Introduction to Compute Abstraction Layer (CAL)

Praveen Bhaniramka
Module Overview

- CAL Overview
- CAL Runtime API
- CAL Compiler API
- CAL Kernel Execution
Learning Objectives

• Understand CAL architecture from a high-level

• Understand the flow of a typical CAL application

• Review and understand various components involved in a CAL application and their characteristics

• Used a simple example to see how to write a simple CAL application
FireStream SDK

Stream Applications

Libraries
- ACML

3rd Party Tools
- RapidMind

Compilers
- Brook +

Profilers
- Shader Analyzer

API Bindings
- 3D Graphics Inter-operability

AMD Runtime

Compute Abstraction Layer

Close-to-the-Metal

AMD Multicore CPUs

AMD Stream Processors
CAL Overview

- Device driver library for AMD Stream Processors
- Programming Model (similar to Brook+, just lower-level)
  - Decouples program compilation from execution completely
    - Programs can be compiled off-line and binaries can later be loaded on the GPU
  - Binary is GPU architecture dependent
- Consists of two main components
  - amdcalcl – Compiler to generate machine binary
  - amdcalrt – Runtime infrastructure for executing GPU programs
CAL Overview

• Provides interface for
  – Compiling GPU kernels
  – Running kernels on the GPU
  – Transferring data between the host and the GPU
  – Managing GPU Local resources
  – Managing Host-side GPU resources
CAL System Architecture

- Host Application
- GPU Kernel
- GPU Local Resources
- Data Transfer
- System Memory
- PCI Express
- Host-side GPU Resources
- GPU Local Memory
CAL Application Architecture

- Consists of two distinct components
  - Host Application
    - Performs application work
    - Sends commands to the GPU using CAL API
    - Manages GPU and Host-side resources
  - GPU Kernel
    - Reads input data
    - Performs parallel computation
    - Writes output data
CALDevice

- Abstraction for a Stream Processor
- Initialized *explicitly* by application
- Attributes can be queried via API
- Executes a GPU Kernel
- Reads inputs and writes to outputs
- Accesses both GPU-local as well as Host-side GPU resources
CALResource

- Abstraction for memory buffer accessible by GPU
  - Local resource – resident on GPU Local memory
  - Remote resource – resident in Host-side GPU memory (PCIe memory)

- Allocated *explicitly* by application

- Can be specific to a device or shared between multiple devices

- Can be 1D or 2D

- Can be used as inputs, outputs or constants

- Total available and maximum size are fixed and can be queried

- Attributes include dimensions, data format and data type
Remote Resources

PCIe

Device can access both

Local Interconnect

Local Resources

Remote Memory

Device 1
Program
Local Memory

Device 2
Program
Local Memory
Remote Resources

- GPU can address and access system memory directly
  - Not all of system memory is accessible from GPU

- Driver allocates dedicated system memory for GPU buffers
  - Can be accessed directly from *GPU kernel* for reading and writing
  - Available in GPU’s address space
  - Application can request allocation from this *pool*
  - Remote resources can be mapped to application’s address space but the reverse is not possible

⇒ If your data is in application’s address space, it needs to be copied to GPU accessible memory first
⇒ It is more efficient to allocate from remote resource directly to avoid the extra copies to and from application address space
Local vs Remote Resources

Not all Host memory is addressable by GPU

Extremely Fast access to Local memory
CALImage

- Abstraction for GPU binary
- Created by compiling and linking multiple GPU-specific CAL objects
- Loaded on GPU using CAL API
- Can be stored for subsequent loading
Recap

• **CALDevice** – Abstraction for a GPU

• **CALResource** – Abstraction for GPU memory
  - *Inputs* – Read-only resources accessed by kernel
  - *Outputs* – Write-only resources written by kernel
  - *Constants* – Read-only constant values used by kernel

• **CALImage** – Binary image for the GPU kernel

• Others...
CAL vs Brook+

**Brook+**
- CALDevice
- CALResource
  - Local
  - Remote
- CALImage

**CAL**
- No API construct
  - Hidden in Runtime Backend
- Stream
- Kernel
CAL Advantages

- Low-level control over GPU resources and operations
  - Potentially higher performance
    Hand-tuned kernels
    Low-level data management
  - Access to features not exposed by higher-level Brook+
  - Explicit platform specific optimizations
CAL Runtime API - Conventions

- ‘C’ Library

- Routines
  - Runtime prefixed `cal`, e.g. `calInit`
  - Return an error code of type `CALresult`

- Types prefixed `CAL`, e.g. `CALfloat`

- Enums prefixed `CAL_`, e.g. `CAL_FORMAT_FLOAT_1`

- Opaque handles to internal data structures prefixed `CAL`, e.g. `CALdevice`
CAL Application – High-Level Flow

1. Open CAL Device
2. Allocate and Setup Resources
3. Perform Device-specific setup
4. Compile Kernel
5. Setup Kernel Parameters
6. Run Kernel!

