Close to the Metal

Justin Hensley

Graphics Product Group

GPGPU

Overview

SIGGRAPH2007

- Sidebar: Quick R600 architecture overview
- Close to the Metal
  - The SDK vs API
- CTM API v1
- CTM SDK : Compute Abstraction Layer
- Conclusion and Questions

SIGGRAPH2007

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

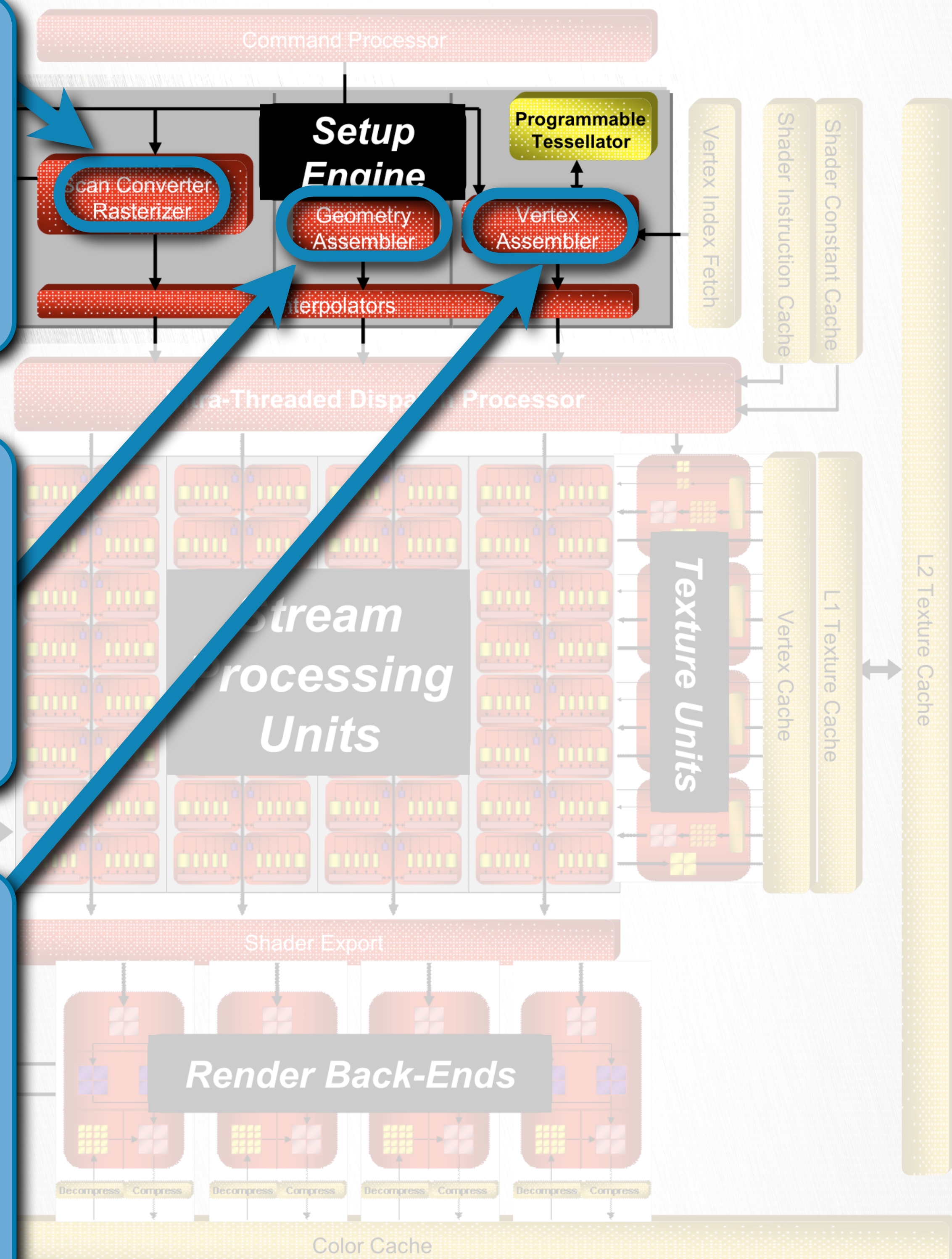
- First "layer" of GPU in design
- Handles command stream fetching, state management, including register subsystems
- Functional highlights:
  - Controls state and decodes command streams
  - Full memory client with Read/Write access
  - Multiple command queues
  - Multiple register interfaces, including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

