Introductions

Jason Yang
• Ph.D. from Massachusetts Institute of Technology
• Thesis focused on:
  − Light fields

Justin Hensley
• Ph.D. from University of North Carolina at Chapel Hill
• Thesis focused on:
  − Asynchronous (clockless) logic for graphics processors
  − Summed area table generation and applications for graphics
  − Ray-tracing using conditional streams

Advanced technology development focusing on GPGPU
Lecture Overview

A short history lesson - Close To the Metal

The Compute Abstraction Layer

R600 ISA
- R610 - Radeon HD 2400
- R630 - Radeon HD 2600
- R600 - Radeon HD 2800
- R670 - Radeon HD 3870 (Support for Double Precision!)

More advanced CAL examples

AMD IL
AMD HLSL

Questions and Discussion
CTM - Close to the Metal

CTM Goals:

Expose relevant parts of the GPU as they really are
• Command Processor
• Data Parallel Processor(s) (e.g. 1 or more SIMDs)
• Memory Controller

Hide all other graphics-specific features

Provide direct communication to device

Eliminate driver implemented procedural API
• Push policy decisions back to application
• Remove constraints imposed by graphics APIs

User responsible for memory management
• Basically given a pointer to giant block of memory

User responsible for generating binary for chip ISA
• Some helper functions included

User responsible for command buffer creation
• Command buffer - list of commands used by the GPUs command processor to the rest of the GPU

User responsible for command buffer submission
• Includes verifying that command buffer has finished before readback
The Evolution of CTM

Good first step, but....
• First version of the API tied too close to hardware, but not close enough

CTM has evolved into two pieces
• HAL : Hardware Abstraction Layer  
  – Device specific, driver like interface  
• CAL : Compute Abstraction Layer  
  – Core API device independent

Optimized multi-core implementation as well as optimized GPU implementations

Heterogeneous computing

The Evolution of CTM

The SDK - alive and well
• A bottom up approach  
• Give application developers low-level access to the GPU for those that want it  
• Provide high-level implementations to those that don’t want low-level access  
• Developers free to implement their own language(s) & environment(s)

The CTM API - evolved into CAL
• Compute Abstraction Layer  
• CAL maintains the flavor of the CTM API  
• Distributed as part of the *AMD Stream Computing SDK*
CAL - Compute Abstraction Layer

Stream Computing SDK

Stream Applications

Compilers
- Stream Extensions for C, C++

Libraries
- ACHL (Math Library)
- AMD Video transcoder library

Eco System
- 3rd Party Developers

AMD Runtime
- AMD Compute Abstraction Layer (CAL)

CTM HAL
- AMD Stream Processors

AMD Multicore CPUs
Example Application 1

Face recognition
- Recognition system uses CAL
- Interoperability with graphics API

Example Application 2

Real-time depth extraction + physics
More details given in sketch Thursday afternoon
CAL Highlights

Memory managed
• Don’t have to manually maintain offsets, etc
• Asynchronous DMA: CPU→GPU, GPU→GPU, GPU→CPU
• Multiple GPUs can share the same “system” memory

Core CAL API is device agnostic

Enables multi-device optimizations
• e.g. Multiple GPUs working together concurrently
• Multiple GPUs show up as multiple CAL devices

Extensions to CAL provide opportunities for device specific optimization

---

**Hello CAL**

```c
int main(int argc, char** argv)
{
    // Initializing CAL
    calInit();
    //-------------------------------------------------------------------------
    // Querying and opening device
    //-------------------------------------------------------------------------
    // Finding number of devices
    CALuint numDevices = 0;
    calDeviceGetCount(&numDevices);
    // Opening device
    CALdevice device = 0;
    calDeviceOpen(&device, 0);
    // Querying device attribs
    CALdeviceattribs attribs;
    calDeviceGetAttribs(&attribs, 0);
    // Creating context w.r.t. to opened device
    CALcontext ctx = 0;
    calCtxCreate(&ctx, device);
}
```
Hello CAL