Done Implicitly in Brook+ CAL Runtime

Done Offline Using brcc
**CAL System Initialization**

`calInit`

```c
CALresult calInit(CALvoid);
```

- Initializes the CAL system, detect GPUs, etc
- Needs to be the first _CAL routine_ in an application
- Returns `CAL_RESULT_ERROR` otherwise
- Needs to be called only once

`calShutdown`

```c
CALresult calShutdown(CALvoid);
```

- Last CAL routine in an application
CAL System Query

calGetVersion

CALresult calGetVersion(CALuint* major, CALuint* minor, CALuint* imp);

- Query the version of the CAL runtime

calDeviceGetCount

CALresult calDeviceGetCount(CALuint* count);

- Query the number of GPUs on the system
- Each GPU is identified by a unique integer ID \([0..N-1]\)
- See/Run FindNumDevices tutorial program
CAL Error Handling

- All CAL routines return `CAL_RESULT_OK` on success and an error code on failure

- `calGetErrorString` can be used to get further information about the error

```c
const CALchar* calGetErrorString(void);
```

- Returns a NULL-terminated string on the last error that occurred
CAL Device Query

calDeviceGetInfo

CALAPI CALresult CALAPIENTRY calDeviceGetInfo(

    CALdeviceinfo* info, // struct to be filled by the runtime
    CALuint ordinal);    // Which device? [0.. N-1]

• Get information on a given device

struct {
    CALtarget target;
    CALuint maxResource1DWidth;
    CALuint maxResource2DWidth;
    CALuint maxResource2DHeight;
} CALdeviceinfo;

GPU ASIC Type e.g. CAL_TARGET_770

Maximum legal dimensions of 1D and 2D resources, e.g. 8096 for 770
CAL Device Query

- E.g. querying the type of GPU on the system

```c
// Get the information on the 0th device
CALdeviceinfo info;
if(calDeviceGetInfo(&info, 0) != CAL_RESULT_OK)
    ERROR_OCCURRED();

switch(info.target)
{
    case CAL_TARGET_670:
        fprintf(stdout, "Device Type = GPU R6V70\n");
        break;
    case CAL_TARGET_770:
        fprintf(stdout, "Device Type = GPU RV770\n");
        break;
}
```
CAL Device Query

calDeviceGetAttribs

CALresult calDeviceGetAttribs(
    CALdeviceattribs* attribs, // filled by the runtime
    CALuint ordinal);          // Which device? [0.. N-1]

• Get detailed device attributes on a given device

struct {
    CALuint struct_size;       // Size of struct
    CALtarget target;            // ASIC identifier, e.g. R670
    CALuint localRAM;          // Local GPU RAM in MB, e.g. 800MB
    CALuint uncachedRemoteRAM; // Uncached remote memory in MB
    CALuint cachedRemoteRAM;   // Cached remote memory in MB
    CALuint engineClock;       // GPU device clock rate in MHz
    CALuint memoryClock;       // GPU memory clock rate in MHz
    ......
} CALdeviceattribs;
**CAL Device Status Query**

calDeviceGetStatus

CALresult calDeviceGetStatus(
    CALdevicestatus* status, // filled by the runtime
    CALdevice device); // Which device? [0.. N-1]

- Get the current status of the given device

struct {
    CALuint struct_size; // Size of struct
    CALuint availLocalRAM; // Available local memory
    CALuint availUncachedRemoteRAM; // Available uncached
    // resource
    CALuint availCachedRemoteRAM; // Available cached resource
} CALdevicestatus;
CAL Device Management

calDeviceOpen
CALresult calDeviceOpen(CALdevice* dev, CALuint ordinal);

- Initialize the device

- Initializes dev on success with handle to device

- Returns CAL_RESULT_ERROR otherwise

calDeviceClose
CALresult calDeviceClose(CALdevice dev);

- Un-initialize the device

- See/Run OpenCloseDevice tutorial program
CAL Resource Allocation

calResAlloc{ ε Local Remote }{ ε 1 2 }D([args]);

- CAL Resource Allocation routines, e.g.

