# *OPENCL™ C++*

**Lee Howes**
**AMD**
**Senior Member of Technical Staff, Stream Computing**

**Benedict Gaster**
**AMD**
**Principle Member of Technical Staff, AMD Research**
**(now at Qualcomm)**

# OPENCL™ TODAY

## WHAT WORKS, WHAT DOESN'T

# *THE PROBLEM TODAY*

- OpenCL™ out of the box provides:
  - C API
  - C Kernel language
- Excellent performance, but programming can be longwinded and difficult
- For example, to enqueue a kernel, the programmer must:

# *THE PROBLEM TODAY*

- OpenCL™ out of the box provides:
  – C API
  – C Kernel language
- Excellent performance, but programming can be longwinded and difficult
- For example, to enqueue a kernel, the programmer must:
  – Select a platform
  – Select a device
  – Create a context
  – Allocate memory objects
  – Copy data to the device
  – Create and compile the program
  – Create a kernel
  – Create a command queue
  – Enqueue the kernel for execution
  – Copy data back from the device

# EXAMPLE – VECTOR ADDITION (HOST PROGRAM)

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0,

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | C
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | C
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, s

// create the program
program = clCreateProgramWithSource(context, 1, &program_

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2], CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

Define platform and queues

ects

ernel

Execute the kernel

Read results on the host

**Ugly, isn’t it.**

**Extreme portability.**

**Luckily… Same boiler-plate code across virtually every OpenCL host code**

# ***OPENCL™ C++***

## *IMPROVING THE PROGRAMMING MODEL*

# OPENCL™ C++ FEATURES

- Khronos has defined a common C++ header file containing a high level interface to OpenCL

- It's much easier than using the C API, but it still needed work

- Improved C++ host API:
  - Interface for all OpenCL C API
  - Statically checked information routines (type safe versions of clGetInfoXX())
  - Automatic reference counting (no need for clRetain()/clRelease())

# *OPENCL™ C++ FEATURES*

- Khronos has defined a common C++ header file containing a high level interface to OpenCL

- It’s much easier than using the C API, but it still needed work

- Improved C++ host API:
  – Interface for all OpenCL C API
  – Statically checked information routines (type safe versions of clGetInfoXX())
  – Automatic reference counting (no need for clRetain()/clRelease())
  – Defaults (platform, context, command queues, and so on.)
  – Kernel functors
  – and more…

# INTERFACE FOR ALL OPENCL™ C API

- Single header file and inside a single namespace

```
#include <CL/cl.hpp>        // Khronos C++ Wrapper API
using namespace cl;
```

- Each base object in the OpenCL C API has a corresponding OpenCL C++ class
  - `cl_device_id` → `cl::Device`
  - `cl_platform_id` → `cl::Platform`
  - `cl_context` → `cl::Context`
  - etc…

- Unlike C API, multiple ways of constructing OpenCL objects are possible through object constructors

- Can still get to corresponding C object through operator()

# *AUTOMATIC REFERENCE COUNTING*

- OpenCL™ object lifetimes are explicitly managed through reference counting (retain, release)
  - Common source of program errors!
  - OpenCL C++ can do this implicitly through object destructors

```
cl_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);

err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

kernel = clCreateKernel(program, “vec_add”, NULL);

err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &memobjs[0]);
err  = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *) &memobjs[1]);
err  = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *) &memobjs[2]);


err = clReleaseKernel(kernel);
err = clReleaseProgram(program);
err = clReleaseMemObject(memobjs[0]);
err = clReleaseMemObject(memobjs[1]);
err = clReleaseMemObject(memobjs[2]);
err = clReleaseCommandQueue(cmd_queue)
err = clReleaseProgram(program);
```

**OpenCL C API reference counts objects**

**Explicit release is required**

# DEFAULTS

- OpenCL™ C++ introduces defaults for common use cases
  - Allows for a simple approach to writing basic applications
  - Excellent for beginners
- Provided defaults include:
  - Platform: simply pick the first one
  - Device: use the CL_DEVICE_TYPE_DEFAULT macro
  - Context: created from the default device
  - CommandQueue: created on the default device and context
- Each OpenCL C++ class includes a static member function:
  - `static Type getDefault();`
- Supports routines to set defaults, which have a transitive effect

## *KERNEL FUNCTORS*

- Current OpenCL™ interface for kernels is extremely verbose:

```
// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &memobjs[0]);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *) &memobjs[1]);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *) &memobjs[2]);

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size, NULL, 0, NULL, NULL);
```

