# Other Programming Models for GPUs

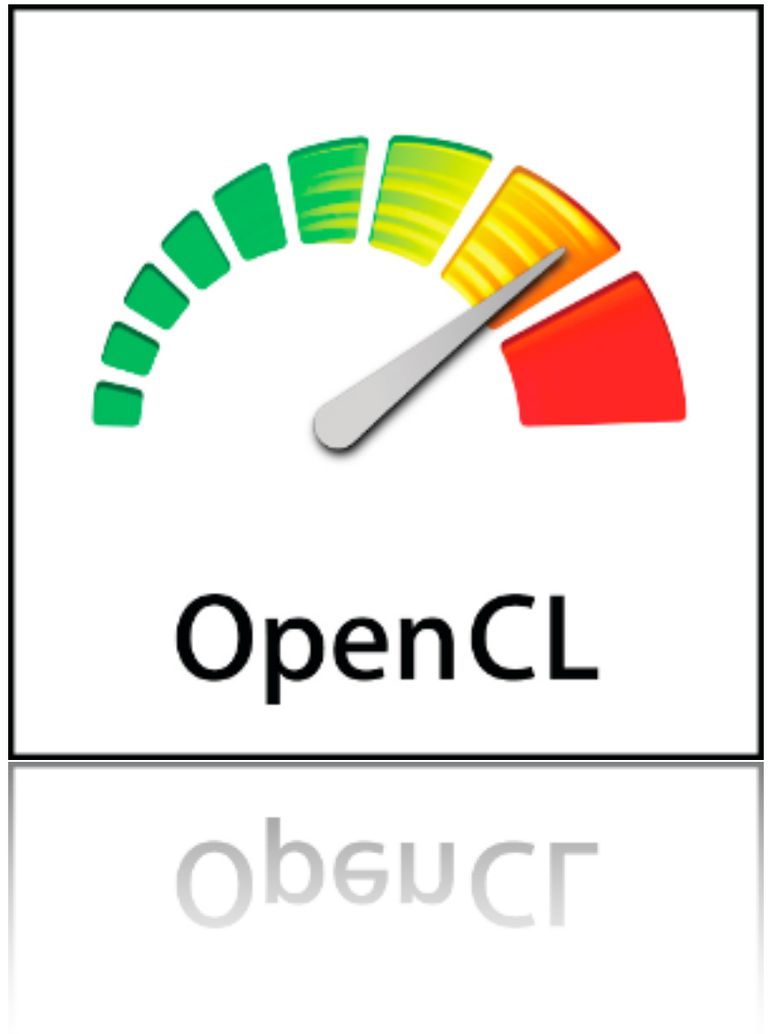
Fall 2012

CITA|ICAT

SciNet

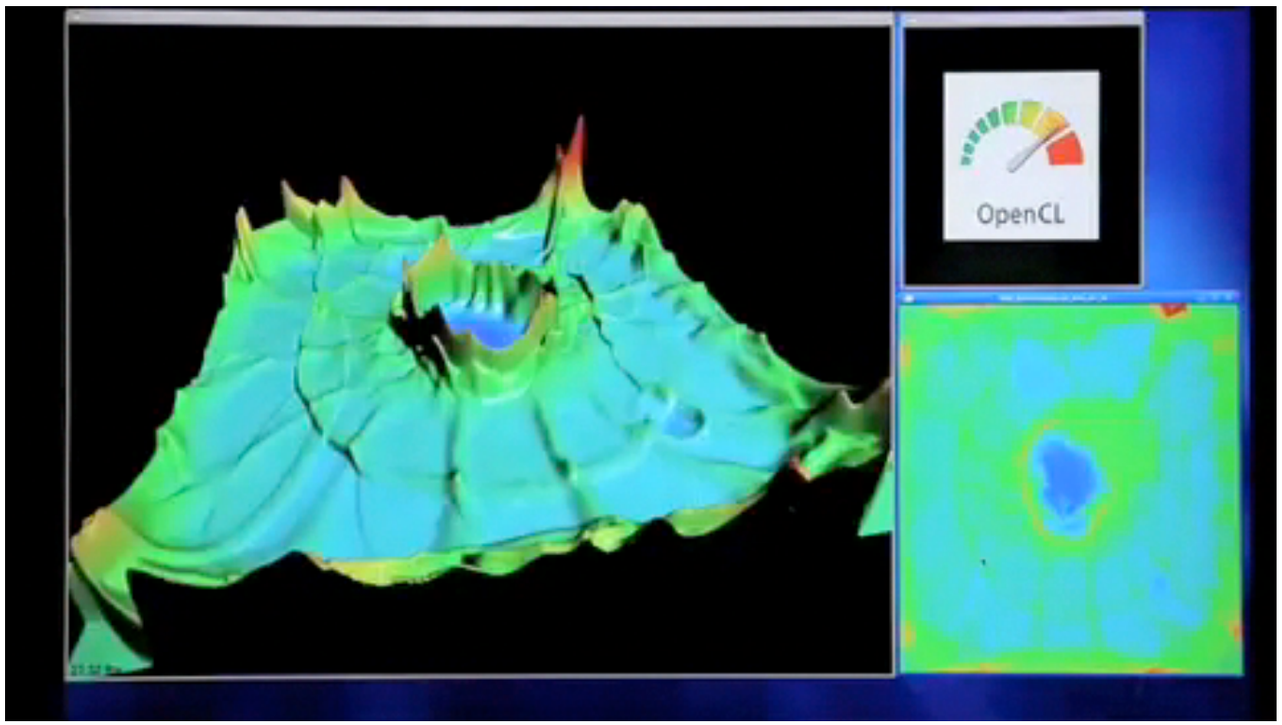
# OpenCL: Open standard

- Driven by Apple (comes standard in Snow Leopard)
- NVIDIA, AMD, Intel, IBM (Cell)
- Exposes a consistent, GPU-like interface to any multicore system



# Heterogeneous, Open

- Can work with various hardware
- IBM Cell, AMD processors, ATI cards, NVidia cards, Intel processors
- Multi- and Many- core
- SC09 demo: parallel CFD running on all of the above at once in *same program*, using MPI to tie them together



# All is not roses

- Code may be portable, but **performance** isn't
- (Autotuning?)
- For short-term future (~32 cores, cache-coherent), OpenMP is going to be the way to go for multicore CPUs

# All is not roses

- You pay for all that extra generality:

# Quick Reference Card

- Page 1

OpenCL API 1.1 Quick Reference Card - Page 1

**OpenCL (Open Computing Language)** is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices.

[n.n.n] refers to the section in the API Specification available at www.khronos.org/opencl.

## The OpenCL Runtime

### Command Queues [5.1]

cl_command_queue **clCreateCommandQueue** (
  cl_context *context*, cl_device_id *device*,
  cl_command_queue_properties *properties*,
  cl_int *errcode_ret*)

*properties:* CL_QUEUE_PROFILING_ENABLE, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE

cl_int **clRetainCommandQueue** (
  cl_command_queue *command_queue*)

cl_int **clReleaseCommandQueue** (
  cl_command_queue *command_queue*)

cl_int **clGetCommandQueueInfo** (
  cl_command_queue *command_queue*,
  cl_command_queue_info *param_name*,
  size_t *param_value_size*, void *param_value*,
  size_t *param_value_size_ret*)

*param_name:* CL_QUEUE_CONTEXT, CL_QUEUE_DEVICE, CL_QUEUE_REFERENCE_COUNT, CL_QUEUE_PROPERTIES

## The OpenCL Platform Layer

The OpenCL platform layer implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

### Contexts [4.3]

cl_context **clCreateContext** (
  const cl_context_properties *properties, cl_uint *num_devices*,
  const cl_device_id *devices, void (CL_CALLBACK*pfn_notify)
    (const char *errinfo, const void *private_info,
    size_t cb, void *user_data),
  void *user_data, cl_int *errcode_ret)

*properties:* CL_CONTEXT_PLATFORM, CL_GL_CONTEXT_KHR, CL_CGL_SHAREGROUP_KHR, CL_{EGL, GLX}_DISPLAY_KHR, CL_WGL_HDC_KHR

cl_context **clCreateContextFromType** (
  const cl_context_properties *properties,
  cl_device_type *device_type*, void (CL_CALLBACK *pfn_notify)
    (const char *errinfo, const void *private_info, size_t cb,
    void *user_data),
  void *user_data, cl_int *errcode_ret)

*properties:* See **clCreateContext**

cl_int **clRetainContext** (cl_context *context*)

cl_int **clReleaseContext** (cl_context *context*)

cl_int **clGetContextInfo** (cl_context *context*,
  cl_context_info *param_name*, size_t *param_value_size*,
  void *param_value, size_t *param_value_size_ret*)

*param_name:* CL_CONTEXT_REFERENCE_COUNT, CL_CONTEXT_{DEVICES, PROPERTIES}, CL_CONTEXT_NUM_DEVICES

### Querying Platform Info and Devices [4.1, 4.2]

cl_int **clGetPlatformIDs** (cl_uint *num_entries*,
  cl_platform_id *platforms, cl_uint *num_platforms*)

cl_int **clGetPlatformInfo** (cl_platform_id *platform*,
  cl_platform_info *param_name*, size_t *param_value_size*,
  void *param_value, size_t *param_value_size_ret*)

*param_name:* CL_PLATFORM_{PROFILE, VERSION}, CL_PLATFORM_{NAME, VENDOR, EXTENSIONS}

cl_int **clGetDeviceIDs** (cl_platform_id *platform*,
  cl_device_type *device_type*, cl_uint *num_entries*,
  cl_device_id *devices, cl_uint *num_devices*)

*device_type:* CL_DEVICE_TYPE_{CPU, GPU}, CL_DEVICE_TYPE_{ACCELERATOR, DEFAULT, ALL}

cl_int **clGetDeviceInfo** (cl_device_id *device*,
  cl_device_info *param_name*, size_t *param_value_size*,
  void *param_value, size_t *param_value_size_ret*)

