A Performance-Oriented Data Parallel Virtual Machine for GPUs

Mark Peercy    Mark Segal    Derek Gerstmann

ATI Research, Inc.
“...significant barriers still exist for the developer who wishes to use the inexpensive power of commodity graphics hardware, whether for in-game simulation of physics or for conventional computational science. These chips are designed for and driven by video game development; the programming model is unusual, the programming environment is tightly constrained, and the underlying architectures are largely secret. The GPU developer must be an expert in computer graphics and its computational idioms to make effective use of the hardware, and still pitfalls abound...”

• Course Description, SIGGRAPH 2005 GPGPU Course
GPU as Compute Device

Interest for using GPU for compute

- Physical Simulations
- Linear Algebra
- Convolution & FFT
- Sorting & Searching
- Final Frame Rendering
- Cutting Edge Real-Time Graphics

These applications exercise a small fraction of features available in graphics hardware...
Current GPU Abstraction

Rendering Pipeline (OpenGL + Direct3D)

- Great for (existing) real-time graphics and games
- Cumbersome for other types of computation
  - Graphics-centric programming model
  - Forced to manage graphics state
- Implemented through graphics driver
  - Mechanism designed to hide hardware
  - Imposes critical policy decisions
    - How / when / where data resides
    - Updates + optimizations driven by games...
A Data Parallel Approach

The Data Parallel Virtual Machine (DPVM)

- Expose relevant parts of the GPU as they *really* are
  - Command Processor
  - Data Parallel Processors
  - 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
The Data Parallel VM

dpvm.dll
- openManagedConnection()
- closeManagedConnection()
- submitCommandBuffer()
- commandBufferConsumed()

App

DPVM
- Command Processor
- Memory Controller
- Data Parallel Processor Array

Host Memory
- Commands
- Instructions
- Constants
- Inputs
- Outputs

GPU Memory
- Commands
- Instructions
- Constants
- Inputs
- Outputs
Command Processor

- Abstracts communication from architecture
  - Commands are architecturally independent
- Accepts command buffers (CBs) in memory
- Interprets commands in buffer
- Distributes work to processor array
- Application manages command buffers
  - Application fills and submits CBs
  - Application handles synchronization
Command Processor

Complete list of Data Parallel Commands

<table>
<thead>
<tr>
<th>Program Execution</th>
<th>Cache Control</th>
<th>Memory Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_cond_val</td>
<td>inv_inst_cache</td>
<td>set_inst_fmt</td>
</tr>
<tr>
<td>set_domain</td>
<td>inv_constf_cache</td>
<td>set_inp_fmt</td>
</tr>
<tr>
<td>start_program</td>
<td>inv_consti_cache</td>
<td>set_out_fmt</td>
</tr>
<tr>
<td>set_out_mask</td>
<td>inv_constb_cache</td>
<td>set_cond_out_fmt</td>
</tr>
<tr>
<td>set_cond_out_mask</td>
<td>inv_cond_out_cache</td>
<td>set_constf_fmt</td>
</tr>
<tr>
<td>set_cond_test</td>
<td>inv_out_cache</td>
<td>set_consti_fmt</td>
</tr>
<tr>
<td>set_cond_loc</td>
<td>flush_out_cache</td>
<td>set_constb_fmt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_perf_counters</td>
</tr>
<tr>
<td>start_perf_counters</td>
</tr>
<tr>
<td>stop_perf_counters</td>
</tr>
<tr>
<td>read_perf_counters</td>
</tr>
</tbody>
</table>
Data Parallel Processors

- Performs floating-point computations
- Accepts binary executable (ELF)
  - Formal application binary interface (ABI)
  - Uses native instruction set architecture (ISA)
    - ISA is architecturally dependent
    - Only ISA needs to be updated for new architectures (i.e. recompile from high-level language)
- Application submits compiled binary
  - ISA goes straight to the hardware
  - Executable is immune to driver changes
Memory Controller

- Services GPU requests to read/write memory
  - Exports graphics memory directly
    - GPU memory (accessible by GPU only)
    - Host memory (accessible by GPU + CPU)
- Application manages memory resources
  - Specifies locations and formats
  - Can cast between formats w/o copying data
  - Controls data submission + cache invalidation
ATI Close-to-the-Metal (CTM)

Implementation (**ATI x1k DPVM**)