- No guarantee of static type safety (both for arguments types or number of arguments)
- OpenCL C++ introduces kernel functors:
  – Type safe, directly callable kernels

```
std::function<Event (const EnqueueArgs&, Buffer, Buffer, Buffer)> vadd =
                    make_kernel<Buffer, Buffer, Buffer>(Program(program_source), “vadd”);

vadd(EnqueueArgs(NDRange(n)), memobj[0], memobj[1], memobj[2]));
```

# *KERNEL FUNCTORS*

- Current OpenCL™ interface for kernels is extremely verbose:

```
// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &memobjs[0]);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *) &memobjs[1]);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *) &memobjs[2]);

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size      , NULL, NULL);
```

**Kernel dispatch is just a function call**

- No guarantee of static type safety (both for arguments ty… or number of arguments)
- OpenCL C++ introduces kernel functors:
  - Type safe, directly callable kernels

```
std::function<Event (const EnqueueArgs&, Buffer, Buffer, Buffer)> vadd =
                  make_kernel<Buffer, Buffer, Buffer>(Program(program_source), “vadd”);

vadd(EnqueueArgs(NDRange(n)), memobj[0], memobj[1], memobj[2]));
```

# PUTTING IT ALL TOGETHER: OPENCL™ C++ VECTOR ADD

```
std::function<Event (const EnqueueArgs&, Buffer, Buffer, Buffer)> vadd =
            make_kernel<Buffer, Buffer, Buffer>(Program(program_source), “vadd”);

memobj[0] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcA);
memobj[1] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcB);
memobj[2] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n);

vadd(EnqueueArgs(NDRange(n)), memobj[0], memobj[1], memobj[2]));

enqueueReadBuffer(memobj[2], CL_TRUE, sizeof(float) * n, dest);
```

# *PUTTING IT ALL TOGETHER: OPENCL™ C++ VECTOR ADD*

```
std::function<Event (const EnqueueArgs&, Buffer, Buffer, Buffer)> vadd =
            make_kernel<Buffer, Buffer, Buffer>(Program(program_source), "vadd");

memobj[0] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcA);
memobj[1] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcB);
memobj[2] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float)

vadd(EnqueueArgs(NDRange(n)), memobj[0], memobj[1], memobj[2]));

enqueueReadBuffer(memobj[2], CL_TRUE, sizeof(float) * n, dest);
```

**Program automatically created and compiled**

# *PUTTING IT ALL TOGETHER: OPENCL™ C++ VECTOR ADD*

```
std::function<Event (const EnqueueArgs&, Buffer, Buffer, Buffer)> vadd =
            make_kernel<Buffer, Buffer, Buffer>(Program(program_source), "vadd");

memobj[0] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcA);
memobj[1] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcB);
memobj[2] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float)

vadd(EnqueueArgs(NDRange(n)), memobj[0], memobj[1], memobj[2]));

enqueueReadBuffer(memobj[2], CL_TRUE, sizeof(float) * n, dest);
```

**Program automatically created and compiled**

**Defaults, no need to reference context, command queue**

# PUTTING IT ALL TOGETHER: OPENCL™ C++ VECTOR ADD

```
std::function<Event (const EnqueueArgs&, Buffer, Buffer, Buffer)> vadd =
            make_kernel<Buffer, Buffer, Buffer>(Program(program_source), “vadd”);

memobj[0] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcA);
memobj[1] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcB);
memobj[2] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float)

vadd(EnqueueArgs(NDRange(n)), memobj[0], memobj[1], memobj[2]));

enqueueReadBuffer(memobj[2], CL_TRUE, sizeof(float) * n, dest);
```

**Program automatically created and compiled**

**Defaults, no need to reference context, command queue**

**No clRelease*XXX* cleanup code required**

# PUTTING IT ALL TOGETHER: OPENCL™ C++ VECTOR ADD

```
std::function<Event (const EnqueueArgs&, Buffer, Buffer, Buffer)> vadd =
            make_kernel<Buffer, Buffer, Buffer>(Program(program_source), “vadd”);

memobj[0] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcA);
memobj[1] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n, srcB);
memobj[2] = Buffer (CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float) * n);

vadd(EnqueueArgs(NDRange(n)), memobj[0], memobj[1], memobj[2]));

enqueueReadBuffer(memobj[2], CL_TRUE, sizeof(float) * n, dest);
```

# OPENCL™ C++

## KERNEL LANGUAGE ENHANCEMENTS

# OPENCL™ C++ ADDRESS SPACES

- OpenCL programming model includes address spaces to explicitly manage memory hierarchy
  - global, local, constant and private address spaces
  - Example:

```
kernel void foo(global int * g, local int * l)
```

