Close to the Metal

Justin Hensley
Graphics Product Group

AMDA

GPGPU
Overview

• Sidebar: Quick R600 architecture overview
• Close to the Metal
  - The SDK vs API
• CTM API v1
• CTM SDK : Compute Abstraction Layer
• Conclusion and Questions
HD 2900 Architecture Overview (aka R600)

the short, short version
Unified Architecture Detail

- Command Processor
- Setup Engine
- Ultra-Threaded Dispatch Processor
- Stream Processing Units
- Texture Units & Caches
  - Memory Read/Write Cache & Stream Out Buffer
- Shader Export
- Render Back-Ends
Command Processor

- Command Processor
- Setup Engine
- Ultra-Threaded Dispatch Processor
- Stream Processing Units
- Texture Units
- Memory Read/Write Cache & Stream Out Buffer
- Shader Export
- Render Back-Ends

- First “layer” of GPU in design
- Handles command stream fetching, state management, including register subsystems
- Functional highlights:
  - Controls state and decodes command streams
  - Full memory client with Read/Write access
  - Multiple command queues
  - Multiple register interfaces, including graphics and system
  - DMA engine for command stream fetching
  - Interrupt system
Setup Engine

- **Pixel generation block**
  - “classic” setup and scan converter
  - Sends data to interpolators
  - Performs HiZ/ Early Z checks
  - Up to 16 fragments / cycle

- **Primitive block**
  - Sends post processed vertex addresses, near neighbor address and topological information to shader core
  - Up to 16 prims / cycle

- **Vertex block**
  - Performs tessellation
    - Supports different tessellation modes
  - Fetches vertex index streams and sends addresses to shader core
  - Up to 16 Verts / cycle
Dispatch Processor

- Command Processor
- Setup Engine
- Ultra-Threaded Dispatch Processor
- Stream Processing Units

**Main control for the shader core**
- Workloads distributed into threads of 64
- 100’s of threads in flight, to hide latency
- Threads sleep after requesting resources that take indefinite amounts of time

**A blend of arbitration and sequencing**
- Arbitration occurs for all resources
  - Occurs for different streams of work
  - ALUs execute pairs of threads interleaved
- Arbitration algorithms are programmable
Stream Processing Units

- Command Processor
- Setup Engine
- Ultra-Threaded Dispatch Processor
- Stream Processing Units
- Texture Units & Caches
- Memory Read/Write Cache & Stream Out Buffer
- Shader Export
- Render Back-Ends

- 4 SIMD Arrays of 16 SPUs

GPGPU AMD
Stream Processing Units

- **Command Processor**
- **Setup Engine**
- **Ultra-Threaded Dispatch Processor**
- **Stream Processing Units**
- **Texture Units & Caches**
- **Memory Read/Write Cache & Stream Out Buffer**
- **Shader Export**
- **Render Back-Ends**

- **4 SIMD Arrays of 16 SPUs**

- **SPU arranged as 5-way scalar shader processor**
  - Co-issue up to 5 scalar FP MAD (Multiply-Add)
  - One of the 5 stream processing units handles transcendental instructions as well
  - 32-bit floating point precision
  - Mul and Add are 1/2 ulp IEEE RTNE
  - Up to 5 integer operations also supported (cmp, add, and) and 1 integer multiplier

- **Issuing**
  - Each SPU works on 4 separate elements, issuing 5 scalar instructions over 4 cycles
  - BEU is issued to separately

- **Branch execution units handle flow control**
  - NOTE: Predication supported directly in ALU

- **General Purpose Registers**
  - Multi-ported but shared among the processors

[Diagram of a stream processing unit with various elements and connections]
Texture Cache

- Command Processor
- Setup Engine
- Ultra-Threaded Dispatch Processor
- Stream Processing Units
- Texture Units & Caches
- Memory Read/Write Cache & Stream Out Buffer
- Shader Export
- Render Back-Ends

- 4 texture units
- 8 Texture Address Processors each
  - 4 filtered and 4 unfiltered
- 20 Texture Samplers each
  - Can fetch a single data value per clock
- 4 filtered texels
  - Can bilinear filter one 64-bit FP color value per clock
  - 128b FP per 2 clocks
- Multi-level texture cache design
  - Unified 32k L1 texel cache
  - 32k structure cache (unfiltered)
  - 256KB shared L2 cache
### A quick look back...

