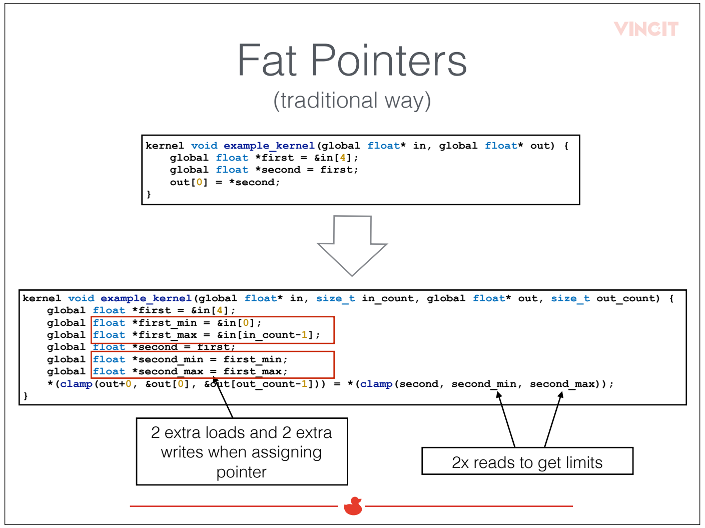
Kernel arguments

Static allocations

What is valid address range of a pointer?

```
*(clamp(address, ??address_min??, ??address_max??))
```

- Now we know all valid memory areas
- There might be complex pointer arithmetics done before memory access is done
- How we know when access is done which one of the all memory areas we should check?
- 1st try to do this we called Fat Pointers



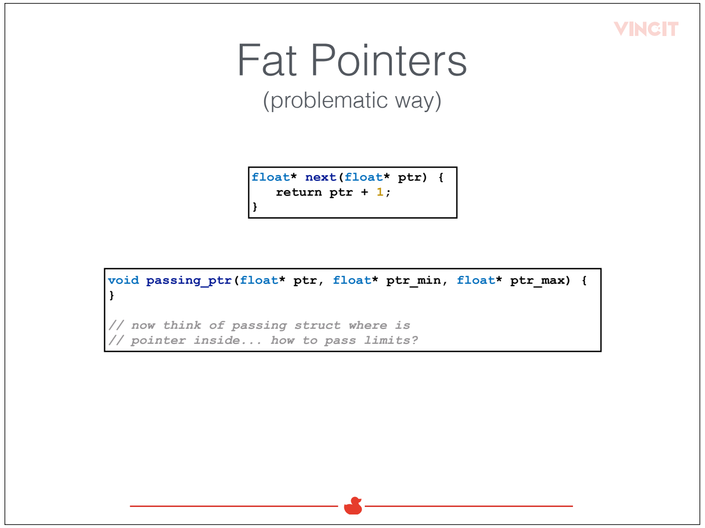
Fat Pointers

(traditional way)

```
kernel void example_kernel(global float* in, global float* out) {
    global float *first = &in[4];
    global float *second = first;
    out[0] = *second;
}
```

```
kernel void example_kernel(global float* in, size_t in_count, global float* out, size_t out_count) {
    global float *first = &in[4];
    global float *first_min = &in[0];
    global float *first_max = &in[in_count-1];
    global float *second = first;
    global float *second_min = first_min;
    global float *second_max = first_max;
    *(clamp(out+0, &out[0], &out[out_count-1])) = *(clamp(second, second_min, second_max));
}
```

- In this approach we added 2 extra variables for each pointer which were used to track correct limits
- This looks to cause quite heavy overhead, luckily most of the limits can be resolved during the instrumentation so it is possible to write efficient code also for this approach



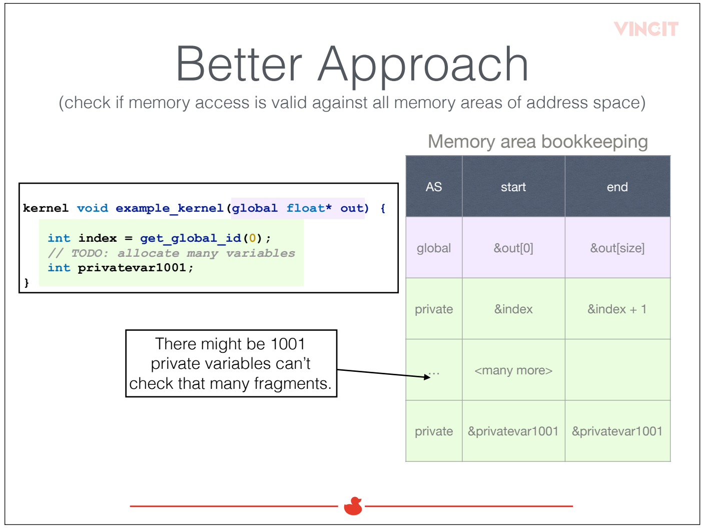
Fat Pointers

(problematic way)

```
float* next(float* ptr) {
    return ptr + 1;
}
```

```
void passing_ptr(float* ptr, float* ptr_min, float* ptr_max) {
}

// now think of passing struct where is
// pointer inside... how to pass limits?
```