- This mostly extends naturally to OpenCL C++, however:

```
struct Colour {
    int r_, g_, b_;
    Colour(int r, int g, int b);
};
kernel foo(global Colour& gColour) {
    Colour pColour = gColour;
}
```

# OPENCL™ C++ ADDRESS SPACES

- OpenCL programming model includes address spaces to explicitly manage memory hierarchy
  - global, local, constant and private address spaces
  - Example:

```
kernel void foo(global int * g, local int * l)
```

- This mostly extends naturally to OpenCL C++, however:

```
struct Colour {
    int r_, g_, b_;
    Colour(int r, int g, int b);
};
kernel foo(global Colour& gColour) {
    Colour pColour = gColour;
}
```

**C++ class member functions carry an implicit this pointer. What is it's address space? Note that there is an implicit copy constructor here.**

# OPENCL™ C++ ADDRESS SPACES CONTINUED

- The copy constructor's address space must be inferred

- First of all we have allowed the developer to specify the address space:

```
struct Shape {
    int setColour(Colour) global + local;
    int setColour(Colour) private;
};
```

- This would not have helped with the implicit copy constructor

# *OPENCL™ C++ ADDRESS SPACES CONTINUED*

- We have extended the type system to infer the address space from context
  - I'm keeping this brief because Ben Gaster will discuss this in more detail later in this session

- Use of auto and decltype allows even explicit *this* pointer use to work

```
operator (const decltype(this)& rhs) -> decltype(this)&
{
    if (this == &rhs) { return *this; }
    …
    return *this;
}
```

# OPENCL™ C++ ADDRESS SPACES CONTINUED

- We have extended the type system to infer the address space from context
  - I'm keeping this brief because Ben Gaster will discuss this in more detail later in this session

- Use of auto and decltype allows even explicit *this* pointer use to work

```
operator (const decltype(this)& rhs) -> decltype(this)&
{
    if (this == &rhs) { return *this; }
    …
    return *this;
}
```

**C+11 features used to handle cases when type of "this" needs to be written down by the developer.**

# OPENCL™ C++ ADDRESS SPACES CONT'D

- Further, we added the ability to parameterise on address spaces:

```
template<address-space aspace>
struct Shape {
   int foo(aspace Colour&) global + local;
   int foo(aspace Colour&) private;
   int bar(void);
};
```

# *OPENCL™ C++ ADDRESS SPACES CONT'D*

- Further, we added the ability to parameterise on address spaces:

```
template<address-space aspace>
struct Shape {
   int foo(aspace Colour&) global + local;
   int foo(aspace Colour&) private;
   int bar(void);
};
```

**Abstract over address space qualifiers.**

# OPENCL™ C++ ADDRESS SPACES CONT'D

- Further, we added the ability to parameterise on address spaces:

```
template<address-space aspace>
struct Shape {
   int foo(aspace Colour&) global + local;
   int foo(aspace Colour&) private;
   int bar(void);
};
```

**Abstract over address space qualifiers.**

**Methods can be annotated with address spaces, controls "this" pointer location. Extended to overloading.**

# OPENCL™ C++ ADDRESS SPACES CONT'D

- Further, we added the ability to parameterise on address spaces:

```
template<address-space aspace>
struct Shape {
   int foo(aspace Colour&) global + local;
   int foo(aspace Colour&) private;
   int bar(void);
};
```

**Abstract over address space qualifiers.**

**Methods can be annotated with address spaces, controls "this" pointer location. Extended to overloading.**

**Default address space for "this" is deduced automatically. Support default constructors.**

# SOME FUTURE EXTENSIONS

*NOT YET PUBLIC, BUT YOU NEVER KNOW*

# *WE'VE LOOKED AT POSSIBLE NEW FEATURES FOR THE FUTURE.*

- Smart pointers
  - To smoothly reuse code between discrete and shared memory infrastructures
  - To allow integration of specialized descriptors and similar features in the future

- Application of Æcute descriptors to OpenCL C++
  - Support optimised code generation through separate descriptions of execution and memory mappings

# *SMART POINTERS*

- A pointer type designed to work on current hardware
- Uses standard C++ design methodologies like custom allocators
  - Strongly typed; maintain locality; could store base and offset

```
cl::Pointer<int> x = cl::malloc<int>(N);
for (int i = 0; i < N; i++) {
    *(x+i) = rand();
}

std::function<
    Event (const cl::EnqueueArgs&,
    cl::Pointer<int>)> plus =
    make_kernel<
        cl::Pointer<int>, int>(
        “kernel void plus(global  Pointer<int> io)"
        “{int i = get_global_id(0);
         *(io+i) = *(io+i) * 2;}");
```

```
plus(EnqueueArgs(NDRange(N)), x);

for (int i = 0; i < N; i++) {
    cout << *(x+i) << endl;
}
```

