CAL Kernel Programming
Module Overview

• CAL Kernel Programming APIs
• Overview of AMD IL
• Overview of AMD GPU ISA
CAL Kernel Programming APIs

- AMD Brook+
- DirectX High Level Shading Language (HLSL)
- AMD Intermediate Language (IL)
- AMD GPU-specific Assembly Instruction Set (ISA)
AMD Brook+

- ‘C’ with Streaming Extensions
- Includes
  - Kernel programming interface
  - Runtime implementation
    - Transparent to application programmer
    - Supports multiple runtimes
  - Runtime API
    - Simple and small API for basic operations
    - Provides only high-level control over stream data
DirectX High Level Shading Language (HLSL)

- ‘C’ for GPU Programming for 3D applications
- Includes
  - Kernel Programming Interface
  - Runtime provided by associated runtime library (DirectX)
  - Front-end compiler provided by Microsoft (\texttt{fxc})
  - Back-end code generation and optimization done by GPU vendors
- Original BrookGPU included as DirectX runtime
  - Generate HLSL for Brook kernels
  - Invoke DirectX runtime calls for stream management and kernel invocation, etc
AMD Intermediate Language

- Psuedo-assembly interface
- Interface is *Architecture Independent*
- Includes graphics-ish commands as well
- Evolves with GPU evolution
- Specifications available with the SDK installation
AMD GPU ISA

• True GPU assembly
• Interface and Implementation is Architecture Dependent
  – Exposes the GPU architecture completely
  – Provides insight into GPU architecture and expected performance
• No optimizations done by the assembler on specified ISA
• Not expected to use directly for application development
  – Very useful for performance profiling
  – Useful for debugging programs
AMD IL Interface

• An IL program consists of
  – Versioning information, declarations, etc
  – Registers (operands, temporaries and results)
  – Instructions

• IL registers are 4-component vectors

• IL instructions that accept vectors by default perform vector instructions
  – Current and future GPUs have superscalar units so scalar instructions are encouraged
    ⇒ Use masking operators when vector instructions are not needed
    ⇒ The compiler will optimize the generated code appropriately
AMD IL – Data Types

• IL is a type-less language
  – Registers do not have specific types
  – Can be used to store 32-bit integers, floats, and 64-bit double precision values
  – Types are defined by the instructions, e.g.
    IMUL, UMUL, MUL and DMUL
    correspond to multiplication of
    signed integers, unsigned integers, floats and doubles respectively
**AMD IL – Instruction Syntax**

\(<\text{instr}>[_<\text{ctrl}>][_<\text{ctrl}\text{(val)}>][<\text{dst}>[_<\text{mod}>][.<\text{write-mask}>]][[, <\text{src}>[_<\text{mod}>][.<\text{swizzle-mask}>]]... \)

Items within <> are replaced by specific words or phrases. For example, \(<\text{instr}> \) could be replaced by \text{mov}.

When more than one item of the same type exists, for example \(<\text{src}>\), a zero-based number is appended to the item name. For example, for 2-input instructions the sources (inputs) would be labeled \text{src0} and \text{src1}.

Items within square brackets, [], are optional.

Items within braces, {}, represent a choice-grouping. For example, \{x|y|z|w\} means to choose \(x, y, z\) or \(w\).

An ellipsis, ..., represents a list of alternatives having the same format as the first alternative in the list.
**Important ALU Instructions**

**ADD** – Addition of two registers

```plaintext
add dst, src0, src1

dst = src0 + src1
```

**MUL** – Multiplication of two registers

```plaintext
mul dst, src0, src1

dst = src0 * src1
```

**MAD** – Multiplication of two registers and addition with third
- More efficient as it performs 2 vector instructions in 1 cycle

```plaintext
mad[_ieee] dst, src0, src1, src2

dst = src0 * src1 + src2
```
Important ALU Instructions

**DIV** – Division of two registers

```
div_zeroop(op) dst, src0, src1
dst = src0 / src1
zeroop() decides value of output if divisor is zero, e.g.
fltmax, zero, infinity, inf Else_max
```