- There are lot of corner cases which made implementation of this method really hard
- Every time when one passes pointer or returns pointer, also limits needs to be passed
- Casting between integers and pointers, limits are lost
- Tracking limits of pointer inside struct probably cannot be implemented this way
- Allocating array of pointers requires allocating 2 extra arrays to be able to track limits
- Aggressive inlining before applying protection to LLVM IR did help to many problem cases



Better Approach

(check if memory access is valid against all memory areas of address space)

```
kernel void example_kernel(global float* out) {

   int index = get_global_id(0);
   // TODO: allocate many variables
   int privatevar1001;
}
```

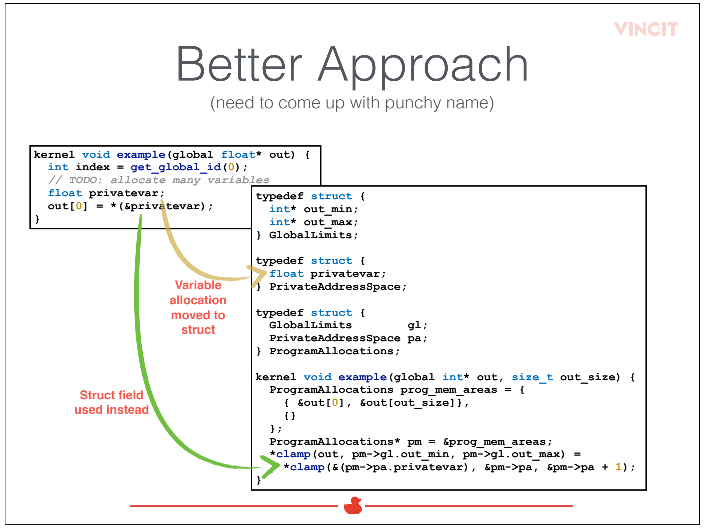
Memory area bookkeeping

AS	start	end
global	&out[0]	&out[size]
private	&index	&index + 1
…	<many more>	
private	&privatevar1001	&privatevar1001

There might be 1001 private variables can't check that many fragments.

- Fat Pointers did seem unnecessary hard
- We can collect bookkeeping of all the valid memory areas
- After that when memory is accesses we check against all the areas of that address space if it is inside one of the memory areas
- Usually global / local memory accesses are easy to track to correct limits instrumentation time
- Those private address space variables which might be accessed indirectly are collected to be one continuous memory area

---



Better Approach

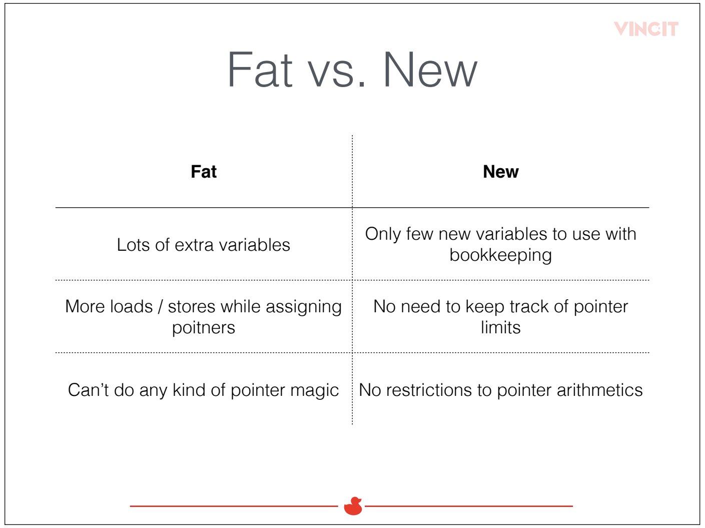
(need to come up with punchy name)

```
kernel void example(global float* out) {
  int index = get_global_id(0);
  // TODO: allocate many variables
  float privatevar;
  out[0] = *(&privatevar);
}
```

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

typedef struct {
  float privatevar;
} PrivateAddressSpace;