<table>
<thead>
<tr>
<th>Year</th>
<th>1998</th>
<th>1999</th>
<th>2000</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistor Size</td>
<td>350 nm</td>
<td>250 nm</td>
<td>180 nm</td>
<td>150 nm</td>
<td>150 nm</td>
<td>130 nm</td>
<td>90 nm</td>
<td>90 nm</td>
<td>80 nm</td>
<td></td>
</tr>
<tr>
<td>Transistor Count</td>
<td>5 million</td>
<td>13 million</td>
<td>30 million</td>
<td>60 million</td>
<td>110 million</td>
<td>110 million</td>
<td>160 million</td>
<td>321 million</td>
<td>384 million</td>
<td>700 million</td>
</tr>
</tbody>
</table>

#### Table

<table>
<thead>
<tr>
<th>Clock Speed</th>
<th>75 MHz</th>
<th>100 MHz</th>
<th>183 MHz</th>
<th>275 MHz</th>
<th>325 MHz</th>
<th>412 MHz</th>
<th>550 MHz</th>
<th>625 MHz</th>
<th>650 MHz</th>
<th>740 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering Pipelines / Shader Processors</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>48</td>
<td>64</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/sec)</td>
<td>0.6</td>
<td>1.6</td>
<td>5.9</td>
<td>8.8</td>
<td>19.8</td>
<td>23.4</td>
<td>37.8</td>
<td>44.8</td>
<td>64.0</td>
<td>106.0</td>
</tr>
<tr>
<td>Pixel Shading / Fill Rate (Mpixels/sec)</td>
<td>75</td>
<td>200</td>
<td>366</td>
<td>1100</td>
<td>2600</td>
<td>3300</td>
<td>8800</td>
<td>10000</td>
<td>31200</td>
<td>47360</td>
</tr>
<tr>
<td>Vertex Processing (Mvertices/sec)</td>
<td>4</td>
<td>8</td>
<td>30</td>
<td>69</td>
<td>325</td>
<td>412</td>
<td>825</td>
<td>1250</td>
<td>1300</td>
<td>11840</td>
</tr>
<tr>
<td>System Bus Bandwidth (GB/sec)</td>
<td>0.53</td>
<td>1.06</td>
<td>1.06</td>
<td>1.06</td>
<td>2.11</td>
<td>2.11</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Eric Demer’s slides available: [http://graphics.stanford.edu/cs448-07-spring/](http://graphics.stanford.edu/cs448-07-spring/)
Close to the Metal (CTM)
Close to the Metal

• 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 CTM SDK
Review: CTM API Goals

• Expose relevant parts of the GPU as they really are
  - Command Processor
  - Data Parallel Processor(s)
  - 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
Evolution of CTM API

• 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
CTM SDK

Stream Applications

Compilers
Stream Extensions for C, C++

Libraries
ACML (Math Library)
AMD Video transcoder library

Eco System
3rd Party Developers
Havok FX™ Peakstream™ Rapidmind™

AMD Runtime
AMD Compute Abstraction Layer (CAL)

CTM HAL
AMD Multicore CPUs
AMD Stream Processors

GPGPU
AMD

Graphics API
Direct X OpenGL
CAL Graphics Binding
ACML SGEMM (includes copy in/out)

ACML SGEMM Comparison - Stream(580) vs CPU(4800+)

- R600: 64,009.5 Mflops
- 150 Gflops (GPU Only)
Software-Hardware Interface

• Developer Ecosystem
  - Libraries (ACML, HavokFX, etc.)
  - Tools / dev env. (RapidMind, Peakstream, etc.)

• Compiled high level languages
  - AMD will provide various implementations
  - Developers free to create their own

• Device independent / portable assembly
  - Assembly spec provided

• Device specific ISA
  - Via device specific extensions to CAL and/or HAL
  - ISA spec provided
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 API Highlights

• Similar to original CTM API
• Memory now 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
• Extensions to CAL provide opportunities for device specific optimization
CAL Memory System

CPU Memory (system / remote)

GPU Memory (local)

GPU Memory (local)

GPU Memory (local)

GPU Memory (local)
CAL Memory System

CPU Memory
(system / remote)

CPU

GPU

GPU Memory (local)

GPU

GPU Memory (local)

CPU Memory (system / remote)
• Allows work to be done when transferring data to from GPU
  - Run shader code, instead of copying
  - Latency can be hidden for some shaders
  - Especially beneficial on output

• Image processing example:
  - 1.8x speedup by eliminating explicit copy
  - Caveat: Amount of improvement depends on chipset!
  - More details during “performance” section
CAL example - overview

1) Initialization
2) Load program module(s)
3) Allocate memory
4) Assign input values
5) Set inputs
6) Execute program(s)
7) Cleanup and exit
CAL example - initialization

```c
static int gDevice = 0;

int main(int argc, char** argv)
{
    CALresult res = CAL_RESULT_OK;

    // open a cal device and create a context
    res = calInit();

    CALuint numDevices = 0;
    res = calDeviceGetCount(&numDevices);
    CHECK_ERROR(res, "There was an error enumerating devices.\n");

    CALdeviceinfo info;
    res = calDeviceGetInfo(&info, 0);
    CHECK_ERROR(res, "There was an error getting device info.\n");

    CALdevice device = 0;
    res = calDeviceOpen(&device, 0);
    CHECK_ERROR(res, "There was an error opening the device.\n");

    CALcontext ctx = 0;
    res = calCtxCreate(&ctx, device);
    CHECK_ERROR(res, "There was an error creating the context.\n");
}
```
CAL example - load modules