`calResAllocLocal2D`

CALresult `calResAllocLocal2D`

```c
CALresource* res,
CALdevice dev,
CALuint width,
CALuint height,
CALformat format,
CALuint flags);
```

- Allocates 2D resource from GPU local memory
- Returns opaque handle to resource `CALResource`
- Can be de-allocated using `calResFree`
CAL Resource Allocation

- Both Local and Remote resources are limited
  - Maximum size of a buffer is given by `calDeviceGetInfo`
  - Total size is given by `calDeviceGetAttribs`

- Always check the returned error code and handle value before using the resource
CAL Resource Format

- Resources can be of different data types
- Resources can have 1, 2 or 4 components per element
- Formats specified using enum CALformat
- Components are arranged in interleaved order
- E.g. following allocates 1024x1024 size resource with 4 32-bit floating point values per element

```c
CALresource resLocal = 0;
calResAllocLocal2D(&resLocal, device,
                    1024, 1024,
                    CAL_FORMAT_FLOAT_4, 0);
```
CAL Resource Format

- **CALformat syntax**

  \[
  \text{CAL\_FORMAT\_\{e UBYTE BYTE USHORT SHORT UINT INT FLOAT DOUBLE\_\{e 1 2 4\}}
  \]

- **E.g.** `CAL\_FORMAT\_FLOAT\_1, CAL\_FORMAT\_BYTE\_4`

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UBYTE, BYTE</td>
<td>Signed/unsigned 8-bit integer</td>
</tr>
<tr>
<td>USHORT, SHORT</td>
<td>Signed/unsigned 16-bit integer</td>
</tr>
<tr>
<td>UINT, INT</td>
<td>Signed/unsigned 32-bit integer</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Single precision 32-bit floating point</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Double precision 64-bit floating point</td>
</tr>
</tbody>
</table>

- **Note:** `CAL\_FORMAT\_DOUBLE\_4` is not supported
CAL Resource Format

FLOAT_1
Each element has 1 float value

C-Style LINEAR memory arrangement in row-major order

```
calResAllocLocal2D(&resLocal, device, 8, 8,
                   CAL_FORMAT_FLOAT_1, 0);
```
CAL Resource Format

**FLOAT_4**
Each element has 4 float values

C-Style LINEAR memory arrangement in row-major order

```c
calResAllocLocal2D(&resLocal, device, 2, 8,
                   CAL_FORMAT_FLOAT_4, 0);
```
Remote Resource Allocation

calResAllocRemote2D
• Allocates 2D resource from GPU remote memory
• Resource can be shared by more than 1 device

CALresult calResAllocRemote2D(
    CALresource* res,
    CALdevice *dev,
    CALuint deviceCount,
    CALuint width,
    CALuint height,
    CALformat format,
    CALuint flags);

e.g. calResAllocRemote2D(&resRemote, &device, 1, 1024, 1024,
CAL_FORMAT_FLOAT_1, 0);
Remote vs Local Resources

- Remote pool for Device 1
- Remote pool shared by multiple devices
- Remote pool for Device n

Device 1
- Kernel
- Local Resources exclusive to Device 1

Device n
- Kernel
- Local Resources exclusive to Device n
Resource Initialization

- Initialization requires accessing resource from CPU
- Done using `calResMap` that maps `CALResource` to the application’s address space

```c
CALresult calResMap(
    CALvoid** pPtr,
    CALuint* pitch,
    CALresource res,
    CALuint flags);
```

- Returned CPU pointer
- Number of elements in each row of mapped buffer
- Input resource to be mapped
- Flags (placeholder for future use)

e.g.
```c
float *dataPtr = NULL;
CALuint pitch = 0;
calResMap((CALVoid**) &dataPtr, &pitch, resLocal, 0);
```
Resource Initialization

- Pitch is the number of elements in a row
  - Number of elements not necessarily equal to requested resource width
  - Actual size of buffer allocated for a 2D resource is given by

\[
\text{Allocated Buffer Size} = \text{Pitch} \times \text{Height} \times \text{Number of components} \times \text{Size of data type}
\]
Resource Initialization