# *SMART POINTERS*

**Construct pointer with specialized allocator**

- A pointer type designed to work on current hardware
- Uses standard C++ design methodologies like custom allo… s
  - Strongly typed; maintain locality; could st… base and offset

```
cl::Pointer<int> x = cl::malloc<int>(N);
for (int i = 0; i < N; i++) {
    *(x+i) = rand();
}

std::function<
    Event (const cl::EnqueueArgs&,
    cl::Pointer<int>)> plus =
    make_kernel<
        cl::Pointer<int>, int>(
        "kernel void plus(global  Pointer<int> io)"
        "{int i = get_global_id(0);
         *(io+i) = *(io+i) * 2;}");
```

```
plus(EnqueueArgs(NDRange(N)), x);

for (int i = 0; i < N; i++) {
    cout << *(x+i) << endl;
}
```

# *SMART POINTERS*

- A pointer type designed to work on current hardware
- Uses standard C++ design methodologies like custom allocators
  - Strongly typed; maintain locality; could store base and offset

```
cl::Pointer<int> x = cl::malloc<int>(N);
for (int i = 0; i < N; i++) {
    *(x+i) = rand();
}

std::function<
    Event (const cl::EnqueueAr
    cl::Pointer<int>)> plus
    make_kernel<
        cl::Pointer<int>, int>(
        “kernel void plus(global  Pointer<int> io)"
        “{int i = get_global_id(0);
         *(io+i) = *(io+i) * 2;}");
```

```
plus(EnqueueArgs(NDRange(N)), x);

for (int i = 0; i < N; i++) {
    cout << *(x+i) << endl;
}
```

**Assign and read pointer**

## SMART POINTERS

**Pass pointer to kernel functor.**

- A pointer type designed to work on current hardware
- Uses standard C++ design methodologies like custom allocators
  - Strongly typed; maintain locality; could store base and offset

```
cl::Pointer<int> x = cl::malloc<int>(N);
for (int i = 0; i < N; i++) {
    *(x+i) = rand();
}

std::function<
    Event (const cl::EnqueueArgs&,
    cl::Pointer<int>)> plus =
    make_kernel<
        cl::Pointer<int>, int>(
        “kernel void plus(global  Pointer<int> io)"
        “{int i = get_global_id(0);
         *(io+i) = *(io+i) * 2;}");
```

```
plus(EnqueueArgs(NDRange(N)), x);

for (int i = 0; i < N; i++) {
    cout << *(x+i) << endl;
}
```

# *SMART POINTERS*

- A pointer type designed to work on current hardware
- Uses standard C++ design methodologies like custom allocators
  - Strongly typed; maintain locality; could store base and offset

```
cl::Pointer<int> x = cl::malloc<int>(N);
for (int i = 0; i < N; i++) {
    *(x+i) = rand();
}

std::function<
    Event (const cl::EnqueueArgs&,
    cl::Pointer<int>)> plus =
    make_kernel<
        cl::Pointer<int>, int>(
        “kernel void plus(global  Pointer<int> io)"
        “{int i = get_global_id(0);
         *(io+i) = *(io+i) * 2;}");
```

```
plus(EnqueueArgs(NDRange(N)), x);

for (int i = 0; i < N; i++) {
    cout << *(x+i) << endl;
}
```

**Same type may be used in the kernel code.**

# SMART POINTERS

- Smart pointers can be used for allocating complex pointer-based data structures
  - Storing the buffer offset allows this to work

```
struct Node {
    int value;
    Pointer<Node> next;
};
Pointer<Node> createNode(int x) {
    Pointer<Node> result = malloc<Node>(1);
    result->value = x;
    result->next = Pointer<Node>();
    return result;
}
```

# SMART POINTERS

- Smart pointers can be used for allocating complex pointer-based data structures
  - Storing the buffer offset allows this to work

```
struct Node {
    int value;
    Pointer<Node> next;
};
Pointer<Node> createNode(int x) {
    Pointer<Node> result = malloc<Node>(1);
    result->value = x;
    result->next = Pointer<Node>();
    return result;
}
```

Specialized allocator used

# *ACCESS/EXECUTE DESCRIPTORS*

- Pointers are often a limitation to parallelism
  - Aliasing can be hard to prove
  - Analysing bounds of pointer accesses can be impossible
  - Computing data movement in advance is infeasible in the general case