**RCP** – Reciprocal

```
rcp_zeroop(op) dst, src0
dst = 1/src0.w
```

**SQRT** – Square Root

```
sqrt dst, src0
dst = sqrt(src0.w)
```
## IL Registers

IL provides registers for various tasks

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Need to declare</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>r#</td>
<td>Temporary</td>
<td>No</td>
<td>mad r0, r0, r0, r0</td>
</tr>
<tr>
<td>cb[#]</td>
<td>Const buffer</td>
<td>Yes</td>
<td>mov o0, cb0[0]</td>
</tr>
<tr>
<td>v0</td>
<td>Position</td>
<td>Yes</td>
<td>mul o0, v0.xyxx, cb0[0]</td>
</tr>
<tr>
<td>o#</td>
<td>Output</td>
<td>Yes</td>
<td>mov o0, v0.xyxx</td>
</tr>
</tbody>
</table>
Reading Inputs - Samplers

• Handle used to “Sample” or read from input buffers (textures in graphics)
  – specify the buffer from which to read
  – the address within this buffer – 2-tuple for 2D buffers
  – use sampling operator

C

    // Buffer A, Address (x, y), Operator [[][]]
    float temp = A[x][y];

Brook+

    // Buffer A, Address vector reg.xy, Operator []
    float templ = A[reg.xy];

IL

    // Buffer resource 0, Address vector v0.xy, Operator
    // sample_resource(#)_sampler(#)
    sample_resource(0)_sampler(0) r0, v0.xy
Reading Inputs - Samplers

- Both the sample operator and output register are type-less
  - The data format and sampling mode needs to be specified in the kernel for each sampler

  e.g.

  dcl_resource_id(0)_type(2d, unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)

  Type of resource (1d vs 2d) specified
  Type of address (normalized vs unnormalized) specified
  - Normalized uses coordinates in [0..1] range
  - Unnormalized uses coordinates in [0..n-1] range
Register Modifiers, Masks and Swizzles

\[ \text{<instr>[_<ctrl>][][_<ctrl(val)>]} \]
\[ \text{[<dst>[_<mod>][.<write-mask>]]} \]
\[ \text{[,] _<src>[_<mod>][.<swizzle-mask>]]...} \]
# Source Modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>_invert</td>
<td>Invert components ((1 - x))</td>
<td>add r0, r1_invert, r2</td>
</tr>
<tr>
<td>_bias</td>
<td>Elements are biased ((x - 0.5))</td>
<td>add r0, r1, r2_bias</td>
</tr>
<tr>
<td>_x2</td>
<td>Multiply elements by 2.0</td>
<td>add r0, r1, r2_x2</td>
</tr>
<tr>
<td>_bx2</td>
<td>Signed scaling. Combined bias and x2 modifiers.</td>
<td>add r0, r1, r2_bx2</td>
</tr>
<tr>
<td>_sign</td>
<td>Signs Elements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Elements less then 0 become -1</td>
<td>mov r0, r1_sign</td>
</tr>
<tr>
<td></td>
<td>Elements equal to 0 become 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Elements greater then 0 become 1</td>
<td></td>
</tr>
<tr>
<td>_divcomp(type)</td>
<td>Performs division based on type. type: y, z, w, unknown</td>
<td>texld_stage(0) r0, vT0_divcomp(y)</td>
</tr>
<tr>
<td>_abs</td>
<td>Takes the absolute value of elements.</td>
<td>mov r0, r1_abs</td>
</tr>
<tr>
<td>_neg(comp)</td>
<td>Provides per element negate</td>
<td>mov r0, r1_neg(comp)</td>
</tr>
</tbody>
</table>
# Destination Modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>_x2</td>
<td>Shift scale modifiers</td>
<td>add_x2 r0, r1, r2</td>
</tr>
<tr>
<td>_x4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_x8</td>
<td>Shift scale modifiers</td>
<td></td>
</tr>
<tr>
<td>_d2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_d4</td>
<td>Saturate or clamp result to [0,1]</td>
<td>add_sat r0, r1, r2, add_x2_sat r0, r1, r2</td>
</tr>
<tr>
<td>_d8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write Masks

- Mask a destination element using specified flag

\[
\text{reg.}\{x|_|0|1}\{y|_|0|1}\{z|_|0|1}\{w|_|0|1}\]

- Letter (x, y, z, w) => element to be written
- Underscore (_\ ) => element \textit{not} to be written
- Zero (0) or One (1) => element forced to zero or one
- Omitting => underscore

E.g.