- Important when dereferencing and addressing the returned pointer, e.g.

  ```c
  for(int i = 0; i < height; i++) {
      // Use the pitch to properly offset into the memory pointer
      float* tmp = &dataPtr[i * pitch];
      for (int j = 0; j < width; j++) {
          // For FLOAT_1, we should initialize temp[j] only
          tmp[j] = (float)(i * width + j);
      }
  }
  
  - Need to unmap the resource once done with initialization
    - Mapped resource cannot be used in a kernel
    - Use `calResUnmap`
CAL Context

• CAL allows multiple connections to a device

• Each connection is associated with local settings
  – Collectively termed as the device state
  – Abstracted in CAL using CALContext

• All device operations need to be specific to a CALContext

• Can be thought of as a separate process on the device
  – Each context has its own address space
  – Only a single context can be active on the device at a time

All context-specific routines are prefixed calCtx, e.g. calCtxCreate
CAL Context

**calCtxCreate**

CALresult calCtxCreate(CALcontext* ctx, CALdevice dev);

- Create a context on the given device

**calCtxDestroy**

CALresult calCtxDestroy(CALcontext ctx);

- Destroy the specified context
- See/Run CreateContext tutorial program
Context-specific Resource Mapping

- GPU kernels need a valid `CALContext` to run
- Resources are accessed by kernels using a context-specific address
  - Represented using an opaque handle `CALMem`
  - Mapping done using `calCtxGetMem` that returns a context-specific handle to a resource

```
calCtxGetMem

CALresult calCtxGetMem(CALmem* mem, CALcontext ctx, CALresource res);
```

e.g.
```
CALmem memLocal = 0;
calCtxGetMem(&memLocal, ctx, resLocal);
```
CAL Application – High-Level Flow

1. Open CAL Device
2. Allocate and Setup Resources
3. Perform Device-specific setup
4. Compile Kernel
5. Setup Kernel Parameters
6. Run Kernel!

- CALdevice
- CALresource
- CALcontext, CALmem
CAL Compiler API - Conventions

- ‘C’ Library

- Routines
  - Compiler prefixed `calcl`, e.g. `calclCompile`
  - Return an error code of type `CALresult`

- Enums prefixed `CAL_`, e.g. `CAL_LANGUAGE_ISA`

- Opaque handles to internal data structures prefixed `CAL`, e.g. `CALObject`
CAL Compiler API

• Compiler allows offline as well as runtime compilation
  – Compiler API used to compile kernels at runtime
  – Compiled binary can be stored as a file and loaded later at runtime

• Typically,
  – Runtime compilation used for development
  – Pre-compiled binaries shipped during deployment
    Prevents overhead of kernel compilation at runtime
    Prevents exposing proprietary kernels
Kernel Programming

• CAL supports two interfaces directly
  – Portable pseudo assembly – AMD Intermediate Language (IL)
    $(CALROOT)/doc/il.pdf
  – GPU-specific Instruction Set Architecture (ISA)
    $(CALROOT)/doc/r600isa.pdf

• Higher level languages can be used with external tools/compilers
  – Brook+
  – AMD High Level Shading Language (HLSL)
Kernel Compilation Steps

1. Application
2. High-Level Language Translator (brcc)
3. AMD IL
4. CAL Compiler
5. GPU-Specific Binary
CAL Compiler Query and Error Handling

calclGetVersion

CALresult calclGetVersion(CALuint* major, CALuint* minor, CALuint* imp);

- Query the CAL Compiler version

calclGetErrorString

const CALchar* calclGetErrorString(void);

- Returns NULL terminated string with information about the last error
Kernel Compilation

calclCompile

- Compile a given kernel to a GPU-specific binary object
  - Returns an opaque handle CALObject

CALresult calclCompile(
  CALobject* obj,       // Generated binary object
  CALlanguage language,  // Kernel language, e.g. CAL_LANGUAGE_IL
  const CALchar* source,  // String for kernel source
  CALtarget target);     // Target GPU, e.g. CAL_TARGET_R770

- The AMD IL is parsed and optimized for the given GPU architecture

- The GPU ISA generated is internally assembled into the returned CALObject
Kernel Specification

- Need to specify as a C String
- Lines of code separated using new line characters

// IL kernel to implement Output = Input * Constant
std::string programIL =

"il_ps_2_0 \n"
"dcl_input_interp(linear) v0.xy__\n"
"dcl_output_generic o0 \n"
"dcl_cb cb0[1] \n"
"dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmyty(float)_fmtz(float)_fmtw(float) \n"
"sample_resource(0)_sampler(0) r0, v0.xyxx \n"
"mul o0, r0, cb0[0] \n"
"ret_dyn \n"
"end \n";
Using GPU ISA

calclAssembleObject

• Assemble the given GPU ISA into a binary object

CALresult calclAssembleObject(
    CALobject* obj,  // Generated binary object
    CALCLprogramType type, // Program type - CAL_PROGRAM_TYPE_PS
    const CALchar* source, // String for kernel source
    CALtarget target);  // Target GPU, e.g. CAL_TARGET_R670

• The GPU ISA is simply assembled into the CALobject - no optimizations are done
Kernel Compilation Steps

Psuedo-Assembly Programming Interface