- In loop nest, not in generated kernel
  - The loop nest may carry dependencies if we can analyse it
  - The generated kernel may have lost this inter-iteration information

- Various techniques partially address this
  - restrict
  - Parallelism guarantees such as the basic implicit vector parallelism provided by OpenCL kernels
  - Automated data movement may still be infeasible or unsafe

# *ACCESS/EXECUTE DESCRIPTORS*

- The decoupled access/execute model attempts to alleviate this
  - Separate the execution domain from its memory accesses
  - Declaratively specify as much as possible of the memory access patter to enable stronger optimisations

- We might, for example, specify two extended pointers and their metadata:

```
typedef cl::Pointer<int,
    cl::Access::Mapping<
        cl::Access::Project<100, 100>,
        cl::Access::Region<3, 3, cl::Access::Clamp>>
    AEcutePointerIN;
typedef cl::Pointer<float,
    cl::Access::Mapping<
        cl::Access::Project<100, 100>>
    AEcutePointerOUT;
```

# *ACCESS/EXECUTE DESCRIPTORS*

- The decoupled access/execute model attempts to alleviate this
  - Separate the execution domain from its memory accesses
  - Declaratively specify as much as possible of the memory access patter to enable stronger optimisations

Projection into 2D space

- We might, for example, specify two extended pointers and their meta[illegible]

```
typedef cl::Pointer<int,
    cl::Access::Mapping<
        cl::Access::Project<100, 100>,
        cl::Access::Region<3, 3, cl::Access::C[illegible]p>>
    AEcutePointerIN;
typedef cl::Pointer<float,
    cl::Access::Mapping<
        cl::Access::Project<100, 100>>
    AEcutePointerOUT;
```

# *ACCESS/EXECUTE DESCRIPTORS*

- The decoupled access/execute model attempts to alleviate this
  - Separate the execution domain from its memory accesses
  - Declaratively specify as much as possible of the memory access patter to enable stronger optimisations

- We might, for example, specify two extended pointers and their metadata:

```
typedef cl::Pointer<int,
    cl::Access::Mapping<
        cl::Access::Project<100, 100>,
        cl::Access::Region<3, 3, cl::Access::Clamp>>
    AEcutePointerIN;
typedef cl::Pointer<float,
    cl::Access::Mapping<
        cl::Access::Project<100, 100>>
    AEcutePointerOUT;
```

**Memory access is a 3x3 region around the projected centre.**

# ÆCUTE DESCRIPTORS

- We then use the pointers in a kernel:

```
kernel void plus( global const AEcutePointerIN in, global AEcutePointerOUT out) {
    int2 wid = (int2)(get_global_id(0), get_global_id(1));
    float sum = 0.f;
    for( int i = 0; i < 3; ++i ) {
        for( int j = 0; j < 3; ++j ) {
            sum += (float)in(j, i);
        }
    }
    out = sum / 9.f;
};
```

***i* and *j* directly iterate over the *region*, not the input addresses. The region may have been copied locally with no update to the addressing necessary**

## SOME ANALYSIS

*DID WE MAKE THINGS GO SLOWLY AND WHAT DID WE GAIN?*

# *PERFORMANCE*

- C++ commonly is thought to have a performance overhead
- It isn't the only myth out there, but it's the relevant one for today

# *PERFORMANCE*

- C++ commonly is thought to have a performance overhead

- It isn't the only myth out there, but it's the relevant one for today

- Performance measurements on the host code showed no difference from OpenCL™ C API
  - Not a bad thing!
  - Productivity, not performance enhancement

- Performance measurements on the device code show no difference from OpenCL C code
  - Not unexpected
  - Static C++ is essentially zero overhead

# CODE SIZE REDUCTION

- Sometimes substantial reduction in code size:

| Application | C lines | C++ lines | Reduction |
|---|---|---|---|
| Vector addition | 268 | 140 | 47.7% |
| Pi computation | 306 | 166 | 45.8% |
| Ocean simulation | 1386 | 533 | 61.5% |
| Particle simulation | 733 | 601 | 18.0% |
| Radix sort | 627 | 593 | 5.4% |

# CONCLUSION

- OpenCL™ C++
  – Productivity abstraction over OpenCL's C interfaces
  – Abstracts both host and device components

- Available today:
  – Downloadable header from www.khronos.org (supports OpenCL 1.2)
  – Header and OpenCL C++ kernel language support in AMD APP SDK 2.7
    - http://developer.amd.com

# Disclaimer & Attribution