*param_name:* CL_DEVICE_TYPE, CL_DEVICE_VENDOR_ID, CL_DEVICE_MAX_COMPUTE_UNITS, CL_DEVICE_MAX_WORK_ITEM_{DIMENSIONS, SIZES}, CL_DEVICE_MAX_WORK_GROUP_SIZE, CL_DEVICE_{NATIVE, PREFERRED}_VECTOR_WIDTH_{CHAR, SHORT, INT, LONG, FLOAT, DOUBLE, HALF}, CL_DEVICE_MAX_CLOCK_FREQUENCY, CL_DEVICE_ADDRESS_BITS, CL_DEVICE_MAX_MEM_ALLOC_SIZE, CL_DEVICE_IMAGE_SUPPORT, CL_DEVICE_MAX_{READ, WRITE}_IMAGE_ARGS, CL_DEVICE_IMAGE2D_MAX_{WIDTH, HEIGHT}, CL_DEVICE_IMAGE3D_MAX_{WIDTH, HEIGHT, DEPTH}, CL_DEVICE_MAX_SAMPLERS, CL_DEVICE_MAX_PARAMETER_SIZE, CL_DEVICE_MEM_BASE_ADDR_ALIGN, CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE, CL_DEVICE_SINGLE_FP_CONFIG, CL_DEVICE_GLOBAL_MEM_CACHE_{TYPE, SIZE}, CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE, CL_DEVICE_GLOBAL_MEM_SIZE, CL_DEVICE_MAX_CONSTANT_{BUFFER_SIZE, ARGS}, CL_DEVICE_LOCAL_MEM_{TYPE, SIZE}, CL_DEVICE_ERROR_CORRECTION_SUPPORT, CL_DEVICE_PROFILING_TIMER_RESOLUTION, CL_DEVICE_ENDIAN_LITTLE, CL_DEVICE_AVAILABLE, CL_DEVICE_COMPILER_AVAILABLE, CL_DEVICE_EXECUTION_CAPABILITIES, CL_DEVICE_QUEUE_PROPERTIES, CL_DEVICE_{NAME, VENDOR, PROFILE, EXTENSIONS}, CL_DEVICE_HOST_UNIFIED_MEMORY, CL_DEVICE_OPENCL_C_VERSION, CL_DEVICE_VERSION, CL_DRIVER_VERSION, CL_DEVICE_PLATFORM

## Buffer Objects

Elements of a buffer object can be a scalar or vector data type or a user-defined structure. Elements are stored sequentially and are accessed using a pointer by a kernel executing on a device. Data is stored in the same format as it is accessed by the kernel.

### Create Buffer Objects [5.2.1]

cl_mem **clCreateBuffer** (cl_context *context*,
  cl_mem_flags *flags*, size_t *size*, void *host_ptr*,
  cl_int *errcode_ret*)

cl_mem **clCreateSubBuffer** (cl_mem *buffer*,
  cl_mem_flags *flags*,
  cl_buffer_create_type *buffer_create_type*,
  const void *buffer_create_info, cl_int *errcode_ret*)

*flags* for **clCreateBuffer** and **clCreateSubBuffer**: CL_MEM_READ_WRITE, CL_MEM_{WRITE, READ}_ONLY, CL_MEM_{USE, ALLOC, COPY}_HOST_PTR

### Read, Write, Copy Buffer Objects [5.2.2]

cl_int **clEnqueueReadBuffer** (
  cl_command_queue *command_queue*, cl_mem *buffer*,
  cl_bool *blocking_read*, size_t *offset*, size_t *cb*,
  void *ptr, cl_uint *num_events_in_wait_list*,
  const cl_event *event_wait_list*, cl_event *event*)

cl_int **clEnqueueWriteBuffer** (
  cl_command_queue *command_queue*, cl_mem *buffer*,
  cl_bool *blocking_write*, size_t *offset*, size_t *cb*,
  const void *ptr, cl_uint *num_events_in_wait_list*,
  const cl_event *event_wait_list*, cl_event *event*)

cl_int **clEnqueueReadBufferRect** (
  cl_command_queue *command_queue*, cl_mem *buffer*,
  cl_bool *blocking_read*, const size_t *buffer_origin*[3],
  const size_t *host_origin*[3], const size_t *region*[3],
  size_t *buffer_row_pitch*, size_t *buffer_slice_pitch*,
  size_t *host_row_pitch*, size_t *host_slice_pitch*,
  void *ptr, cl_uint *num_events_in_wait_list*,
  const cl_event *event_wait_list*, cl_event *event*)

cl_int **clEnqueueWriteBufferRect** (
  cl_command_queue *command_queue*, cl_mem *buffer*,
  cl_bool *blocking_write*, const size_t *buffer_origin*[3],
  const size_t *host_origin*[3], const size_t *region*[3],
  size_t *buffer_row_pitch*, size_t *buffer_slice_pitch*,
  size_t *host_row_pitch*, size_t *host_slice_pitch*,
  void *ptr, cl_uint *num_events_in_wait_list*,
  const cl_event *event_wait_list*, cl_event *event*)

cl_int **clEnqueueCopyBuffer** (
  cl_command_queue *command_queue*,
  cl_mem *src_buffer*, cl_mem *dst_buffer*, size_t *src_offset*,
  size_t *dst_offset*, size_t *cb*,
  cl_uint *num_events_in_wait_list*,
  const cl_event *event_wait_list*, cl_event *event*)

cl_int **clEnqueueCopyBufferRect** (
  cl_command_queue *command_queue*,
  cl_mem *src_buffer*, cl_mem *dst_buffer*,
  const size_t *src_origin*[3], const size_t *dst_origin*[3],
  const size_t *region*[3], size_t *src_row_pitch*,
  size_t *src_slice_pitch*, size_t *dst_row_pitch*,
  size_t *dst_slice_pitch*, cl_uint *num_events_in_wait_list*,
  const cl_event *event_wait_list*, cl_event *event*)

### Map Buffer Objects [5.2.3]

void * **clEnqueueMapBuffer** (
  cl_command_queue *command_queue*, cl_mem *buffer*,
  cl_bool *blocking_map*, cl_map_flags *map_flags*,
  size_t *offset*, size_t *cb*, cl_uint *num_events_in_wait_list*,
  const cl_event *event_wait_list*, cl_event *event*,
  cl_int *errcode_ret*)

### Map Buffer Objects [5.4.1-2]

cl_int **clRetainMemObject** (cl_mem *memobj*)

cl_int **clReleaseMemObject** (cl_mem *memobj*)

cl_int **clSetMemObjectDestructorCallback** (
  cl_mem *memobj*, void (CL_CALLBACK *pfn_notify)
    (cl_mem *memobj*, void *user_data),
  void *user_data)

cl_int **clEnqueueUnmapMemObject** (
  cl_command_queue *command_queue*, cl_mem *memobj*,
  void *mapped_ptr*, cl_uint *num_events_in_wait_list*,
  const cl_event *event_wait_list*, cl_event *event*)

### Query Buffer Object [5.4.3]

cl_int **clGetMemObjectInfo** (cl_mem *memobj*,
  cl_mem_info *param_name*, size_t *param_value_size*,
  void *param_value, size_t *param_value_size_ret*)

*param_name:* CL_MEM_{TYPE, FLAGS, SIZE, HOST_PTR}, CL_MEM_{MAP_REFERENCE}_COUNT, CL_MEM_{OFFSET, CONTEXT, ASSOCIATED_MEMOBJECT}

## Program Objects

### Create Program Objects [5.6.1]

cl_program **clCreateProgramWithSource** (
  cl_context *context*, cl_uint *count*, const char **strings*,
  const size_t *lengths, cl_int *errcode_ret*)

cl_program **clCreateProgramWithBinary** (
  cl_context *context*, cl_uint *num_devices*,
  const cl_device_id *device_list, const size_t *lengths*,
  const unsigned char **binaries, cl_int *binary_status*,
  cl_int *errcode_ret*)

cl_int **clRetainProgram** (cl_program *program*)

cl_int **clReleaseProgram** (cl_program *program*)

### Build Program Executable [5.6.2]

cl_int **clBuildProgram** (cl_program *program*,
  cl_uint *num_devices*, const cl_device_id *device_list*,
  const char *options, void (CL_CALLBACK*pfn_notify)
    (cl_program *program*, void *user_data),
  void *user_data)

### Build Options [5.6.3]

**Preprocessor:** *(-D processed in order listed in clBuildProgram)*
-D name -D name=definition -I dir

**Optimization options:**
-cl-opt-disable, -cl-strict-aliasing, -cl-mad-enable, -cl-no-signed-zeros, -cl-finite-math-only, -cl-fast-relaxed-math, -cl-unsafe-math-optimizations

**Math intrinsics:**
-cl-single-precision-constant -cl-denorms-are-zero

**Warning request/suppress:**
-w -Werror

**Control OpenCL C language version:**
-cl-std=CL1.1 // OpenCL 1.1 specification.

### Query Program Objects [5.6.5]

cl_int **clGetProgramInfo** (cl_program *program*,
  cl_program_info *param_name*, size_t *param_value_size*,
  void *param_value, size_t *param_value_size_ret*)

*param_name:* CL_PROGRAM_{REFERENCE_COUNT}, CL_PROGRAM_{CONTEXT, NUM_DEVICES, DEVICES}, CL_PROGRAM_{SOURCE, BINARY_SIZES, BINARIES}

[Program Objects Continue >]