// load module
CALmodule module;
res = calModuleLoadFile( &module, ctx, filename );
CHECK_ERROR( res, "There was an error loading the program module.\n" );

NOTE: Modules can be created “online” via CAL compiler interface plugins, or “offline” via external tools (compilers, etc)

Load pre-compiled module from file
### CAL example - memory allocation

```c
// allocate input and output resources and map them into the context
CALresource constRes;
res = calResAllocRemote1D( &constRes, &device, 1, 16, CAL_FORMAT_FLOAT4, 0, 0 );
CHECK_ERROR( res, "There was an error allocating the constant resource.\n" );

CALmem constMem;
res = calCtxGetMem( &constMem, ctx, constRes );
CHECK_ERROR( res, "There was an error getting memory from the constant resource.\n" );

CALresource outputRes;
res = calResAllocRemote2D( &outputRes, &device, 1, BufferWidth, BufferHeight,
CAL_FORMAT_FLOAT4, 0, 0 );
CHECK_ERROR(res, "There was an error allocating the output resource.\n" );

CALmem outputMem;
res = calCtxGetMem( &outputMem, ctx, outputRes );
CHECK_ERROR(res, "There was an error getting memory from the output resource.\n" );
```

**Allocate system (CPU) resource for output buffer**

**Get handle to actual memory**

**Allocate system (CPU) resource for constants**
CAL example - set input values

// clear the resources to known values
float* fdata;
int* idata;
CALuint pitch;

// set constant values
res = calMemMap( (CALvoid**)&idata, &pitch, ctx, constMem, 0);
idata[0] = InputValue;
idata[1] = InputValue;
idata[2] = InputValue;
idata[3] = InputValue;
res = calMemUnmap( ctx, constMem);
res = calMemMap( (CALvoid**)&fdata, &pitch, ctx, outputMem, 0);
for (int i = 0; i < BufferHeight; i++)
{
    float* tmp = &fdata[i * pitch * 4];
    for (int j = 0; j < 4 * BufferWidth; j++)
    {
        tmp[j] = OutputValue;
    }
}
res = calMemUnmap(ctx, outputMem);
// setup the program's inputs and outputs
CALname constName;
res = calModuleGetName( &constName, ctx, module, "cb0" );
CHECK_ERROR( res, "There was an error finding the constant buffer.\n" );

res = calCtxSetMem( ctx, constName, constMem );
CHECK_ERROR( res, "There was an error setting the constant buffer memory.\n" );

CALname outName;
res = calModuleGetName( &outName, ctx, module, "o0" );
CHECK_ERROR( res, "There was an error finding the program output.\n" );

res = calCtxSetMem( ctx, outName, outputMem );
CHECK_ERROR( res, "There was an error setting the program output.\n" );

Set the memory to the appropriate symbol

Get the name (location) of the symbol in the module
Can submit multiple programs and what for a specific one to finish or do work on the CPU while waiting
CAL example - cleanup & exit

// cleanup and exit
    calCtxSetMem(ctx, constName, 0);
    calCtxSetMem(ctx, outName, 0);

    calModuleUnload(ctx, module);

    calCtxReleaseMem(ctx, constMem);
    calResFree(constRes);

    calCtxReleaseMem(ctx, outputMem);
    calResFree(outputRes);

    calCtxDestroy(ctx);
    calDeviceClose(device);
Aside: Predicting Performance

- It is very useful to predict theoretical performance when working on shaders.
- Quite easy with CTM since you can get the ISA even if you use a high-level language.
- Spreadsheets are quite useful for this:
  - Compute theoretical performance.
  - Compute pixels per clock, etc.
  - Easy to see how close an implementation is to peak performance.
- More in “Performance” talk in afternoon.
GPUShaderAnalyzer

HLSL/GLSL shader code window

GPU ISA Disassembly Window

Performance estimates for different AMD GPUs

http://ati.amd.com/developer
Predicting Performance Preview

• ALU bound:

\[
\frac{(#\text{pixels}) \times (#\text{alu instructions})}{(\text{alu/clk}) \times (3\text{D engine speed})}
\]

• MEMORY bound:

\[
\frac{(#\text{pixels}) \times (\text{input + output bits per pixel})}{(\text{bus width}) \times (\text{memory speed})}
\]
Conclusion and Questions

• AMD accelerated computing software stack
  - CTM SDK

• Developer free to use programming interface of preference
  - Device specific ISA
  - Device independent intermediate assembly language
  - High-level language(s)
  - Ecosystem of libraries & 3rd party tools

• Information contact:
  streamcomputing@amd.com