- Radeon x1k architecture (eg x1300 - x1950)
  - Exposes hardware resources (DX9 SM3.0+)
  - Native ISA (ASM text + binary formats)
- Runtime library
  - Low-level driver components
- Support libraries
  - Assembler + Disassembler
  - Command buffer packer
ATI Close-to-the-Metal (CTM)

Processor Resources (Radeon x1k)

- **x16 Inputs (textures)**
  - float1/2/4

- **x4 Outputs (MRT)**...
  - float1/2/4
  - assigned \((x,y)\)
  - or **xINF Outputs**
    - float1
    - arbitrary \((x,y)\)

- **x512 Instructions**
  - any combination...
    - ALU / FLOW CONTROL
    - INPUT / OUTPUT

- **x256 Float Constants**
  - float4

- **x32 Integer Constants**
  - int4

- **x32 Boolean Constants**
  - bool1

- **x128 Registers (GPR)**
  - float4
ATI Close-to-the-Metal (CTM)

Processor Resources (Radeon x1k)

- **x16 Inputs (textures)**
  - float1/2/4
- **x4 Outputs**
  - float1/2/4
  - assigned \((x,y)\)
  - \text{... or xINF Outputs}
  - float1
  - arbitrary \((x,y)\)
- **x512 Instructions**
  - any combination...
  - ALU / FLOW CONTROL
  - INPUT / OUTPUT
- **x256 Float Constants**
  - float4
- **x32 Integer Constants**
  - int4
- **x32 Boolean Constants**
  - bool1
- **x128 Registers (GPR)**
  - float4

SCATTER!
ATI Close-to-the-Metal (CTM)

Additional Features (beyond SM3.0)

- **Scatter** (output float1 values to arbitrary locations)
- **Read + Modify + Write** in a single program
- Fast tiled memory formats
  - **Fetch4** (retrieve x4 float1 in a single clock)
- ABI w/native ISA allows hand-tuned optimizations
- Ability to read/write **directly** to/from host memory
- Avoid non-IEEE floating-point optimizations
- Application dictates granularity of CB submission
  - Save binary CB offline and load at runtime
CTM Usage Example

Open a Connection and Allocate Resources

```c
// Open a connection to the CTM device
ManagedDeviceInfo DevInfo;
VM = OpenManagedConnection( "/dev/pcie0", &DevInfo );

// Allocate command buffer, program, constants, inputs and outputs in host memory
CBufAddressGPU = DevInfo.baseAddressSYS + 0 * 1024 * 1024;
CBufAddressCPU = DevInfo.baseAddressCPU + 0 * 1024 * 1024; // + 1MB (1 MB TOTAL)
ProgramAddressGPU = DevInfo.baseAddressSYS + 1 * 1024 * 1024;
ProgramAddressCPU = DevInfo.baseAddressCPU + 1 * 1024 * 1024; // + 1MB (2 MB TOTAL)
FloatConstAddressGPU = DevInfo.baseAddressSYS + 2 * 1024 * 1024;
FloatConstAddressCPU = DevInfo.baseAddressCPU + 2 * 1024 * 1024; // + 1MB (3 MB TOTAL)
IntConstAddressGPU = DevInfo.baseAddressSYS + 3 * 1024 * 1024;
IntConstAddressCPU = DevInfo.baseAddressCPU + 3 * 1024 * 1024; // + 1MB (4 MB TOTAL)
InputAddressGPU = DevInfo.baseAddressSYS + 4 * 1024 * 1024;
InputAddressCPU = DevInfo.baseAddressCPU + 4 * 1024 * 1024; // + 1MB (5 MB TOTAL)
OutputAddressGPU = DevInfo.baseAddressSYS + 5 * 1024 * 1024;
OutputAddressCPU = DevInfo.baseAddressCPU + 5 * 1024 * 1024; // + 1MB (6 MB TOTAL)
```
Fill Memory Buffers with Application Data

```c
// ... continued ...

// Load a compiled binary program from a file
LoadElfBinaryFromFileName("MyProgram.elf", &ProgramSize, &ProgramBinary);

// Copy binary program into host memory
memcpy( ProgramAddressCPU, ProgramBinary, ProgramSize);

// Copy constant data into host memory
memcpy( FloatConstAddressCPU, FloatConstantData, 256 * 4 * sizeof(float) );
memcpy( IntConstAddressCPU, IntConstantData, 32 * 4 * sizeof(unsigned int) );

// Copy input data into host memory
memcpy( InputAddressCPU, InputData, 1 * 1024 * 1024 );
```
Create a Command Buffer and Populate It

```
// Create a command buffer in host memory and fill it with commands
CB = new CommandBuffer( CBufAddressCPU, 1 * 1024 * 1024 );
CB << SetIntegerConstantsFormatCommand( IntConstAddressGPU, 0, 0, 0 ); // defaults (32x4)
CB << SetFloatConstantsFormatCommand( FloatConstAddressGPU, 0, 0, 0 ); // defaults (256x4)
CB << SetInputFormatCommand( 0, InputAddressGPU, FLOAT4, 0, InputWidth, InputHeight );
CB << SetOutputFormatCommand( 0, OutputAddressGPU, FLOAT4, 0, OutputWidth, OutputHeight );
CB << SetInstructionFormatCommand( ProgramAddressGPU, 0, 0, 0 );
CB << InvalidateIntegerConstantsCacheCommand();
CB << InvalidateFloatConstantsCacheCommand();
CB << InvalidateInstructionCacheCommand();
CB << FlushOutputCacheCommand();
CB << SetDomainCommand( 0, 0, OutputWidth - 1, OutputHeight - 1 );
CB << StartProgramCommand();
```
/ ... continued ...