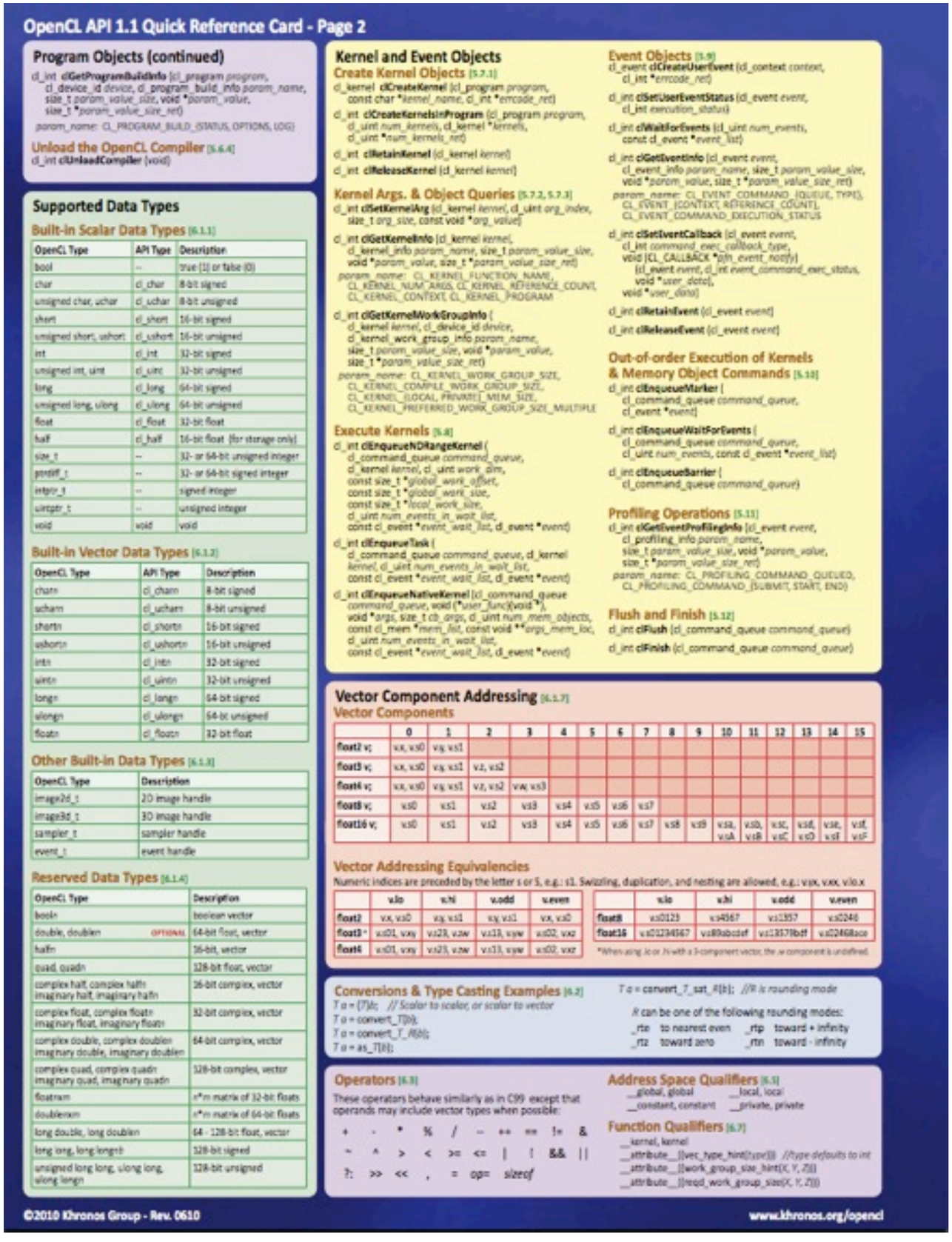


www.khronos.org/opencl

CITA|ICAT

# Quick Reference Card

- Page 2



OpenCL API 1.1 Quick Reference Card - Page 2

**Program Objects (continued)**

cl_int **clGetProgramBuildInfo** (cl_program *program*, cl_device_id *device*, cl_program_build_info *param_name*, size_t *param_value_size*, void **param_value*, size_t **param_value_size_ret*)

*param_name:* CL_PROGRAM_BUILD_(STATUS, OPTIONS, LOG)

**Unload the OpenCL Compiler** [5.6.4]

cl_int **clUnloadCompiler** (void)

**Supported Data Types**

**Built-in Scalar Data Types** [6.1.1]

| OpenCL Type | API Type | Description |
|---|---|---|
| bool | -- | true (1) or false (0) |
| char | cl_char | 8-bit signed |
| unsigned char, uchar | cl_uchar | 8-bit unsigned |
| short | cl_short | 16-bit signed |
| unsigned short, ushort | cl_ushort | 16-bit unsigned |
| int | cl_int | 32-bit signed |
| unsigned int, uint | cl_uint | 32-bit unsigned |
| long | cl_long | 64-bit signed |
| unsigned long, ulong | cl_ulong | 64-bit unsigned |
| float | cl_float | 32-bit float |
| half | cl_half | 16-bit float (for storage only) |
| size_t | -- | 32- or 64-bit unsigned integer |
| ptrdiff_t | -- | 32- or 64-bit signed integer |
| intptr_t | -- | signed integer |
| uintptr_t | -- | unsigned integer |
| void | void | void |

**Built-in Vector Data Types** [6.1.2]

| OpenCL Type | API Type | Description |
|---|---|---|
| charn | cl_charn | 8-bit signed |
| ucharn | cl_ucharn | 8-bit unsigned |
| shortn | cl_shortn | 16-bit signed |
| ushortn | cl_ushortn | 16-bit unsigned |
| intn | cl_intn | 32-bit signed |
| uintn | cl_uintn | 32-bit unsigned |
| longn | cl_longn | 64-bit signed |
| ulongn | cl_ulongn | 64-bit unsigned |
| floatn | cl_floatn | 32-bit float |

**Other Built-in Data Types** [6.1.3]

| OpenCL Type | Description |
|---|---|
| image2d_t | 2D image handle |
| image3d_t | 3D image handle |
| sampler_t | sampler handle |
| event_t | event handle |

**Reserved Data Types** [6.1.4]

| OpenCL Type | Description |
|---|---|
| booln | boolean vector |
| double, doublen OPTIONAL | 64-bit float, vector |
| halfn | 16-bit, vector |
| quad, quadn | 128-bit float, vector |
| complex half, complex halfn imaginary half, imaginary halfn | 16-bit complex, vector |
| complex float, complex floatn imaginary float, imaginary floatn | 32-bit complex, vector |
| complex double, complex doublen imaginary double, imaginary doublen | 64-bit complex, vector |
| complex quad, complex quadn imaginary quad, imaginary quadn | 128-bit complex, vector |
| floatnxm | n*m matrix of 32-bit floats |
| doublenxm | n*m matrix of 64-bit floats |
| long double, long doublen | 64 - 128-bit float, vector |
| long long, long longn | 128-bit signed |
| unsigned long long, ulong long, ulong longn | 128-bit unsigned |

**Kernel and Event Objects**

**Create Kernel Objects** [5.7.1]

cl_kernel **clCreateKernel** (cl_program *program*, const char **kernel_name*, cl_int **errcode_ret*)

cl_int **clCreateKernelsInProgram** (cl_program *program*, cl_uint *num_kernels*, cl_kernel **kernels*, cl_uint **num_kernels_ret*)

cl_int **clRetainKernel** (cl_kernel *kernel*)

cl_int **clReleaseKernel** (cl_kernel *kernel*)

**Kernel Args. & Object Queries** [5.7.2, 5.7.3]

cl_int **clSetKernelArg** (cl_kernel *kernel*, cl_uint *arg_index*, size_t *arg_size*, const void **arg_value*)

cl_int **clGetKernelInfo** (cl_kernel *kernel*, cl_kernel_info *param_name*, size_t *param_value_size*, void **param_value*, size_t **param_value_size_ret*)

*param_name:* CL_KERNEL_FUNCTION_NAME, CL_KERNEL_NUM_ARGS, CL_KERNEL_REFERENCE_COUNT, CL_KERNEL_CONTEXT, CL_KERNEL_PROGRAM

cl_int **clGetKernelWorkGroupInfo** (cl_kernel *kernel*, cl_device_id *device*, cl_kernel_work_group_info *param_name*, size_t *param_value_size*, void **param_value*, size_t **param_value_size_ret*)

*param_name:* CL_KERNEL_WORK_GROUP_SIZE, CL_KERNEL_COMPILE_WORK_GROUP_SIZE, CL_KERNEL_(LOCAL, PRIVATE)_MEM_SIZE, CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE

**Execute Kernels** [5.8]

cl_int **clEnqueueNDRangeKernel** (cl_command_queue *command_queue*, cl_kernel *kernel*, cl_uint *work_dim*, const size_t **global_work_offset*, const size_t **global_work_size*, const size_t **local_work_size*, cl_uint *num_events_in_wait_list*, const cl_event **event_wait_list*, cl_event **event*)

cl_int **clEnqueueTask** (cl_command_queue *command_queue*, cl_kernel *kernel*, cl_uint *num_events_in_wait_list*, const cl_event **event_wait_list*, cl_event **event*)

cl_int **clEnqueueNativeKernel** (cl_command_queue *command_queue*, void (**user_func*)(void **), void **args*, size_t *cb_args*, cl_uint *num_mem_objects*, const cl_mem **mem_list*, const void ***args_mem_loc*, cl_uint *num_events_in_wait_list*, const cl_event **event_wait_list*, cl_event **event*)

**Event Objects** [5.9]

cl_event **clCreateUserEvent** (cl_context *context*, cl_int **errcode_ret*)

cl_int **clSetUserEventStatus** (cl_event *event*, cl_int *execution_status*)

cl_int **clWaitForEvents** (cl_uint *num_events*, const cl_event **event_list*)

cl_int **clGetEventInfo** (cl_event *event*, cl_event_info *param_name*, size_t *param_value_size*, void **param_value*, size_t **param_value_size_ret*)

*param_name:* CL_EVENT_COMMAND_(QUEUE, TYPE), CL_EVENT_(CONTEXT, REFERENCE_COUNT), CL_EVENT_COMMAND_EXECUTION_STATUS

cl_int **clSetEventCallback** (cl_event *event*, cl_int *command_exec_callback_type*, void (CL_CALLBACK **pfn_event_notify*)(cl_event *event*, cl_int *event_command_exec_status*, void **user_data*), void **user_data*)

cl_int **clRetainEvent** (cl_event *event*)

cl_int **clReleaseEvent** (cl_event *event*)

**Out-of-order Execution of Kernels & Memory Object Commands** [5.10]

cl_int **clEnqueueMarker** (cl_command_queue *command_queue*, cl_event **event*)

cl_int **clEnqueueWaitForEvents** (cl_command_queue *command_queue*, cl_uint *num_events*, const cl_event **event_list*)

cl_int **clEnqueueBarrier** (cl_command_queue *command_queue*)

**Profiling Operations** [5.11]

cl_int **clGetEventProfilingInfo** (cl_event *event*, cl_profiling_info *param_name*, size_t *param_value_size*, void **param_value*, size_t **param_value_size_ret*)

*param_name:* CL_PROFILING_COMMAND_QUEUED, CL_PROFILING_COMMAND_(SUBMIT, START, END)

**Flush and Finish** [5.12]

cl_int **clFlush** (cl_command_queue *command_queue*)

cl_int **clFinish** (cl_command_queue *command_queue*)

**Vector Component Addressing** [6.1.7]

**Vector Components**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **float2 v;** | v.x, v.s0 | v.y, v.s1 | | | | | | | | | | | | | | |
| **float3 v;** | v.x, v.s0 | v.y, v.s1 | v.z, v.s2 | | | | | | | | | | | | | |
| **float4 v;** | v.x, v.s0 | v.y, v.s1 | v.z, v.s2 | v.w, v.s3 | | | | | | | | | | | | |
| **float8 v;** | v.s0 | v.s1 | v.s2 | v.s3 | v.s4 | v.s5 | v.s6 | v.s7 | | | | | | | | |
| **float16 v;** | v.s0 | v.s1 | v.s2 | v.s3 | v.s4 | v.s5 | v.s6 | v.s7 | v.s8 | v.s9 | v.sA, v.sA | v.sB, v.sB | v.sC, v.sC | v.sD, v.sD | v.sE, v.sE | v.sF, v.sF |

**Vector Addressing Equivalencies**

Numeric indices are preceded by the letter s or S, e.g.: s1. Swizzling, duplication, and nesting are allowed, e.g.: v.yx, v.xx, v.lo.x

| | v.lo | v.hi | v.odd | v.even |
|---|---|---|---|---|
| **float2** | v.x, v.s0 | v.y, v.s1 | v.y, v.s1 | v.x, v.s0 |
| **float3** * | v.s01, v.xy | v.s23, v.zw | v.s13, v.yw | v.s02, v.xz |
| **float4** | v.s01, v.xy | v.s23, v.zw | v.s13, v.yw | v.s02, v.xz |

| | v.lo | v.hi | v.odd | v.even |
|---|---|---|---|---|
| **float8** | v.s0123 | v.s4567 | v.s1357 | v.s0246 |
| **float16** | v.s01234567 | v.s89abcdef | v.s13579bdf | v.s02468ace |

*When using .lo or .hi with a 3-component vector, the .w component is undefined.