// Compiling Device Program
//------------------------------------------------------------
CALobject obj = NULL;
CALimage image = NULL;
CALlanguage lang = CAL_LANGUAGE_IL;
std::string program = programIL;
std::string kernelType = "IL";
if (calclCompile(&obj, lang, program.c_str(), attribs.target) != CAL_RESULT_OK)
{
    fprintf(stdout, "Program compilation failed. Exiting.\n");
    return 1;
}
if (calclLink(&image, &obj, 1) != CAL_RESULT_OK)
{
    fprintf(stdout, "Program linking failed. Exiting.\n");
    return 1;
}

Hello CAL

// Allocating and initializing resources
//------------------------------------------------------------
// Input and output resources
CALresource inputRes = 0;
CALresource outputRes = 0;
calResAllocLocal2D(&inputRes, device, 256, 256, CAL_FORMAT_FLOAT_1, 0);
calResAllocLocal2D(&outputRes, device, 256, 256, CAL_FORMAT_FLOAT_1, 0);

// Constant resource
CALresource constRes = 0;
calResAllocRemote1D(&constRes, &device, 1, 1, CAL_FORMAT_FLOAT_4, 0);
**CAL: Memory**

**Remote Memory**
- Remote in the view of the GPU
- “System Memory”

**Local Memory**
- Local memory on for GPU

```
calResAlloc*1D()
calResAlloc*2D()
```
- Implies pitch can be used for better alignment

**CAL Memory System**

- CPU
  - CPU Memory (system / remote)

- GPU
  - GPU Memory (local)

- System Memory
  - Remote Memory
  - Local Memory
CAL: Memory Tiling

Linear Memory - Row-Major Order

Tiled Memory - Cache Friendly (multiple formats)

eexample: 4x4

Hello CAL

```
// Filling values in input buffer
float* fdata = NULL;
CALuint pitch = 0;
CALmem inputMem = 0;

// Mapping resource to CPU
// Returns 'fdata' as a CPU accessible pointer to resource 'inputRes'
calCtxGetMem(&inputMem, ctx, inputRes);
calResMap((CALvoid**)&fdata, &pitch, inputRes, 0);
for (int i = 0; i < 256; ++i)
{
    float* tmp = fdata[i * pitch];
    for (int j = 0; j < 256; ++j)
    {
        tmp[j] = (float)(i * pitch + j);
    }
}
calResUnmap(inputRes);
```
// Filling values in constant
float* constPtr = NULL;
CALuint constPitch = 0;
CALmem constMem = 0;

// Mapping resource to CPU
calCtxGetMem(&constMem, ctx, constRes);
calResMap((CALvoid**)&constPtr, &constPitch, constRes, 0);
constPtr[0] = 0.5f, constPtr[1] = 0.0f;
constPtr[2] = 0.0f, constPtr[3] = 0.0f;
calResUnmap(constRes);

// Mapping output resource to CPU and initializing values
void* data = NULL;

// Getting memory handle from resources
CALmem outputMem = 0;
calCtxGetMem(&outputMem, ctx, outputRes);
calResMap(&data, &pitch, outputRes, 0);
memset(data, 0, pitch * 256 * sizeof(float));
calResUnmap(outputRes);

//-------------------------------------------------------------------------
// Loading module and setting domain
//-------------------------------------------------------------------------

// Creating module using compiled image
CALmodule module = 0;
calModuleLoad(&module, ctx, image);

// Defining symbols in module
CALfunc func = 0;
CALname inName = 0, outName = 0, constName = 0;

// Defining entry point for the module
calModuleGetEntry(&func, ctx, module, "main");
calModuleGetName(&inName, ctx, module, "in");
calModuleGetName(&outName, ctx, module, "out");
calModuleGetName(&constName, ctx, module, "const");

// Setting input and output buffers
// used in the kernel
calCtxSetMem(ctx, inName, inputMem);
calCtxSetMem(ctx, outName, outputMem);
calCtxSetMem(ctx, constName, constMem);