- Render Back-Ends

Command Processor
Setup Engine
Programmable Tessellator
Scan Converter / Rasterizer
Geometry Assembler
Vertex Assembler
Interpolators
Ultra-Threaded Dispatch Processor
Stream Processing Units
Texture Units
Shader Export
Render Back-Ends
Color Cache
Vertex Index Fetch
Shader Instruction Cache
Shader Constant Cache
Vertex Cache
L1 Texture Cache
L2 Texture Cache

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 shadercore
  - 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



GPGPU AMD

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

GPGPU AMD

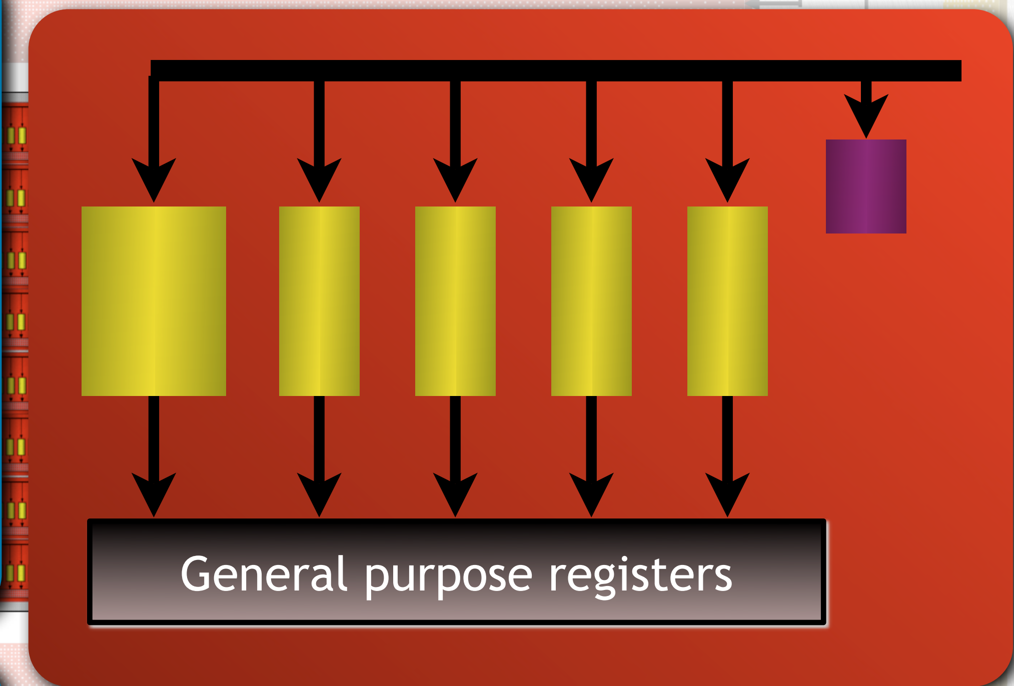
Stream Processing Units

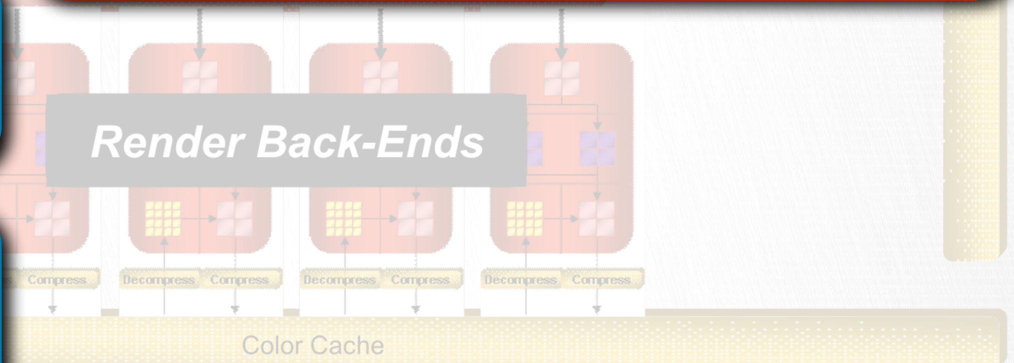
- Command Processor
- Setup Engine
- Ultra-Threaded Dispatch Processor
- Stream Processing Units
- Texture Units & Caches
- Memory Read/Write Cache & Stream Out Buffer
- Shader Export
- Render Back-Ends

- 4 SIMD Arrays of 16 SPUs

Stream Processing Units

- **Issuing**
  - Each SPU works on 4 separate elements, issuing 5 scalar instructions over 4 cycles
  - BEU is issued to separately
- **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
- **Branch execution units handle flow control**
  - NOTE: Predication supported directly in ALU
- **General Purpose Registers**
  - Multi-ported but shared among the processors





Texture Cache

- 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...