**Conversions & Type Casting Examples** [6.2]

*T a = (T)b; // Scalar to scalar, or scalar to vector*
*T a = convert_T(b);*
*T a = convert_T_R(b);*
*T a = as_T(b);*

*T a = convert_T_sat_R(b); //R is rounding mode*

*R* can be one of the following rounding modes:

| | | | |
|---|---|---|---|
| _rte | to nearest even | _rtp | toward + infinity |
| _rtz | toward zero | _rtn | toward - infinity |

**Operators** [6.3]

These operators behave similarly as in C99 except that operands may include vector types when possible:

\+ - * % / -- ++ == != & ~ ^ > < >= <= | ! && || ?: >> << , = op= sizeof

**Address Space Qualifiers** [6.5]

__global, global __local, local
__constant, constant __private, private

**Function Qualifiers** [6.7]

__kernel, kernel
__attribute__((vec_type_hint(*type*))) *//type defaults to int*
__attribute__((work_group_size_hint(X, Y, Z)))
__attribute__((reqd_work_group_size(X, Y, Z)))

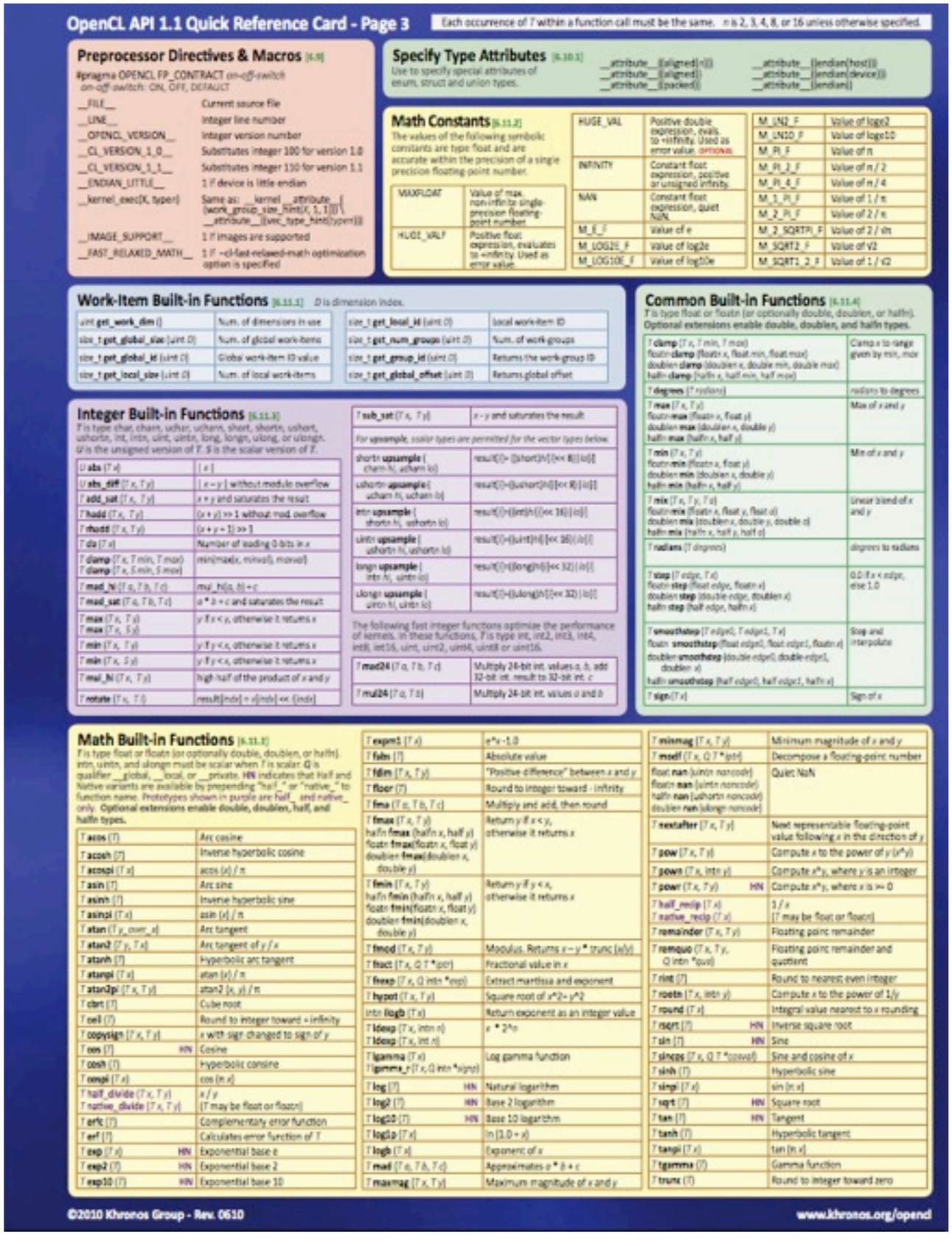