// Command buffer has been packed in memory, now submit it to CTM
SubmitId = SubmitCommandBuffer( VM, CBufAddressGPU, 1 * 1024 * 1024 );

// Wait until command buffer is completely consumed
while ( CommandBufferConsumed( VM, SubmitId ) == 0 ) { /* spin and wait */ }

// Output values have now been written to host memory, process results
ProcessResults( OutputAddressCPU, 1 * 1024 * 1024 );

// Close the CTM device
CloseManagedConnection( VM );

// DONE!
CTM HLSL->ISA Example

**HLSL**

```hlsl
uniform float4 scale;
uniform float4 bias;
uniform sampler2D data;

float4 main(float2 index: VPOS) : COLOR
{
    return (tex2D(data, index) * scale + bias); // output
}
```

**PS3**

```ps3
// OP | DST | SRC0 | SRC1 | SRC2
ps_3_0

dcl vPos.xy

dcl_2d s0 // s0 = data

mov r2.xy, vPos // r2 = index

texld r0, r2, s0 // r0 = tex2d(data, index)
mov r1, c0 // r1 = scale

mad oC0, r0, r1, c1 // output = r0 * r1 + bias
```

**ISA**

```isa
// (mod) | OP | DST | SRC0 | SRC1 | SRC2 | SRC3
main:
I000:  /p/i/v TEXT r0 r0.rgrr s0 // r0 = tex2d(data, index)
I001:  /p MAD r0.xxx o0 c0 r0 c1 // output.rgb = r0 * scale + bias
       mad r0.x o0 r0 c0 c1 // output.a = r0 * scale + bias
       END

HALT
```
CTM Example Applications

Runtime comparison (*Graphics API vs CTM*)

<table>
<thead>
<tr>
<th>App</th>
<th>Benefit</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix-Matrix Multiply</td>
<td>x10</td>
<td>CB, ISA, mem-formats, mem-offsets, interleaving, fetch4</td>
</tr>
<tr>
<td>FFT</td>
<td>x2</td>
<td>CB, ISA, interleaving</td>
</tr>
<tr>
<td>GPURay</td>
<td>x2</td>
<td>CB, mem-formats</td>
</tr>
<tr>
<td>QJulia</td>
<td>x2</td>
<td>CB, mem-formats</td>
</tr>
</tbody>
</table>

*Measured on a single Radeon x1900*
Conclusion

Benefits of the Data Parallel Approach

- Straight-forward programming model
  - Allows hand-tuned optimizations
- Exposes actual hardware device
  - Direct control over memory + processors
  - Application binary interface + native ISA
- Application is responsible for all *policy* decisions
- Allows *consistent* performance for compute
Future Work

Other things to explore...

- Open area for tool development
  - Low-level profilers + debuggers
- New opportunities for compiler research
  - ISA provides new target for code generation
  - Support for new high-level languages
  - Non-graphics based optimizations
  - Resource management for data parallel apps
- Extensions to expose more graphics functionality
Special Thanks...

ATI Research, Inc.

- Mark Peercy, Mark Segal, Alex Chalflin, Alpana Kaulgud, Raja Kodori, and everyone else...

Stanford University

- Mike Houston, Daniel Horn

Graphics Hardware Workshop

- Hot3D Program Chairs
QUESTIONS?

For more information contact:

researcher@ati.com