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

kernel void example(global int* out, size_t out_size) {
  ProgramAllocations prog_mem_areas = {
    { &out[0], &out[out_size]},
    {}
  };
  ProgramAllocations* pm = &prog_mem_areas;
  *clamp(out, pm->gl.out_min, pm->gl.out_max) =
    *clamp(&(pm->pa.privatevar), &pm->pa, &pm->pa + 1);
}
```

Variable allocation moved to struct
Struct field used instead

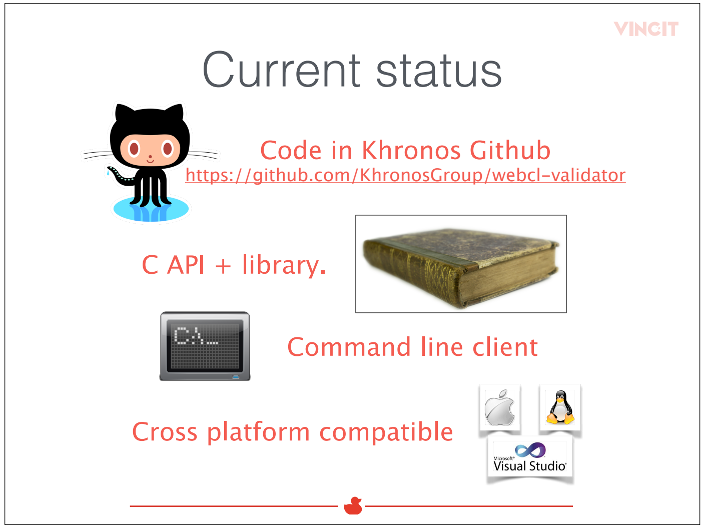
- In OpenCL C level one can make variables to be in continuous area by putting them to struct
- Bookkeeping struct is allocated in start of kernel
- Now tracing code is not necessary anymore, we just look places where are memory accesses and clamp them to be inside one of the valid memory areas, most of the time one check is enough



Fat vs. New

Fat	New
Lots of extra variables	Only few new variables to use with bookkeeping
More loads / stores while assigning poitners	No need to keep track of pointer limits
Can't do any kind of pointer magic	No restrictions to pointer arithmetics

- We got rid of most of the drawbacks of Fat Pointer approach



Current status

Code in Khronos Github
https://github.com/KhronosGroup/webcl-validator

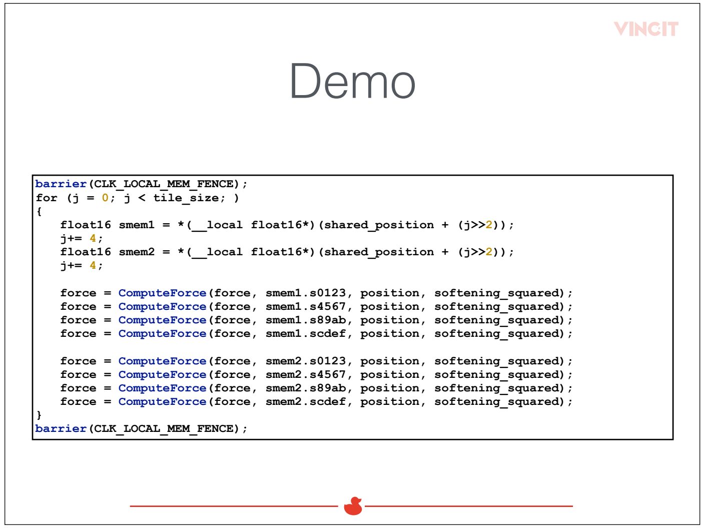
C API + library.

Command line client

Cross platform compatible

C:_
Visual Studio

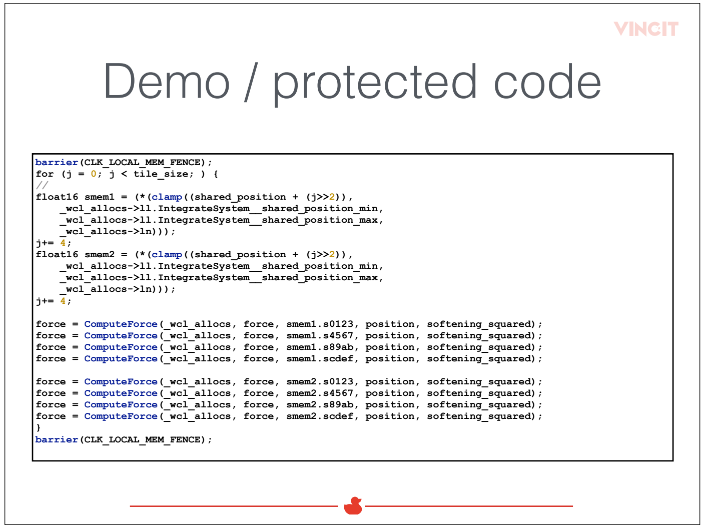
- We made Clang/LLVM based source to source transformation version of the method for Khronos WebCL working group
- Can be used with dynamic/static library or with command line client
- Tested with Visual Studio (Windows), Xcode (OSX) and gcc (Linux)



Demo