www.khronos.org/opencl

CITA|ICAT

# Quick Reference Card

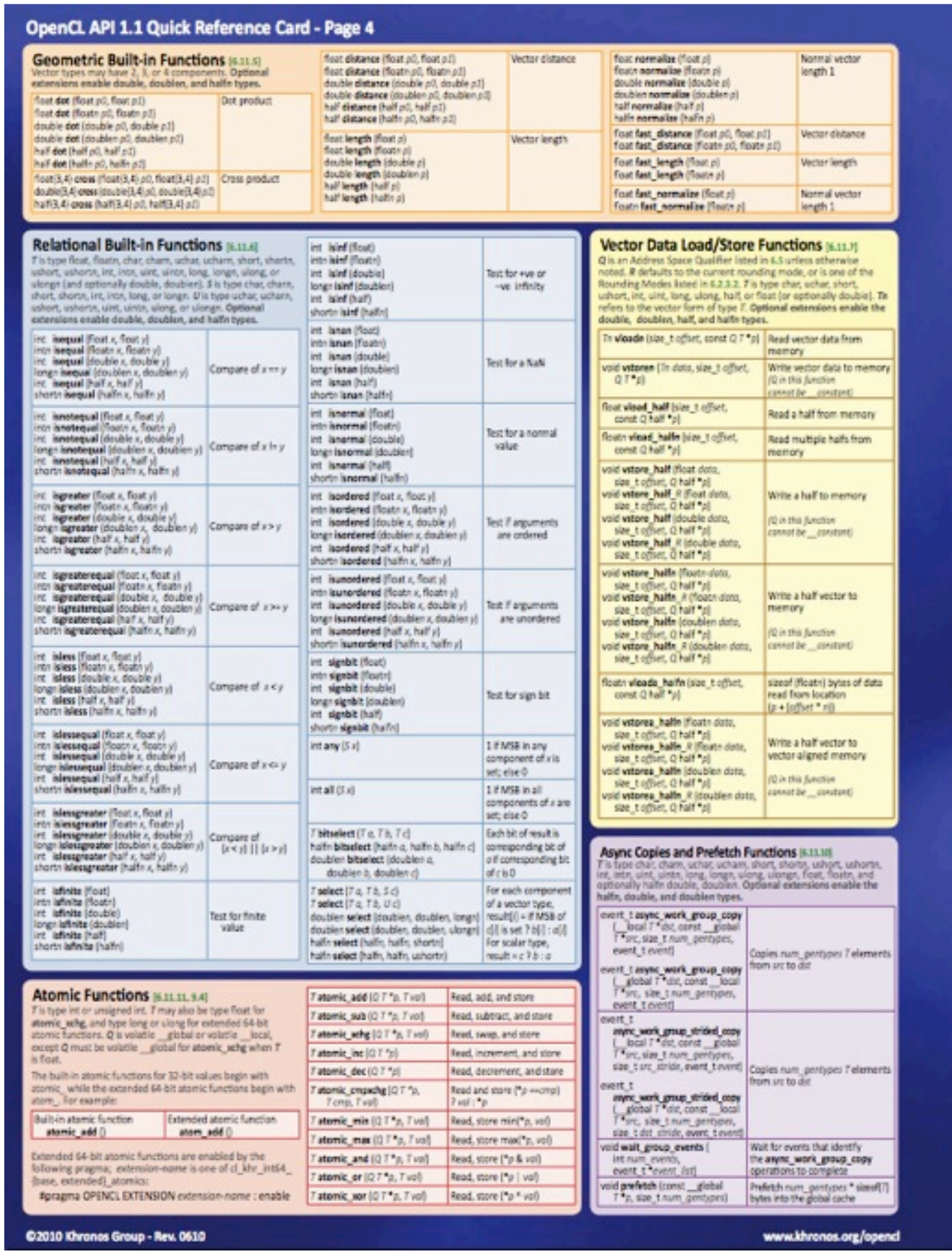
- Page 3




CITA|ICAT

# Quick Reference Card

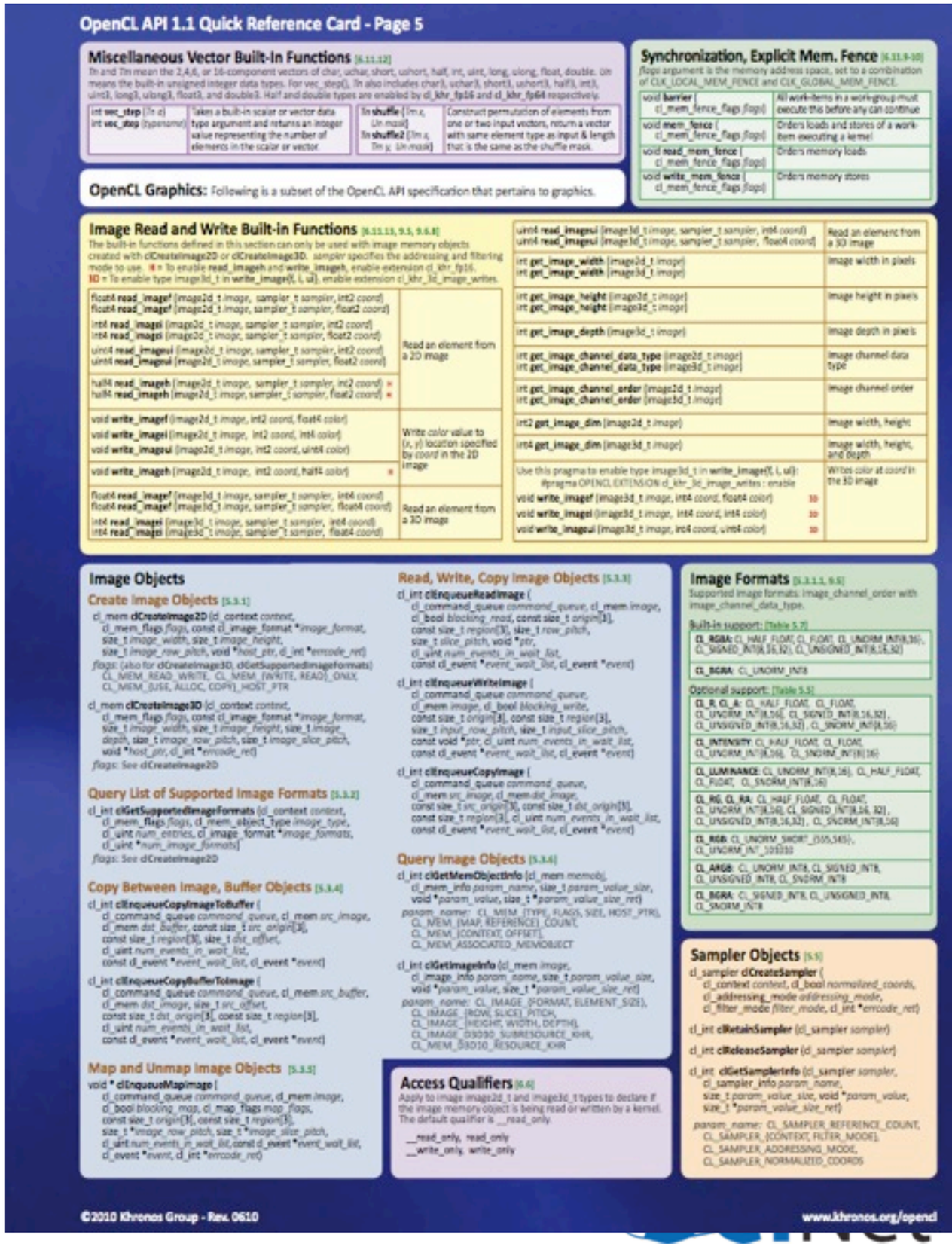
- Page 4




CITA|ICAT

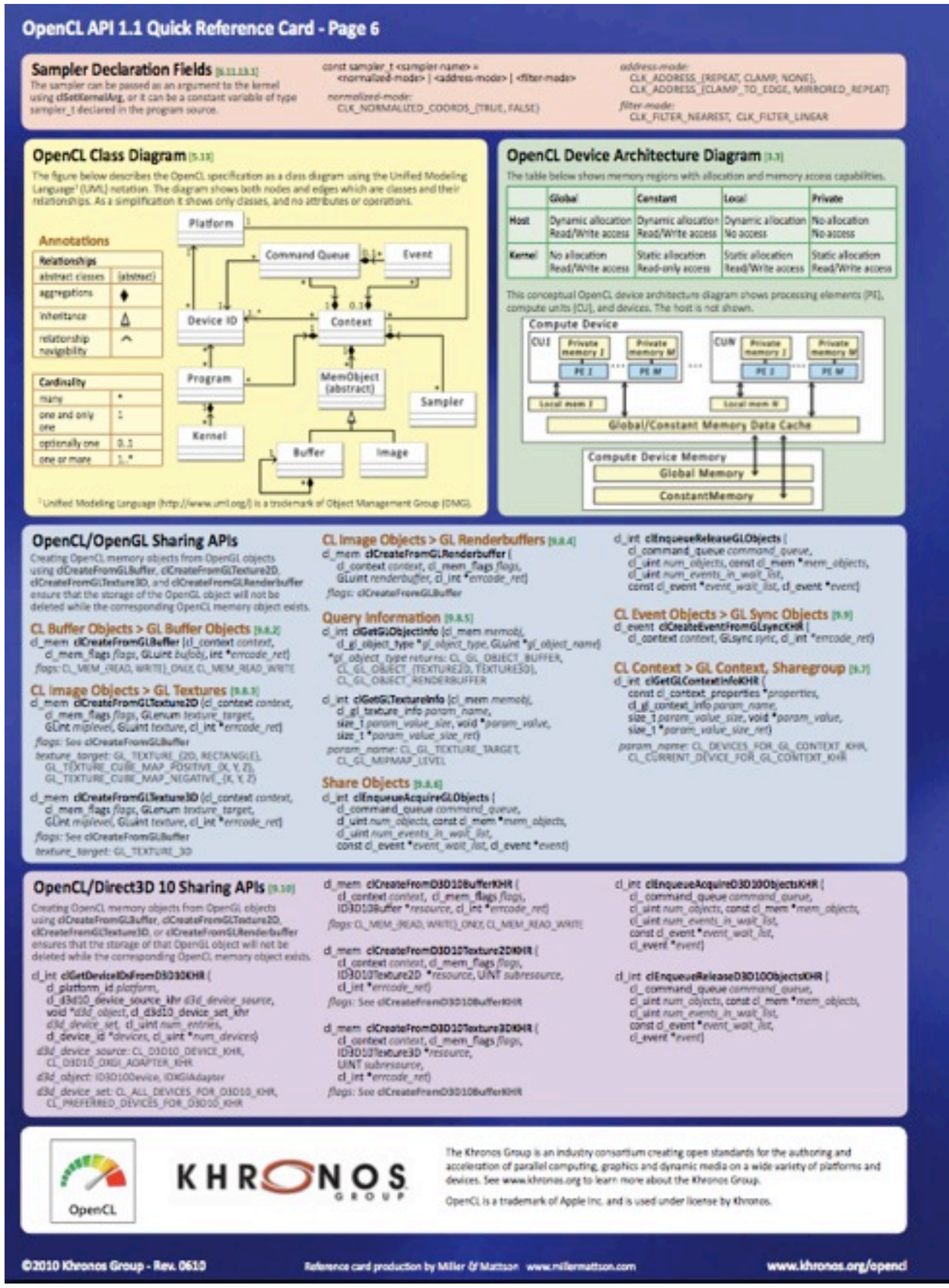
# Quick Reference Card

- Page 5



# Quick Reference Card

- Page 6
- Only touches on C++ bindings, etc.
- Necessary and inevitable consequence of generality.
- But works, and works a lot of places

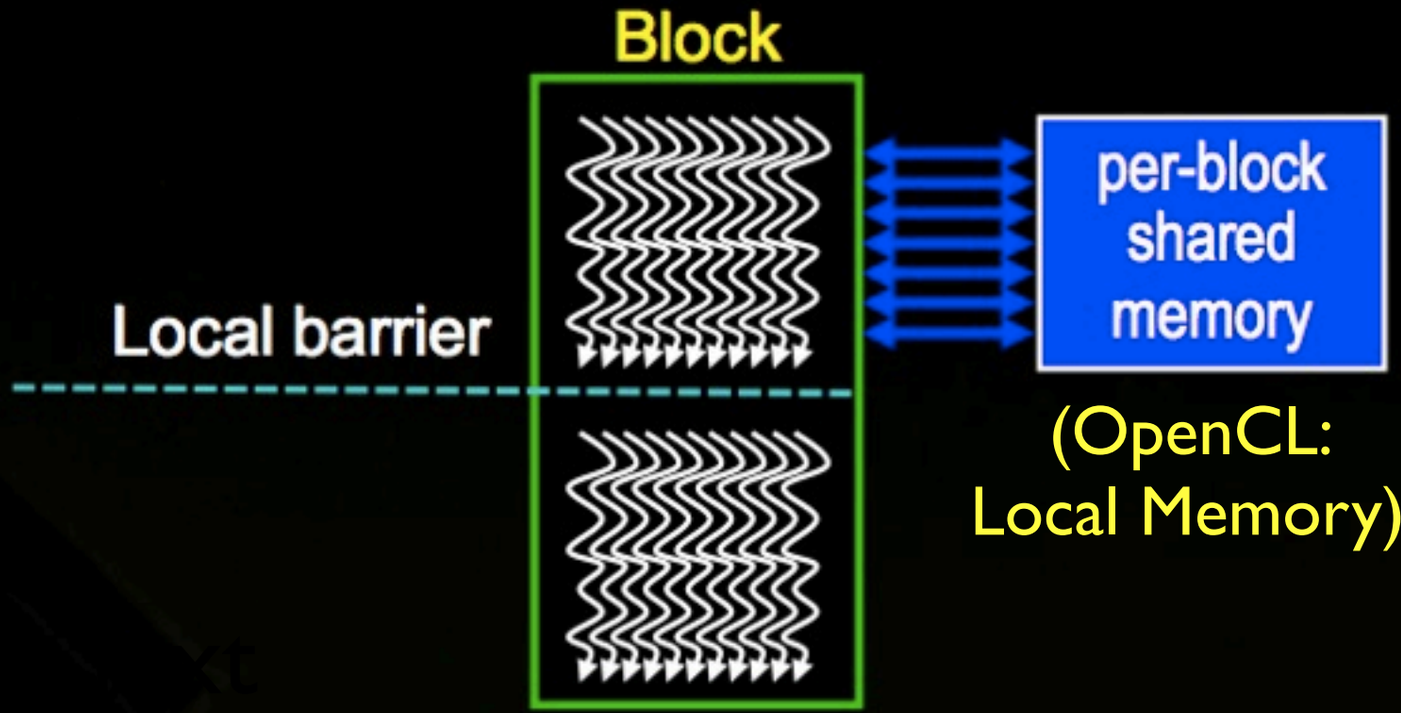


# CUDA In One Slide (And OpenCL)

Thread (OpenCL: Work Item)


(OpenCL:
Private Memory)

(OpenCL: Work Group)




(OpenCL:
Local Memory)

# CUDA vs OpenCL terminology

- Since maps to similar hardware, basic concepts the same
- Some terminology changes; some better, some worse.