\[
\text{mov r0.x_zw, r1}
\]
\[
\text{mov r0.x_01, r1}
\]
Swizzle-mask

- Manipulate the default positions of elements in a register to be used as a source/destination

\[ \text{reg.}\{x\,|\,y\,|\,z\,|\,w\,|\,0\,|\,1\}\{y\,|\,z\,|\,w\,|\,x\,|\,0\,|\,1\}\{z\,|\,w\,|\,x\,|\,y\,|\,0\,|\,1\}\{w\,|\,x\,|\,y\,|\,z\,|\,0\,|\,1\}\]

  - Swizzle mask is position dependent
  - Letter \((x, y, z, w)\) => element to be read/written
  - Zero \((0)\) or One \((1)\) => element forced to zero or one
  - Omitting => default value

E.g.

\[
\begin{align*}
\text{mov } r0, r1.yz ; & \text{ same as } \text{mov } r0, r1.yzzw \\
\text{mov } r0, r1.yzxl ; & \text{ force } r0.w \text{ to be } 1 \\
\end{align*}
\]
Comments

Semi-colon used for comment

e.g.

; this is a comment

mov r0, r1 ; this is another comment
Overview of AMD GPU ISA
AMD GPU ISA

• Reviewing assembly is handy for performance profiling and debugging

• CAL provides interfaces to dump dis-assembly from CALimage

  `calclDisassembleImage`

  ```c
  void calclDisassembleImage(const CALimage image, CALLogFunction logfunc);
  ```

  • Disassembles the CAL image

  `calclDisassembleObject`

  ```c
  void calclDisassembleObject(const CALobject* obj, CALLogFunction logfunc);
  ```

  • Disassembles the CAL object
AMD GPU ISA

• Disassembler
  – Both outputs the disassembled ISA on a line-by-line basis
  – Uses application specified log function

```c
typedef void (*CALLogFunction)(const char* msg);
void disasm(const char *msg) {
    fprintf(stderr, "%s\n", msg);
}
calclDisassembleImage(image, disasm);
calclDisassembleObject(object, disasm);
```

• Command line utilities – amudisasm, amuasm
  – Dis-assemble a given binary image
  – Assemble a given text GPU ISA kernel
AMD IL to GPU ISA

INPUT: AMD IL Kernel

OUTPUT: R600 ISA

Performance statistics for different GPUs
AMD GPU ISA - Program Structure

- ISA Program consists of set of Instruction Clauses
  - ALU Clauses
  - TEX Clauses
- ISA Maps directly to the underlying hardware
  - Computational Core consists of 5-way ALU units
  - ALU Clause consists of one or more ALU Instruction Groups
  - ALU Instruction Group consists of up-to 5 ALU instructions
    - Represented by x, y, z, w, t
- TEX Clauses perform memory read operations
  - Texture mapping in 3D graphics
  - Additional Flags are used for compute-specific features
kernel void test(int N,
float A<>,
float B<>,
out float C<>)
{
    float a = A + 5.0f;
    float b = B + 5.0f;
    C = a + b;
}
AMD GPU ISA

• Reveals Information on
  – Actual number of ALU cycles in kernel
  – Data Dependencies
  – ALU utilization

• Does not reveal Information on
  – Total number of cycles for execution
    • Memory access latencies are not accounted for
; --------- PS Disassembly ---------
00 TEX: ADDR(48) CNT(1) VALID_PIX
  0  SAMPLE R0, R0.xy0x, t0, s0
     UNNORM(XYZW)
01 ALU: ADDR(32) CNT(4) KCACHE0(CB0:0-15)
  1  x: MUL  R0.x,  R0.x,  KC0[0].x
       y: MUL  R0.y,  R0.y,  KC0[0].y
       z: MUL  R0.z,  R0.z,  KC0[0].z
       w: MUL  R0.w,  R0.w,  KC0[0].w
02 EXP_DONE: PIX0, R0
END_OF_PROGRAM
Important Tokens

R# - 128-bit GPRs

KCACHE# - GPU Constant Cache

SAMPLE – Memory Read Operation

F_TO_I – Floating point to Integer conversion

LOOP_DX10 – DX10-style for loop
Q&A and Recap

- CAL Kernel Programming Interfaces
  - Brook+
  - HLSL
  - AMD IL
  - AMD GPU ISA