	Rage Pro	Rage 128	Radeon	Radeon 8500	Radeon 9700 Pro	Radeon 9800 XT	Radeon X850 XT Platinum Edition	Radeon X1800 XT	Radeon X1950 XTX	Radeon HD 2900 XT
Year	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007
Transistor Size	350 nm	250 nm	180 nm	150 nm	150 nm	150 nm	130 nm	90 nm	90nm	80nm
Transistor Count	5 million	13 million	30 million	60 million	110 million	110 million	160 million	321 million	384 million	700 million
Clock Speed	75 MHz	100 MHz	183 MHz	275 MHz	325 MHz	412 MHz	550 MHz	625 MHz	650 MHz	740 MHz
Rendering Pipelines / Shader Processors	1	2	2	4	8	8	16	16	48	64
Memory Bandwidth (GB/sec)	0.6	1.6	5.9	8.8	19.8	23.4	37.8	44.8	64.0	106.0
Pixel Shading / Fill Rate (Mpixels/sec)	75	200	366	1100	2600	3300	8800	10000	31200	47360
Vertex Processing (Mvertices/sec)	4	8	30	69	325	412	825	1250	1300	11840
System Bus Bandwidth (GB/sec)	0.53	1.06	1.06	1.06	2.11	2.11	4	4	4	4
DirectX Version				DirectX	DirectX	DirectX	DirectX	DirectX	DirectX	DirectX

- 1.7x / year
- 1.3x / year
- 1.6x / year
- 1.8x / year
- 2.0x / year
- 2.4x / year
- 1.3x / year

Eric Demer's slides available:
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++

gcc

Microsoft™

Libraries

ACML (Math Library)

AMD Video transcoder library

Eco System

3rd Party Developers

Havok FX™
Peakstream™
Rapidmind™

Graphics API
Direct X
OpenGL

CAL Graphics Binding

AMD Runtime

AMD Compute Abstraction Layer (CAL)