| CUDA | OpenCL |
|---|---|
| Thread | Work Item |
| Block | Work Group |
| Device Mem | Global Mem |
| Constant Mem | Constant Mem |
| Shared Mem | Local Mem |
| Local Mem | Private Mem |

# CUDA vs OpenCL kernel code

- Since maps to similar hardware, basic concepts the same
- Some terminology changes; some better, some worse.

| CUDA | OpenCL |
|---|---|
| __global__ | __kernel |
| __device__ (function) | |
| __constant__ | __constant |
| __device__ (mem) | __global |
| __shared__ | __local |
| Local Mem | Private Mem |
| __syncthreads() | barrier() |

# CUDA vs OpenCL kernel code

| CUDA | OpenCL |
|---|---|
| gridDim | get_num_groups() |
| blockDim | get_local_size() |
| blockIdx | get_group_id() |
| threadIdx | get_local_id() |
| - | get_global_id() |
| - | get_global_size() |

- Functions rather than variables for place in grid
- “global work size” and “local work size” vs blockDim*gridDim and BlockDim
- get_global_id() calculates global position for you

```cuda
__global__ void cuda_saxpb(const float *xd,
                           const float a,
                           const float b,
                           float *yd, const int n) {

    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i<n) {
        yd[i] = a*xd[i]+b;
    }
    return;
}
```

CUDA

---

```opencl
__kernel void opencl_saxpb(__global const float *x,
                           const float a, const float b,
                    __global       float *y)
{
    int i = get_global_id(0);
    if (i < get_global_size(0) )
        y[i] = a*x[i] + b;
}
```

OpenCL

CITA|ICAT

SciNet

# CUDA

```
/* run GPU code */
CHK_CUDA( cudaMalloc(&xd, n*sizeof(float)) );
CHK_CUDA( cudaMalloc(&yd, n*sizeof(float)) );

tick(&gputimer);
CHK_CUDA( cudaMemcpy(xd, x, n*sizeof(float), cudaMemcpyHostToDevice) );

blocksize = (n+nblocks-1)/nblocks;
for (i=0; i<niters; i++) {
    cuda_saxpb<<<nblocks, blocksize>>>(xd, a, b, yd, n);
}
CHK_CUDA( cudaMemcpy(ycuda, yd, n*sizeof(float), cudaMemcpyDeviceToHost) );
gputime = tock(&gputimer);

CHK_CUDA( cudaFree(xd) );
CHK_CUDA( cudaFree(yd) );
```

```c
/* create OpenCL device & context */
cl_platform_id clPlatform;
err = clGetPlatformIDs(1, &clPlatform, NULL);
chk(err, "Get Platform`");

/* query all devices available to the context */

cl_device_id device;
err = clGetDeviceIDs(clPlatform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
chk(err, "Get Device IDs");

cl_context hContext;
hContext = clCreateContext(0, 1, &device, NULL, NULL, &err);
chk(err, "Get Context");

/* create a command queue for first device the context reported */
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, device, 0, &err);
chk(err, "Create Queue");
```

```c
/* create & compile program */
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, nlines, kernelsrc, 0, &err);
chk(err, "Create Program");

err = clBuildProgram(hProgram, 1, &device, NULL, NULL, NULL);
buildchk(err, "Build Program");

/* create kernel */
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "opencl_saxpb", &err);
chk(err, "Create Kernel");

/* allocate device memory */
cl_mem yd;
cl_mem xd;
tick(&gputimer);
xd = clCreateBuffer(hContext,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        n * sizeof(cl_float),
        x,
        &err);
chk(err, "Create xd");
```

```c
yd = clCreateBuffer(hContext,
        CL_MEM_COPY_HOST_PTR,
        n * sizeof(cl_float),
        yopencl,
        &err);
chk(err, "Create yd");

/* setup parameter values */
err  = clSetKernelArg(hKernel, 0, sizeof(cl_mem),   (void *)&xd);
err |= clSetKernelArg(hKernel, 1, sizeof(cl_float), (void *)&a);
err |= clSetKernelArg(hKernel, 2, sizeof(cl_float), (void *)&b);
err |= clSetKernelArg(hKernel, 3, sizeof(cl_mem),   (void *)&yd);
chk(err, "Set args");

/* execute kernel */
const size_t knsize=n;
const size_t kblocksize=blocksize;
err = clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                                &knsize, &kblocksize, 0, 0, 0);
chk(err, "Enqueue Kernel");
```

```c
// copy results from device back to host
err = clEnqueueReadBuffer(hCmdQueue, yd, CL_TRUE, 0,
            n * sizeof(cl_float),
            yopencl, 0, 0, 0);
chk(err, "Enqueue Read");

clReleaseMemObject(xd);
clReleaseMemObject(yd);
clReleaseProgram(hProgram);
clReleaseKernel(hKernel);
clReleaseCommandQueue(hCmdQueue);
clReleaseContext(hContext);
```

# OpenCL

CITA|ICAT

SciNet

# More general - more verbose

- CUDA makes simple things simple - easy to do with fixed view of hardware
- OpenCL doesn't.  But more complicated situations are exactly the same; learning curve steep, but short.
- Let's step through this (new OpenCL 1.1 C++ bindings a little nicer)

```
/* create OpenCL device & context */
cl_platform_id clPlatform;
err = clGetPlatformIDs(1, &clPlatform, NULL);
chk(err, "Get Platform`");

/* query all devices available to the context */

cl_device_id device;
err = clGetDeviceIDs(clPlatform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
chk(err, "Get Device IDs");

cl_context hContext;
hContext = clCreateContext(0, 1, &device, NULL, NULL, &err);
chk(err, "Get Context");

/* create a command queue for first device the context reported */
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, device, 0, &err);
chk(err, "Create Queue");

/* create & compile program */
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, nlines, kernelsrc, 0, &err);
chk(err, "Create Program");

err = clBuildProgram(hProgram, 1, &device, NULL, NULL, NULL);
buildchk(err, "Build Program");

/* create kernel */
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "opencl_saxpb", &err);
chk(err, "Create Kernel");

/* allocate device memory */
cl_mem yd;
cl_mem xd;
tick(&gputimer);
xd = clCreateBuffer(hContext,
       CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
       n * sizeof(cl_float),
       x,
       &err);
chk(err, "Create xd");
```

```
yd = clCreateBuffer(hContext,
       CL_MEM_COPY_HOST_PTR,
       n * sizeof(cl_float),
       yopencl,
       &err);
chk(err, "Create yd");

/* setup parameter values */
err  = clSetKernelArg(hKernel, 0, sizeof(cl_mem),   (void *
err |= clSetKernelArg(hKernel, 1, sizeof(cl_float), (void *
err |= clSetKernelArg(hKernel, 2, sizeof(cl_float), (void *
err |= clSetKernelArg(hKernel, 3, sizeof(cl_mem),   (void *
chk(err, "Set args");

/* execute kernel */
const size_t knsize=n;
const size_t kblocksize=blocksize;
err = clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                             &knsize, &kblocksize, 0, 0, 0)
chk(err, "Enqueue Kernel");

// copy results from device back to host
err = clEnqueueReadBuffer(hCmdQueue, yd, CL_TRUE, 0,
          n * sizeof(cl_float),
          yopencl, 0, 0, 0);
chk(err, "Enqueue Read");

clReleaseMemObject(xd);
clReleaseMemObject(yd);
clReleaseProgram(hProgram);
clReleaseKernel(hKernel);
clReleaseCommandQueue(hCmdQueue);
clReleaseContext(hContext);
```

CITA|ICAT

SciNet

```c
/* create OpenCL device & context */
cl_platform_id clPlatform;
err = clGetPlatformIDs(1, &clPlatform, NULL);
chk(err, "Get Platform`");

/* query all devices available to the context */
cl_device_id device;
err = clGetDeviceIDs(clPlatform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
chk(err, "Get Device IDs");

cl_context hContext;
hContext = clCreateContext(0, 1, &device, NULL, NULL, &err);
chk(err, "Get Context");

/* create a command queue for first device the context reported */
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, device, 0, &err);
chk(err, "Create Queue");
```

Have to do something like this in CUDA for multiple GPUs

```c
/* create OpenCL device & context */
cl_platform_id clPlatform;
err = clGetPlatformIDs(1, &clPlatform, NULL);
chk(err, "Get Platform`");

/* query all devices available to the context */
cl_device_id device;
err = clGetDeviceIDs(clPlatform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
chk(err, "Get Device IDs");

cl_context hContext;
hContext = clCreateContext(0, 1, &device, NULL, NULL, &err);
chk(err, "Get Context");

/* create a command queue for first device the context reported */
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, device, 0, &err);
chk(err, "Create Queue");
```

Doesn't have to be a GPU! "Host" could be a "device"

```
const char *kernelsrc[] = {
"__kernel void opencl_saxpb(__global const float *x,",
"      const float a, const float b,",
"      __global float *y)",
" { ",
"      int i = get_global_id(0);",
"      if (i < get_global_size(0) )",
"          y[i] = a*x[i] + b;",
" }"
};
const int nlines=8;
```

```
/* create & compile program */
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, nlines, kernelsrc, 0, &err);
chk(err, "Create Program");

err = clBuildProgram(hProgram, 1, &device, NULL, NULL, NULL);
buildchk(err, "Build Program");

/* create kernel */
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "opencl_saxpb", &err);
chk(err, "Create Kernel");
```

Have to compile, build program and identify kernels within the program! Allows real heterogenaity; also, run-time rebuilding of program (CUDA can now do this too)

As a result of run-time "device" compilation, build the main program just with normal compiler, link appropriate OpenCL libraries.

```
[ljdursi@tpb1 class5]$ make
gcc -Wall -g -O3 -o block-saxpb block-saxpb.c -lOpenCL -lm
[ljdursi@tpb1 class5]$
```

```
/* allocate device memory */
cl_mem yd;
cl_mem xd;
tick(&gputimer);
xd = clCreateBuffer(hContext,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        n * sizeof(cl_float),
        x,
        &err);
chk(err, "Create xd");

yd = clCreateBuffer(hContext,
        CL_MEM_COPY_HOST_PTR,
        n * sizeof(cl_float),
        yopencl,
        &err);
chk(err, "Create yd");
```

Malloc, Memcpy can be done in one step.