// Setting domain
CALdomain domain = {0, 0, 256, 256};
Hello CAL

//-------------------------------------------------------------------------
// Executing program and waiting for program to terminate
//-------------------------------------------------------------------------

// Event to check completion of the program
CALevent e = 0;
calCtxRunProgram(&e, ctx, func, &domain);

// Checking whether the execution of the program is complete or not
while (calCtxIsEventDone(ctx, e) == CAL_RESULT_PENDING);

// Reading output from output resources
calResMap((CALvoid**)&fdata, &pitch, outputRes, 0);
for (int i = 0; i < 8; ++i)
{
    float* tmp = &fdata[i * pitch];
    for (int j = 0; j < 8; ++j)
    {
        printf("%f ", tmp[j]);
    }
    printf("\n");
}
calResUnmap(outputRes);

//-------------------------------------------------------------------------
// Cleaning up
//-------------------------------------------------------------------------

// Unloading the module
calModuleUnload(ctx, module);

// Freeing compiled program binary
calcFreeImage(image);
calcFreeObject(obj);

// Releasing resource from context
ca1ctxReleaseMem(ctx, inputMem);
calctxReleaseMem(ctx, constMem);
calctxReleaseMem(ctx, outputMem);

// Deallocating resources
calResFree(outputRes);
calResFree(constRes);
calResFree(inputRes);

// Destroying context
calCtxDestroy(ctx);

// Closing device
calDeviceClose(device);

// Shutting down CAL
ca1shutdown();

return 0;
GPU Shader Analyzer

*Great* tool for looking ISA generated by higher level languages
• Helps understand what’s going on in the hardware

Good for estimating performance
Good for testing code optimizations
• Can directly see how changes effect instruction count
**R600 ISA**

Three separate instruction streams
- ALU - arithmetic instructions (ADD, SUB, etc)
- TEX - loading data from memory
- COND - control flow

ALU operations organized as ‘clauses’
- Maximum size to clause
- Certain registers only valid for the clause
- Thread can not be ‘swapped-out’ while in a clause

---

**ALU Clause**

00 ALU: ADDR(32) CNT(9)
  0  x: ADD R123.x, R0.y, -0.5
  y: ADD R123.y, R0.x, -0.5
  t: RECIP_IEEE R127.w, C0.x
  1  z: ADD R123.z, PV(0).x, 0.5
  w: ADD R123.w, PV(0).y, 0.5
  t: RECIP_IEEE R122.z, C0.y
  2  x: MUL R0.x, PV(1).w, R127.w
  y: MUL R0.y, PV(1).z, PS(1).x
  t: RECIP_IEEE R0.w, C1.x
TEX clause

Threads ‘swapped-out’ on texture reads
• Separate hardware performs texture access
• Used to hide latency
• Can have multiple samples per clause

01 TEX: ADDR(64) CNT(1)
  3  SAMPLE R0.xy__, R0.xyxx, t1, s1  WHOLE_QUAD

Entire Assembly

; ------------------- PS Disassembly -------------------
00 ALU: ADDR(32) CNT(9)
  0  x: ADD  R123.x,  R0.y, -0.5
  y: ADD  R123.y,  R0.x, -0.5
  t: RECIP_IEEE R127.w,  C0.x
  1  x: ADD  R123.x,  PV(0).x, 0.5
  w: ADD  R123.w,  PV(0).y, 0.5
  t: RECIP_IEEE R122.z,  C0.y
  2  x: MUL  R0.x,  PV(1).w,  R122.w
  y: MUL  R0.y,  PV(1).z,  PS(1).x
  t: RECIP_IEEE R0.w,  C1.x
  01 TEX: ADDR(64) CNT(1)
  3  SAMPLE R0.xy__, R0.xyxx, t1, s1  WHOLE_QUAD
02 ALU: ADDR(41) CNT(5)
  4  x: ADD  R123.x,  R0.y, 0.5
  y: ADD  R123.y,  R0.x, 0.5
  t: RECIP_IEEE R122.z,  C1.y
  5  x: MUL  R0.x,  PV(4).y,  R0.w
  y: MUL  R0.y,  PV(4).x,  PS(4).x
  03 TEX: ADDR(66) CNT(1) VALID_PIX
  6  SAMPLE R0, R0.xyxx, t0, s0
04 ALU: ADDR(46) CNT(5)
  7  x: MOV  R1.x,  R0.x
  y: MOV  R1.y,  R0.y
  z: MOV  R1.z,  R0.z
  w: MOV  R1.w,  R0.w
  t: MOV  R1.x,  R1.x  POMERGE
  05 EXIT: DONE: END, HD
END_OF_PROGRAM
More Advanced CAL Example