```c
AMD IL
il_ps_2_0
dcl_output_generic o0
mov o0, v0.xyxx
ret_dyn
end
```

Application can specify IL or ISA directly
**Binary Image Generation**

`calclLink`

- Link the specified `CALObject`s into a binary image
- Returns an opaque handle `CALImage`

```c
CALresult calclLink(
    CALimage* image, // Returned binary image
    CALobject* obj,  // Array of objects to be linked
    CALuint objCount); // Number of objects in above array
```

- **E.g.** `calclLink(&image, &object, 1);`

- Allows linking-in multiple objects compiled for different GPU ASICs
Binary Image Management

• The generated CALimage can be stored for subsequent use
  – Need to copy the contents of CALimage to a regular C buffer and then stored

  calclImageWrite

  CALresult calclImageWrite(CALvoid* buffer, CALuint size, CALimage image);

• Serialize the content of CALimage to the supplied buffer

• buffer needs to be allocated by application

• size can be queried using calclImageGetSize
Binary Image Management

- Need to initialize the given C buffer to a CALImage

  calImageRead
  
  CALresult calImageRead(
    CALimage *image,       // Returned CALimage handle
    const CALvoid* buffer, // Buffer from which to
    CALuint size); // Size of buffer

- Create a new CALImage and serialize into it from the supplied buffer
Compiler Memory Management

- Memory for CALObject and CALImage are allocated internally by the compiler
  - Need to be free’ed once the memory is not needed

```c
CALresult calclFreeObject(CALObject obj);
CALresult calclFreeImage(CALImage image);
```

Free the object and image respectively
CAL Application – High-Level Flow

1. Open CAL Device
2. Allocate and Setup Resources
3. Perform Device-specific setup
4. Compile Kernel
5. Setup Kernel Parameters
6. Run Kernel!

Variables:
- CALdevice
- CALresource
- CALcontext, CALmem
- CALobject, CALimage
Kernel Loading

- The compiled CALimage is not bound to a CALcontext

```
calModuleLoad
```

- **Loads a given CALimage as a context-specific**
  CALmodule

```
CALresult calModuleLoad(
    CALmodule* module, // Returned handle to CALmodule
    CALcontext ctx, // Context in which to load the CALimage
    CALimage image); // Pre-linked CALimage
```

- **E.g.**

```
CALmodule module = 0;
calModuleLoad(&module, ctx, image);
```
Kernel Parameter Binding

• Need to link various CALmem handles to corresponding variable names in the kernel