Can give many options (eg, READ_ONLY for input values).

```
/* setup parameter values */
err  = clSetKernelArg(hKernel, 0, sizeof(cl_mem),   (void *)&xd);
err |= clSetKernelArg(hKernel, 1, sizeof(cl_float), (void *)&a);
err |= clSetKernelArg(hKernel, 2, sizeof(cl_float), (void *)&b);
err |= clSetKernelArg(hKernel, 3, sizeof(cl_mem),   (void *)&yd);
chk(err, "Set args");
```

Have to manually marshal arguments to kernel launch.

```
err = clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                             &knsize, &kblocksize, 0, 0, 0);
chk(err, "Enqueue Kernel");
```

```
/* setup parameter values */
err  = clSetKernelArg(hKernel, 0, sizeof(cl_mem),   (void *)&xd);
err |= clSetKernelArg(hKernel, 1, sizeof(cl_float), (void *)&a);
err |= clSetKernelArg(hKernel, 2, sizeof(cl_float), (void *)&b);
err |= clSetKernelArg(hKernel, 3, sizeof(cl_mem),   (void *)&yd);
chk(err, "Set args");
```

Have to manually marshal arguments to kernel launch.

```
err = clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                             &knsize, &kblocksize, 0, 0, 0);
chk(err, "Enqueue Kernel");
```

```
// copy results from device back to host
err = clEnqueueReadBuffer(hCmdQueue, yd, CL_TRUE, 0,
                n * sizeof(cl_float),
                yopencl, 0, 0, 0);
chk(err, "Enqueue Read");
```

```
clReleaseMemObject(xd);
clReleaseMemObject(yd);
clReleaseProgram(hProgram);
clReleaseKernel(hKernel);
clReleaseCommandQueue(hCmdQueue);
clReleaseContext(hContext);
```

Copy-back, cleanup not that different

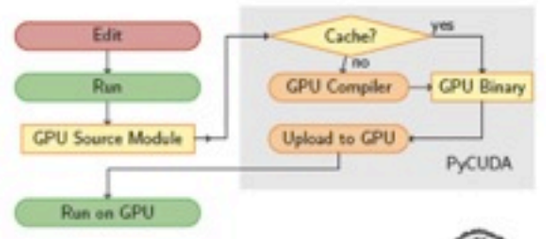
# Resources

- PyCUDA/PyOpenCL
- **http://mathema.tician.de/software/pycuda**
- Does a lot of the tedious initialization stuff for you
- Can focus on getting efficient kernels
- Can move to C/C++ later

## Python + CUDA = PyCUDA


- All of CUDA in a modern scripting language
- Full Documentation
- Free, open source (MIT)
- Also: PyOpenCL

- CUDA C Code = Strings
- Generate Code Easily
  - Automated Tuning
- Batteries included: GPU Arrays, RNG, ...
- Integration: `numpy` arrays, Plotting, Optimization, ...




BROWN

CITA|ICAT

SciNet

# CUDA

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

saxpb = SourceModule("""
  __global__ void cuda_saxpb(const float *x, const float a,
                      const float b, float *y, const int n)
  {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < n)
        y[i] = a*x[i] + b;
  }
  """)

npts = 1600
nblocks = 10
blocksize = (npts + nblocks-1)/nblocks
a = 5.
b = -1.
x = (numpy.random.randn(npts)).astype(numpy.float32)
y = numpy.empty_like(x)

# do GPU calculation

x_gpu = gpuarray.to_gpu(x)
y_gpu = gpuarray.to_gpu(y)

cuda_saxpb = saxpb.get_function(cuda_saxpb)
cuda_saxpb(x_gpu, a, b, y_gpu, npts, block=(blocksize,1,1))

# do CPU calculation
y = a * x - b

print 'GPU, CPU differ by ', numpy.sum(y_gpu - y)
```

# OpenCL

```
import pyopencl as cl
import numpy

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

prg = cl.Program(ctx,"""
  __kernel void cl_saxpb(__global const float *x,
                         const float a, const float b,
                         __global float *y)
  {
    int i = get_global_id(0)
    if (i < get_global_size(0) )
        y[i] = a*x[i] + b;
  }
  """)

npts = 1600
nblocks = 10
blocksize = (npts + nblocks-1)/nblocks
a = 5.
b = -1.

x = (numpy.random.randn(npts)).astype(numpy.float32)
y = numpy.empty_like(x)

mf = cl.mem_flags
x_cl = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=x)
y_cl = cl.Buffer(ctx, mf.WRITE_ONLY, y.nbytes)

# do GPU calculation

prg.cl_saxpb(queue, x.shape, None, x_cl, a, b, y_cl)

y_gpu_res = numpy.empty_like(y)
cl.enqueue_read_buffer(queue, y_buf, y_gpu_res).wait()

# do CPU calculation
y = a * x - b

print 'GPU, CPU differ by ', numpy.sum(y_gpu_res - y)
```

CITA|ICAT

SciNet

# Resources

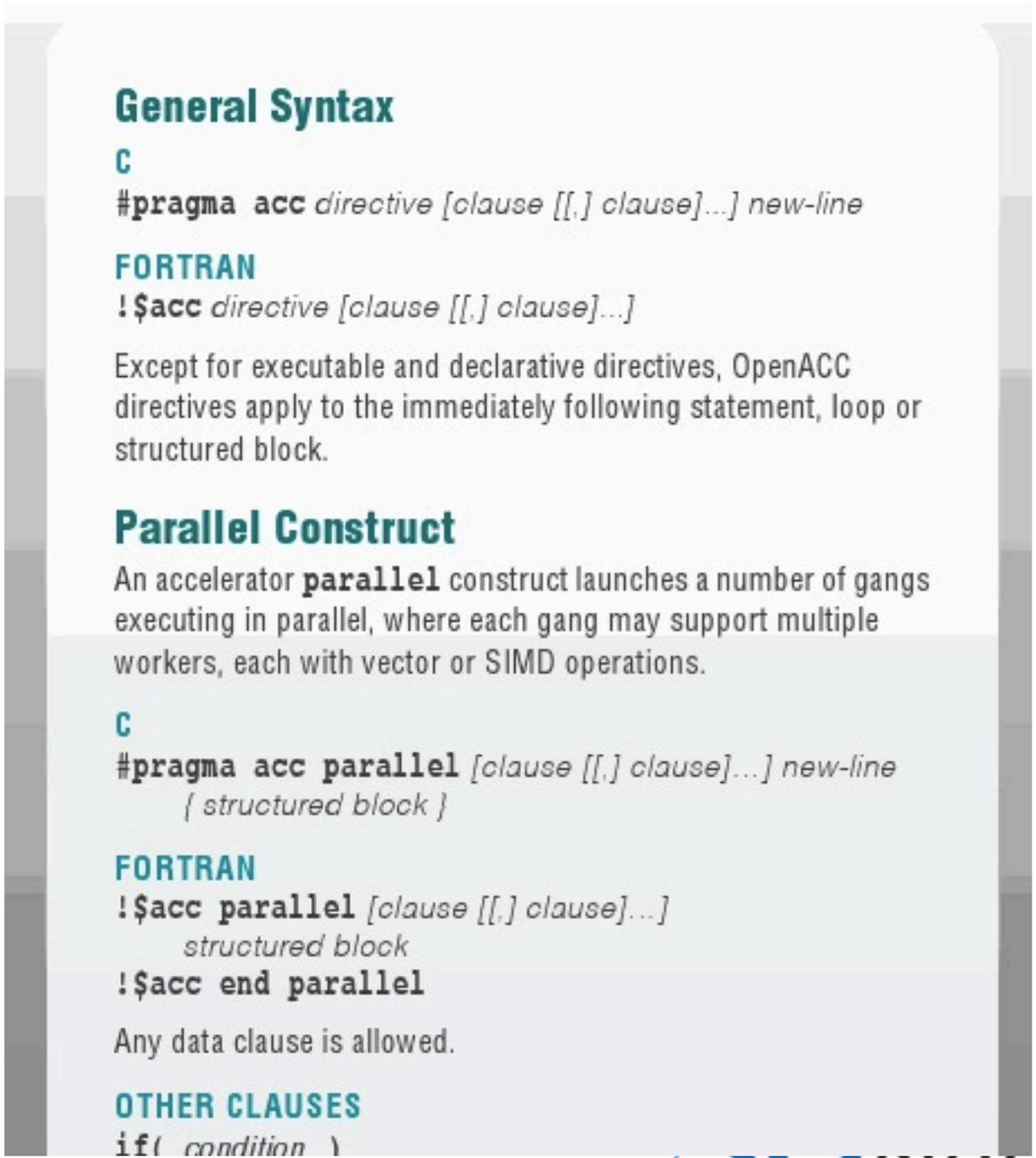
- NVidia (http://www.nvidia.com/object/cuda_opencl_new.html )
- Mac Research ( http://macresearch.org/opencl)
- Apple (http://developer.apple.com/ )
- Book ( http://www.fixstars.com/en/company/books/opencl/ )
- AMD ( http://developer.amd.com/zones/OpenCLZone )

# Directives-Based GPGPU Programming: PGI Accelerator


# OpenACC

http://www.openacc-standard.org/

- CAPS, Cray, PGI, NVidia
- OpenMP for GPU
- Take existing proprietary compilers, find a minimal common feature set, implement
- Summer 2012



**General Syntax**

C

```
#pragma acc directive [clause [[,] clause]...] new-line
```

FORTRAN

```
!$acc directive [clause [[,] clause]...]
```

Except for executable and declarative directives, OpenACC directives apply to the immediately following statement, loop or structured block.

**Parallel Construct**

An accelerator **parallel** construct launches a number of gangs executing in parallel, where each gang may support multiple workers, each with vector or SIMD operations.

C

```
#pragma acc parallel [clause [[,] clause]...] new-line
    { structured block }
```

FORTRAN

```
!$acc parallel [clause [[,] clause]...]
    structured block
!$acc end parallel
```

Any data clause is allowed.

OTHER CLAUSES

```
if( condition )
```

CITA|ICAT

# PGI Accelerator

- Builds on strong OpenMP, vector performance
- On SciNet ARC:
  `module load use.experimental pgi`

```
!$acc region
      do k = 1,n1
       do i = 1,n3
        c(i,k) = 0.0
        do j = 1,n2
         c(i,k) = c(i,k) + a(i,j) * b(j,k)
        enddo
       enddo
      enddo
!$acc end region
```


http://www.pgroup.com/resources/accel.htm

CITA|ICAT

SciNet

# #pragma acc

- Define code region, data region, scheduling.
- In some cases, (esp Fortran) `#pragma acc region` + static analysis can do some pretty amazing things

```
32 !$acc region
33         do j=1,N
34             c(:,j) = 0.
35             do k=1,N
36                 do i=1,N
37                     c(i,j) = c(i,j) + a(i,k)*b(k,j)
38                 enddo
39             enddo
40         enddo
41 !$acc end region
```