```
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);
```

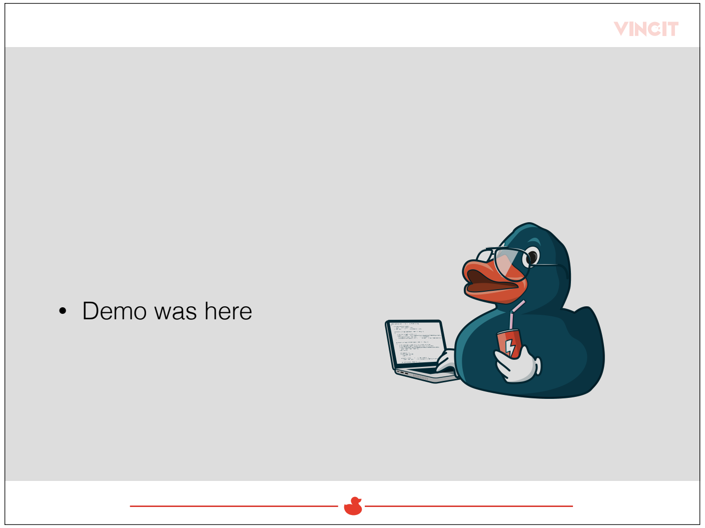
- Core loop was optimised by making as wide memory accesses as possible (float16 + splitting it to 4 parts for computation)



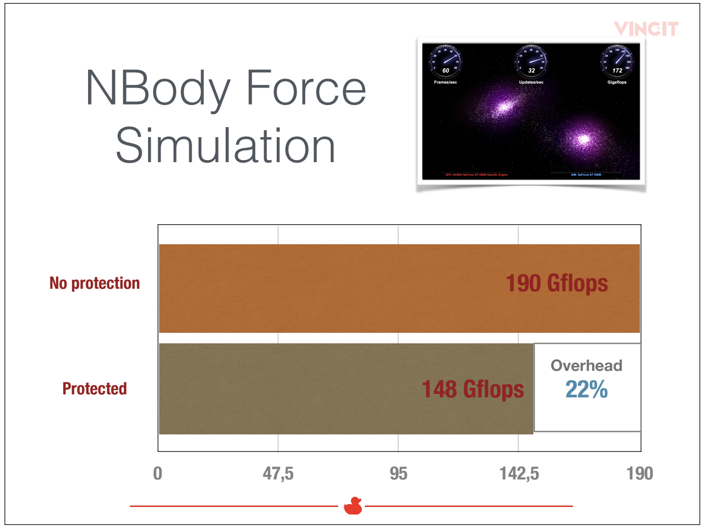
Demo / protected code

```
barrier(CLK_LOCAL_MEM_FENCE);
for (j = 0; j < tile_size; ) {
//
float16 smem1 = (*(clamp((shared_position + (j>>2)),
    _wcl_allocs->ll.IntegrateSystem__shared_position_min,
    _wcl_allocs->ll.IntegrateSystem__shared_position_max,
    _wcl_allocs->ln)));
j+= 4;
float16 smem2 = (*(clamp((shared_position + (j>>2)),
    _wcl_allocs->ll.IntegrateSystem__shared_position_min,
    _wcl_allocs->ll.IntegrateSystem__shared_position_max,
    _wcl_allocs->ln)));
j+= 4;

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

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

- Instrumentation added two extra clamping calls



- Demo was here



NBody Force Simulation

- In this case overhead was 22% on GPU
- If local memory would have been accessed in float4 pieces instead of float16 overhead would have been around 70%
- There is a lot of room for optimisations in memory protection algorithm, but already the simplest implementation of it performs really nicely
- Time to apply protection takes around 50-100ms with modern CPU

Thank you.

Mikael Lepistö
Passionate Software Engineer @ Vincit Ltd.
mikael.lepisto@vincit.fi

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

- More details about WebCL validator can be found from:
- https://github.com/KhronosGroup/webcl-validator/tree/master/doc