AMD Multicore CPUs

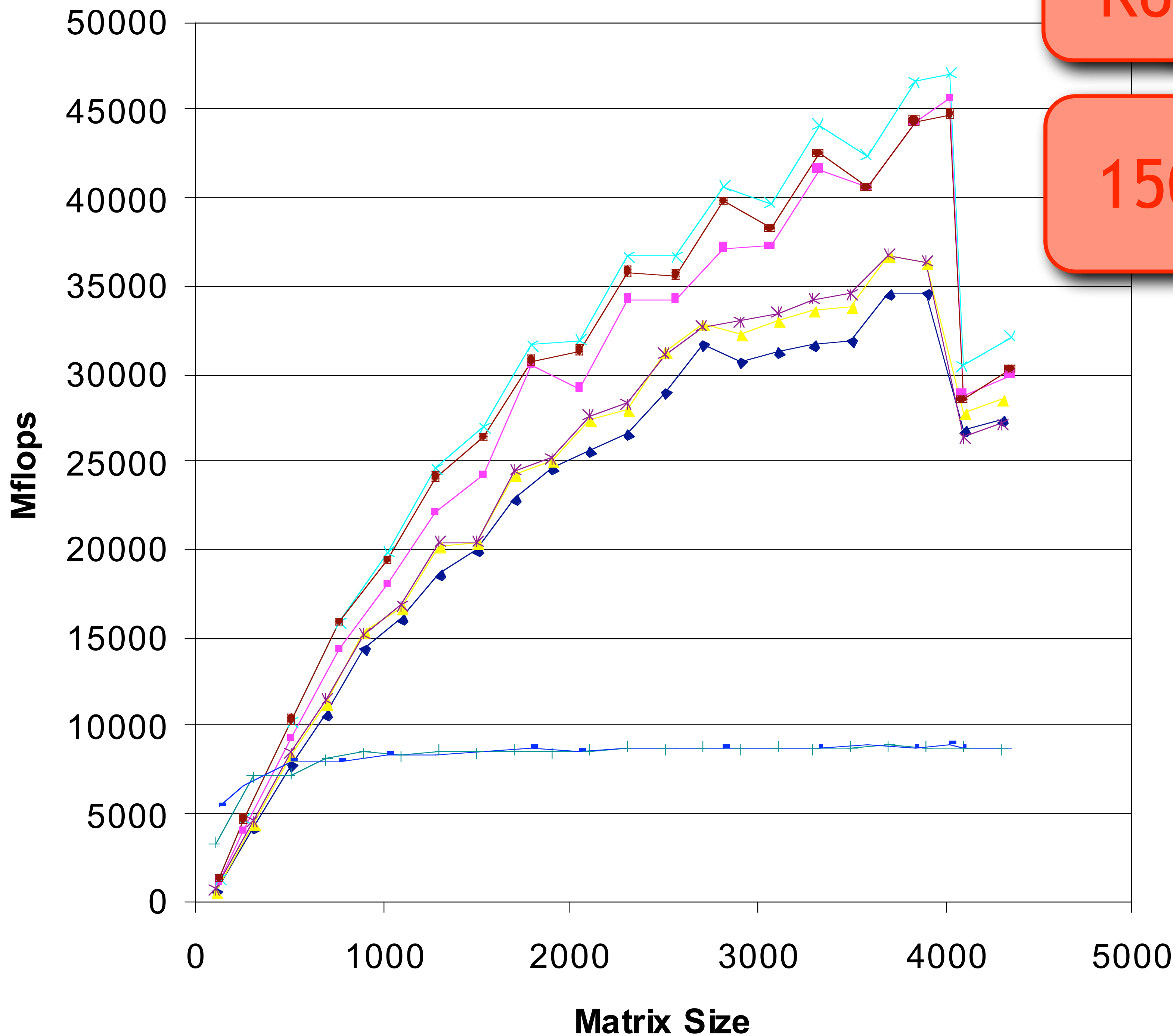
CTM HAL

AMD Stream Processors

GPGPU AMD

ACML SGEMM (includes copy in/out)

SIGGRAPH2007



R600: 64,009.5 Mflops

150 Gflops (GPU Only)

GPGPU AMD

Software-Hardware Interface

SIGGRAPH2007

- **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

SIGGRAPH2007

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

Example Application 2

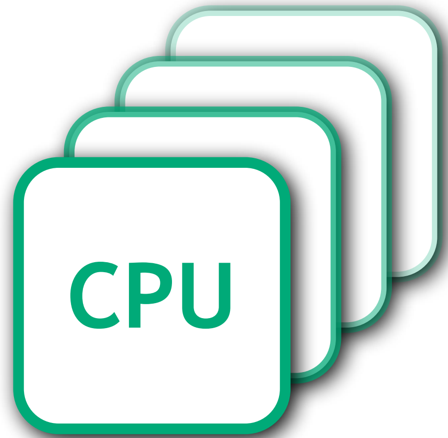
- Real-time depth extraction + physics
- More details given in sketch Thursday afternoon

Advanced
Human Computer Interface
Demo

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

GPU Memory
(local)

GPU

GPU Memory
(local)

GPU

GPU Memory
(local)

GPGPU AMD

CAL Memory System

GPU Access to System Memory

- **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

```
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(r, "There was an error enumerating devices.\n");

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

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

    CALcontext ctx = 0;
    res = calCtxCreate( &ctx, device );
    CHECK_ERROR(r, "There was an error creatint the context.\n");
```

Initialize CAL

Get the number of devices

Get device 0 info

Open device 0

Create a device context

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" );
```

Load pre-compiled module from file

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

CAL example - memory allocation

```
//  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);
```

Get a pointer

Set constants

Unmap when done

Get a pointer

Set memory to known value

Unmap when done

CAL example - set inputs

```
//  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

CAL example - run compute kernel

```
//  get the program entry point
CALfunc func;
res = calModuleGetEntry( &func, ctx, module, "main" );
CHECK_ERROR( res, "There was an error finding the program entry point.\n" );

//  set computational domain
 CALdomain rect;
rect.x = 0;
rect.y = 0;
rect.width  = BufferWidth;
rect.height = BufferHeight;

//  run the program, wait for completion
CALevent event;
res = calCtxRunProgram( &event, ctx, func, &rect );
CHECK_ERROR(r, "There was an error running the program.\n");

//  wait for function to finish
while (calCtxIsEventDone(ctx, event) == CAL_RESULT_PENDING);
```

Get the entry point that we care about from module

Set compute domain

Run our program

Wait for it to finish

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 );
```

Unload module

Release memory

Release context & device

Aside: Predicting Performance

- It is <u>very</u> 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

GPU ShaderAnalyzer - A pAction.hlsl

File Edit Help

Source Code

Function: GeneralFilter3x3