Dense Matrix Multiplication

\[(AB)_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}\]

- Read \(n\) values
- Read \(n\) values
- Write 1 value

\(A(m,n) \times B(n,k) = C(m,k)\)

2\(n\) scalar operations
Basic Implementation - poor coherence

For each element:

```c
input A; // Input matrix (m, n) in size
input B; // Input matrix (n, k) in size
output C; // Output matrix (m, k) in size

void main()
{
    // Initialize the element to zero
    C[i][j] = 0.0;
    // Iterate over i'th row in matrix A and j'th column in matrix B
    // to compute the value of C[i][j]
    for (k=0; k<n; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

Improving cache hit ratio of the algorithm

• Divide the input and output matrices into sub-matrices
• Compute the product matrix one block at a time, by multiplying blocks from the input matrices.
Dense Matrix Multiply

Each matrix block now becomes 1/16 of the original matrix size
• divided into 4 blocks with 4 values per element
• The number of output values computed and written by each device kernel is 16.

Four-component input contains a 2x2 micro-tile of data values from the original
• pre-pass required if needed

The matrix multiplication done inside the loop essentially computes a 2x2 micro-tile in the output

Output is 4-component element
• post-process pass to reorder data if needed

Blocked Matrix Multiply - Code

Assumes tiling! :

```c
input A00, A01, A10, A11; // Input matrices (n/4, n/4) in size, 4-values per element
input B00, B01, B10, B11; // Input matrices (n/4, k/4) in size, 4-values per element
output C00, C01, C10, C11; // Output matrices (n/4, k/4) in size, 4-values per element

main() {
    // Iterate over i'th row in matrix A and j'th column in matrix B
    // to compute the values of C00[i,j], C01[i,j], C10[i,j] and C11[i,j]
    for (k = 0; k < n/8; k++)
    {
        C00[i,j].xyzw += A00[i,k].xxzz * B00[k,j].xyxy + A10[i,k].yyww * B01[k,j].zwzw;
        C10[i,j].xyzw += A00[i,k].xxzz * B10[k,j].xyxy + A10[i,k].yyww * B11[k,j].zwzw;
        C01[i,j].xyzw += A01[i,k].xxzz * B00[k,j].xyxy + A11[i,k].yyww * B01[k,j].zwzw;
        C11[i,j].xyzw += A01[i,k].xxzz * B10[k,j].xyxy + A11[i,k].yyww * B11[k,j].zwzw;
    }
}
```
Thread Safety

CAL Compiler routines are not thread-safe.
- Applications invoking compiler routines from multiple threads need to do proper synchronization to serialize the invocation of these routines.

CAL Runtime routines that are either context-specific or device-specific are thread-safe.
- All other CAL runtime routines are not thread-safe.

If the same context is shared among multiple threads, then invocation of the calCtx* functions must be serialized by the application.

Using multiple GPUs
Questions & Discussion

AMD Stream Computing SDK
• http://ati.amd.com/technology/streamcomputing/

• Fellowship deadline coming soon - February 15th
  – note: this is through the “Graphics Product Group”

• Internship opportunities
  – Compiler work
  – Language
  – CAL / Brook+ / ACML