```
$ pgf90 -Minfo=all,accel -fast -ta=nvidia,cc2.0 -o matrixf-gpu matrixf.f90

     30, Memory zero idiom, array assignment replaced by call to pgf90_mzero8
     31, Loop not vectorized/parallelized: too deeply nested
     32, Generating copyout(c(1:n,1:n))
         Generating copyin(b(1:n,1:n))
         Generating copyin(a(1:n,1:n))
     33, Loop is parallelizable
     34, Loop is parallelizable
         Accelerator kernel generated
         33, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
         34, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
         Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
     35, Loop carried reuse of 'c' prevents parallelization
     36, Loop is parallelizable
         Accelerator kernel generated
         33, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
         35, !$acc do seq(16)
             Cached references to size [16x16] block of 'a'
             Cached references to size [16x16] block of 'b'
         36, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
         Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
         Generated 2 prefetch instructions for the loop
```

# Data clauses

- `#pragma acc copyin()` - copies in array (or subarray) from host-kernel
- `copyout()` - copies out data
- `copy()` - in and out
- `local()` - GPU-local array, doesn't need to be copied
- `region` - one or more kernels

```
$ pgf90 -Minfo=all,accel -fast -ta=nvidia,cc2.0 -o matrixf-gpu matrixf.f90

     30, Memory zero idiom, array assignment replaced by call to pgf90_mzero8
     31, Loop not vectorized/parallelized: too deeply nested
     32, Generating copyout(c(1:n,1:n))
         Generating copyin(b(1:n,1:n))
         Generating copyin(a(1:n,1:n))
     33, Loop is parallelizable
     34, Loop is parallelizable
         Accelerator kernel generated
         33, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
         34, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
         Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
     35, Loop carried reuse of 'c' prevents parallelization
     36, Loop is parallelizable
         Accelerator kernel generated
         33, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
         35, !$acc do seq(16)
             Cached references to size [16x16] block of 'a'
             Cached references to size [16x16] block of 'b'
         36, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
         Generated 3 alternate versions of the loop
         Generated vector sse code for the loop
         Generated 2 prefetch instructions for the loop
```

# Data clauses

- `#pragma acc for parallel(N)` - breaks loop up between blocks, with blocksize N
- `#pragma acc for vector(N)` - parallelizes loop over threads in block, with vector size `Nregion` - one or more kernels
- `unroll` - force unrolling; `kernel` - force separate kernel

```
double jacobi(double **restrict a, const int N, const int niters) {

    const double w0 = -15./6., w1 = 4./3., w2 = -1./12;
    double **restrict tmp = twodMalloc(N,N);
    double maxchange;
    #pragma acc data region copyout(tmp[0:N-1][0:N-1]) \
                            copyin(a[0:N-1][0:N-1]) \
                            copyout(maxchange)
    {
    for (int iter=0; iter<niters; iter++) {
        maxchange = 0.;
        {
        #pragma acc region
        for (int i=1; i<N-1; ++i)
            for (int j=1; j<N-1; ++j) {
                tmp[j][i] = w0*a[j][i] +
                      w1 * (a[j-1][i] + a[j][i-1] +
                            a[j+1][i] + a[j][i+1]) +
                     w2 * (a[j-1][i-1] + a[j+1][i-1] +
                            a[j+1][i+1] + a[j-1][i+1]);
                double change = fabs(tmp[i][j] - a[i][j]);
                maxchange = fmax(maxchange, change);
            }

        #pragma acc region
        for (int i=0; i<N-1; ++i)
            for (int j=0; j<N-1; ++j)
                a[j][i] = tmp[j][i];
        }
    }
    }
    twodFree(tmp);
    return maxchange;
}
```

```
double jacobi(double **restrict a, const int N, const int niters) {

    const double w0 = -15./6., w1 = 4./3., w2 = -1./12;
    double **restrict tmp = twodMalloc(N,N);
    double maxchange;
    #pragma acc data region copyout(tmp[0:N-1][0:N-1]) \
                            copyin(a[0:N-1][0:N-1]) \
                            copyout(maxchange)
    {
    for (int iter=0; iter<niters; iter++) {
        maxchange = 0.;
        {
        #pragma acc region
        for (int i=1; i<N-1; ++i)
            for (int j=1; j<N-1; ++j) {
                tmp[j][i] = w0*a[j][i] +
                      w1 * (a[j-1][i] + a[j][i-1] +
                            a[j+1][i] + a[j][i+1]) +
                    w2 * (a[j-1][i-1] + a[j+1][i-1] +
                            a[j+1][i+1] + a[j-1][i+1]);
                double change = fabs(tmp[i][j] - a[i][j]);
                maxchange = fmax(maxchange, change);
            }

        #pragma acc region
        for (int i=0; i<N-1; ++i)
            for (int j=0; j<N-1; ++j)
                a[j][i] = tmp[j][i];
        }
    }
    }
    twodFree(tmp);
    return maxChange;
}
```

All pointers need to be **restrict**

```
double jacobi(double **restrict a, const int N, const int niters) {

    const double w0 = -15./6., w1 = 4./3., w2 = -1./12;
    double **restrict tmp = twodMalloc(N,N);
    double maxchange;
    #pragma acc data region copyout(tmp[0:N-1][0:N-1]) \
                            copyin(a[0:N-1][0:N-1]) \
                            copyout(maxchange)
    {
    for (int iter=0; iter<niters; iter++) {
        maxchange = 0.;
        {
        #pragma acc region
        for (int i=1; i<N-1; ++i)
            for (int j=1; j<N-1; ++j) {
                tmp[j][i] = w0*a[j][i] +
                      w1 * (a[j-1][i] + a[j][i-1] +
                            a[j+1][i] + a[j][i+1]) +
                    w2 * (a[j-1][i-1] + a[j+1][i-1] +
                            a[j+1][i+1] + a[j-1][i+1]);
                double change = fabs(tmp[i][j] - a[i][j]);
                maxchange = fmax(maxchange, change);
            }

        #pragma acc region
        for (int i=0; i<N-1; ++i)
            for (int j=0; j<N-1; ++j)
                a[j][i] = tmp[j][i];
        }
    }
    }
    twodFree(tmp);
    return maxChange;
}
```

A data region can contain multiple code regions

```
double jacobi(double **restrict a, const int N, const int niters) {

    const double w0 = -15./6., w1 = 4./3., w2 = -1./12;
    double **restrict tmp = twodMalloc(N,N);
    double maxchange;
    #pragma acc data region copyout(tmp[0:N-1][0:N-1]) \
                            copyin(a[0:N-1][0:N-1]) \
                            copyout(maxchange)
    {
    for (int iter=0; iter<niters; iter++) {
        maxchange = 0.;
        {
        #pragma acc region
        for (int i=1; i<N-1; ++i)
            for (int j=1; j<N-1; ++j) {
                tmp[j][i] = w0*a[j][i] +
                      w1 * (a[j-1][i] + a[j][i-1] +
                            a[j+1][i] + a[j][i+1]) +
                     w2 * (a[j-1][i-1] + a[j+1][i-1] +
                            a[j+1][i+1] + a[j-1][i+1]);
                double change = fabs(tmp[i][j] - a[i][j]);
                maxchange = fmax(maxchange, change);
            }

        #pragma acc region
        for (int i=0; i<N-1; ++i)
            for (int j=0; j<N-1; ++j)
                a[j][i] = tmp[j][i];
        }
    }
    }
    twodFree(tmp);
    return maxChange;
}
```

This is recognized as a reduction and implemented

```
double jacobi(double **restrict a, const int N, const int niters) {

    const double w0 = -15./6., w1 = 4./3., w2 = -1./12;
    double **restrict tmp = twodMalloc(N,N);
    double maxchange;
    #pragma acc data region copyout(tmp[0:N-1][0:N-1]) \
                            copyin(a[0:N-1][0:N-1]) \
                            copyout(maxchange)
    {
    for (int iter=0; iter<niters; iter++) {
        maxchange = 0.;
        {
        #pragma acc region
        for (int i=1; i<N-1; ++i)
            for (int j=1; j<N-1; ++j) {
                tmp[j][i] = w0*a[j][i] +
                      w1 * (a[j-1][i] + a[j][i-1] +
                            a[j+1][i] + a[j][i+1]) +
                    w2 * (a[j-1][i-1] + a[j+1][i-1] +
                            a[j+1][i+1] + a[j-1][i+1]);
                double change = fabs(tmp[i][j] - a[i][j]);
                maxchange = fmax(maxchange, change);
            }

        #pragma acc region
        for (int i=0; i<N-1; ++i)
            for (int j=0; j<N-1; ++j)
                a[j][i] = tmp[j][i];
        }
    }
    }
    twodFree(tmp);
    return maxchange;
}
```

Note funny loop ordering

```
double jacobi(double **restrict a, const int N, const int niters) {

    const double w0 = -15./6., w1 = 4./3., w2 = -1./12;
    double **restrict tmp = twodMalloc(N,N);
    double maxchange;
    #pragma acc data region copyout(tmp[0:N-1][0:N-1]) \
                            copyin(a[0:N-1][0:N-1]) \
                            copyout(maxchange)
    {
    for (int iter=0; iter<niters; iter++) {
        maxchange = 0.;
        {
        #pragma acc region
        for (int i=1; i<N-1; ++i)
            for (int j=1; j<N-1; ++j) {
                tmp[j][i] = w0*a[j][i] +
                      w1 * (a[j-1][i] + a[j][i-1] +
                            a[j+1][i] + a[j][i+1]) +
                    w2 * (a[j-1][i-1] + a[j+1][i-1] +
                            a[j+1][i+1] + a[j-1][i+1]);
                double change = fabs(tmp[i][j] - a[i][j]);
                maxchange = fmax(maxchange, change);
            }

        #pragma acc region
        for (int i=0; i<N-1; ++i)
            for (int j=0; j<N-1; ++j)
                a[j][i] = tmp[j][i];
        }
    }
    }
    twodFree(tmp);
    return maxChange;
}
```

Multid loops must be rectangular

# Cannot appear in compute region:

- jumps out of region
- non-inlinable function calls

# Compiler feedback

- non-stride-1 access (uncoallesced)
- loop schedule (can be overridden)
- inferred data movement
- lack of parallelization due to - required privitization, data dependency
  - Not enough feedback here!

# accellmath

- `#include <accelmath.h>`
- fast GPU math library (exp, sin, cos...)

# accel.h

- `#include <accel.h>`
- Gives you access to
  `acc_get_num_devices()`,
  `acc_set_device_num()`
  - Can also set by environment variable
  - Each thread/MPI task can set own device

# OpenACC

- Version 1.0 this summer (12.8)
- Early betas now available
- OpenACC-inspired features now part of a technical report proposed for OpenMP 4.0
- Still need to understand what's going on under the hood (CUDA) to make code fast