```
277 OneOutput_PS GeneralFi ter3x3( float2 vpos: VPOS )
278 {
279    float2 pos0 = (vp     0.5) / gImageSize;
280    float2 dx = float2(   0, 0.0) / gImageSize;
281    float2 dy = float2(0.0, 1.0) / gImageSize;
282
283    float sum = dot( gKernel0.rgb, 1.0 );
284    float4 texel = gKernel0.r * tex2D( gInput, pos0 - dx
285    texel += gKernel0.g * tex2D( gInput, pos0 - dy );
286    texel += gKernel0.b * tex2D( gInput, pos0 + dx - dy
287
288    sum += dot( gKernel1.rgb, 1.0 );
289    texel += gKernel1.r * tex2D( gInput, pos0 - dx );
290    texel += gKernel1.g * tex2D( gInput, pos0 );
291    texel += gKernel1.b * tex2D( gInput, pos0 + dx );
292
293    sum += dot( gKernel2.rgb, 1.0 );
294    texel += gKernel2.r * tex2D( gInput, pos0 - dx + dy
295    texel += gKernel2.g * tex2D( gInput, pos0 + dy );
296    texel += gKernel2.b * tex2D( gInput, pos0 + dx + dy
297
```

HLSL Compiler

Compile | Options

Target: ps_3_0

- Avoid Flow Control
- Prefer Flow Control
- Skip Optimization
- Use DX9 Semantics

Macro Definitions

Symbol	Value

Right-click to add macros.

Object Code

Format: Radeon HD 2900 (R600) Assembly

```
; -------------- PS Disassembly ------------- -------
00 ALU: ADDR(32) CNT(30)
      0  x: ADD   R123.x,  R0.y, -0.5
         y: MOV   R127.y,  0.0f
         z: ADD   R123.z,  R0.x, -0.5
         w: MOV   R127.w,  0.0f
         t: RECIP_IEEE  R127.z,  C0.x
      1  x: MOV   R123.x,  PS(0).x
         y: ADD   R126.y,  PV(0).z,  0.5
         z: MULADD  R125.z,  C2.y,  1.0f,  C2.z
         w: ADD   R126.w,  PV(0).x,  0.5
         t: RECIP_IEEE  R3.x,  C0.y
      2  x: MOV   R127.x,  PS(1).x
         y: MULADD  R7.y,  PV(1).w,  PS(1).x, -R127.w
         z: MULADD  R7.z,  PV(1).y,  R127.z, -PV(1).x
         w: MUL   R2.w,  PV(1).w,  PS(1).x
         t: MULADD  R0.z,  PV(1).y,  R127.z, -R127.y
      3  x: ADD   R1.x, -R3.x,  PV(2).y
         y: ADD   R2.y, -R3.x,  PV(2).w
         z: MULADD  R1.z,  R127.z,  0.0f,  PV(2).z
         w: MULADD  R0.w,  R126.w,  R3.x, -PV(2).x
```

Compiler Statistics (Using Catalyst 7.6)

Name	GPR	Min	Max	Avg	Est Cycles(Bi)	ALU:TEX(Bi)	Est Cycles(Tri)	ALU:TEX(Tri)	Est Cycles(Aniso)	ALU:TEX(Aniso)	BottleNeck(Bi)	BottleNeck(Tri)	BottleNeck(Aniso)	Throughput(Bi)	Throughput(Tri)	Throughput(Aniso)
Radeon 9700	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Radeon x800	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Radeon x850	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Radeon x1800	15	26.00	26.00	26.00	26.00	2.89	26.00	2.41	26.00	2.06	ALU	ALU	ALU	385 MPixels	385 MPixels	385 MPixels
Radeon x1900	15	9.33	12.60	10.20	9.33	1.04	10.80	0.86	12.60	0.74	ALU	TEX	TEX	1071 MPixels	926 MPixels	794 MPixels
Radeon HD 2900	13	9.00	12.60	10.20	9.00	0.44	10.80	0.37	12.60	0.32	TEX	TEX	TEX	1319 MPixels	1099 MPixels	942 MPixels
Radeon HD 2400	13	9.00	12.60	10.20	9.00	0.89	10.80	0.74	12.60	0.63	TEX	TEX	TEX	356 MPixels	296 MPixels	254 MPixels
Radeon HD 2600	13	9.00	12.60	10.20	9.00	0.59	10.80	0.49	12.60	0.42	TEX	TEX	TEX	711 MPixels	593 MPixels	508 MPixels

D3D Assembly Statistics

Performance estimates for different AMD GPUs

http://ati.amd.com/developer

Predicting Performance Preview

- **ALU bound:**

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

- **MEMORY bound:**

$$\frac{(\#\,\text{pixels}) \times (\text{input} + \text{output bits per pixel})}{(\text{bus width}) \times (\text{memory speed})}$$

Conclusion and Questions

SIGGRAPH2007

- **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