```
il_ps_2_0
dcl_input_interp(linear) v0.xy__
dcl_output_generic o0
dcl_cb cb0[1]
mul o0, v0.xyxx, cb0[0]
ret_dyn
end
```

• Steps
  – Get handle to parameter names from module
  – Associate locations with memory handles
Kernel Parameter Binding

calModuleGetName

- Get handle to a given kernel parameter from the module
- Returns an opaque handle CALname given the parameter string

CALresult calModuleGetName(
    CALname* name,       // Returned handle to the parameter
    CALcontext ctx,      // CAL Context
    CALmodule module,    // CAL module loaded in the context
    const CALchar* varName); // string for parameter name

- E.g.

    CALname output;
    calModuleGetName(&output, ctx, module, "o0");
# Kernel Parameter Binding

<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Resource</td>
<td>i#</td>
<td>i0, i1, i2, ... i127</td>
</tr>
<tr>
<td>Output Resource</td>
<td>o#</td>
<td>o0, o1, o2, ... o7</td>
</tr>
<tr>
<td>Constant Resource</td>
<td>cb#</td>
<td>cb0, cb1, ... cb15</td>
</tr>
</tbody>
</table>
Constant Buffers

- Read-only resources, similar to Inputs
  - Used for small arrays of up to 4096 elements
  - Legal to have 1, 2 or 4 components per element.
  - All CAL formats mentioned in cal.h header file are supported except for 8-bit (BYTE, UBYTE) and 16-bit (SHORT, USHORT) types.
  - Can declare and use up to 16 1D resources as constant buffers. 2D constant buffers are not allowed.
  - Typically allocated using remote resources.

- Copied to a special cache on the GPU

  ⇒ Reading is more efficient than reading regular inputs.

  ⇒ For small input arrays, it is preferable to use constant buffers
Kernel Parameter Binding

calCtxSetMem

• Associate the given CALmem with the given CALname

CALresult calCtxSetMem(
    CALcontext ctx, // CAL context
    CALname name,   // Parameter name from calModuleGetName
    CALmem mem);    // Memory handle for the parameter

• E.g.

calCtxSetMem(ctx, input, inputMem);
CAL Application – High-Level Flow

1. Open CAL Device
2. Allocate and Setup Resources
3. Perform Device-specific setup
4. Compile Kernel
5. Setup Kernel Parameters
6. Run Kernel!

CALdevice
CALresource
CALcontext, CALmem
CALobject, CALimage
CALname, CALmem binding
Kernel Execution – Domain

- Kernel is executed over a given range of elements in the output buffer
  - Elements are processed in parallel on the GPU
  - Also referred to as the domain of execution
  - Represented in CAL using `CALdomain`

```c
struct {
    CALuint x; // x origin of domain
    CALuint y; // y origin of domain
    CALuint width; // width of domain
    CALuint height; // height of domain
} CALdomain;
```
Kernel Execution - Asynchronous

- CAL kernels run on the GPU in a separate thread of execution
  - CPU is free to perform other operations in parallel
- Routine used to launch kernel on the GPU is non-blocking
  - The routine does not wait for the GPU to finish
  - Application needs to synchronize the kernel completion explicitly
  - The above routine returns an event that the application can wait on using CALevent
Kernel Execution - Module Entry Point

- CAL runtime requires the entry point in the CALmodule for the kernel

- Represented using CALfunc
  
  `calModuleGetEntry`

- Returns the entry point for a given function in the binary module
  
  ```c
  CALresult calModuleGetEntry(
      CALfunc* func, // Returned handle to function
      CALcontext ctx, // CAL context
      CALmodule module, // CAL module
      const CALchar* procName); // Function name - "main"
  ```

- E.g.
  
  ```c
  CALfunc entry = 0;
  calModuleGetEntry(&entry, ctx, module, "main");
  ```
Kernel Execution – Launching

calCtxRunProgram

CALresultado calCtxRunProgram(
    CALevent* event, // Returned event for this command
    CALcontext ctx,  // CALcontext
    CALfunc func,    // Entry point in the module
    const CALdomain* domain); // Execution domain

- Launch the kernel on the GPU associated with the specified context

- The routine returns almost immediately with the CALevent for synchronizing with the application thread
Kernel Execution – Synchronization

calCtxIsEventDone

CALresult calCtxIsEventDone(CALcontext ctx, CALevent event);

• Poll the CALevent for completion

• Returns CAL_RESULT_PENDING if the event is not complete

• Typical execution:

  while(calCtxIsEventDone(ctx, event) == CAL_RESULT_PENDING);

• Note: the above is a busy wait that wastes CPU cycles
  – It might be better to perform some CPU work before polling for the event
Kernel Execution – Synchronization

- Important note
  - CAL implements a lazy evaluation scheme
    ⇒ Tries to queue up commands before dispatching them to the GPU
    ⇒ `calCtxIsEventDone` performs an *implicit flush* to force the dispatch of commands to the GPU
    ⇒ Calling `calCtxIsEventDone` once after the `calCtxRunProgram` guarantees that the command is sent to the GPU

```c
    calCtxRunProgram(&event, ctx, entry, &domain);
    // Flush commands
    calCtxIsEventDone(ctx, event);
    // Do other CPU work
    ....
    // Wait for completion
    while(calCtxIsEventDone(ctx, event) == CAL_RESULT_PENDING);
```
CAL Application – High-Level Flow

1. Open CAL Device
2. Allocate and Setup Resources
3. Perform Device-specific setup
4. Compile Kernel
5. Setup Kernel Parameters
6. Run Kernel!

Variables:
- CALdevice
- CALresource
- CALcontext, CALmem
- CALobject, CALimage
- CALname, CALmem binding
- CALfunc, CALevent
End of Module 5