HSAIL: PORTABLE COMPILER IR FOR HSA

HOT CHIPS TUTORIAL - AUGUST 2013

BEN SANDER
AMD SENIOR FELLOW
STATE OF GPU COMPUTING

• GPUs are fast and power efficient: high compute density per-mm and per-watt
• But: Can be hard to program

Today’s Challenges
- Separate address spaces
  - Copies
  - Can’t share pointers
- New language required for compute kernel
  - EX: OpenCL™ runtime API
  - Compute kernel compiled separately than host code

Emerging Solution
- HSA Hardware
  - Single address space
  - Coherent
  - Virtual
  - Fast access from all components
  - Can share pointers
- Bring GPU computing to existing, popular, programming models
  - Single-source, fully supported by compiler
  - HSAIL compiler IR (Cross-platform!)
WHAT IS HSAIL?

- **HSAIL** is the intermediate language for parallel compute in HSA
  - Generated by a high level compiler (LLVM, gcc, Java VM, etc)
  - Low-level IR, close to machine ISA level
  - Compiled down to target ISA by an IHV “Finalizer”
  - Finalizer may execute at run time, install time, or build time

- Example: OpenCL™ Compilation Stack using HSAIL

High-Level Compiler Flow (Developer)

- OpenCL™ Kernel
- EDG or CLANG
- SPIR
- LLVM
- HSAIL

Finalizer Flow (Runtime)

- HSAIL
- Finalizer
- Hardware ISA
KEY HSAIL FEATURES

- Parallel
- Shared virtual memory
- Portable across vendors in HSA Foundation
- Stable across multiple product generations
- Consistent numerical results (IEEE-754 with defined min accuracy)
- Fast, robust, simple finalization step (no monthly updates)
- Good performance (little need to write in ISA)
- Supports all of OpenCL™ and C++ AMP™
- Support Java, C++, and other languages as well
AGENDA

- Introduction
- HSA Parallel Execution Model
- HSAIL Instruction Set
- Example – in Java!
- Key Takeaways
HSA PARALLEL EXECUTION MODEL
Basic Idea:

Programmer supplies a “kernel” that is run on each work-item. Kernel is written as a single thread of execution.

Each work-item has a unique coordinate.

Programmer specifies grid dimensions (for scope of problem).

Programmer optionally specifies work-group dimensions (for optimized communication).
CONVOLUTION / SOBEL EDGE FILTER

\[ G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \]

\[ G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \]

\[ G = \sqrt{G_x^2 + G_y^2} \]
CONVOLUTION / SOBEL EDGE FILTER

\[ G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \]

\[ G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \]

\[ G = \sqrt{G_x^2 + G_y^2} \]
CONVOLUTION / SOBEL EDGE FILTER

\[ G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \]

\[ G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \]

\[ G = \sqrt{G_x^2 + G_y^2} \]
HSAIL INSTRUCTION SET
HSAIL INSTRUCTION SET - OVERVIEW

- Similar to assembly language for a RISC CPU
  - Load-store architecture
    - `ld_global_u64  $d0, [$d6 + 120] ; $d0= load($d6+120)`
    - `add_u64  $d1, $d2, 24 ; $d1= $d2+24`

- 136 opcodes (Java™ bytecode has 200)
  - Floating point (single, double, half (f16))
  - Integer (32-bit, 64-bit)
  - Some packed operations
  - Branches
  - Function calls
    - Platform Atomic Operations: and, or, xor, exch, add, sub, inc, dec, max, min, cas
      - Synchronize host CPU and HSA Component!

- Text and Binary formats (“BRIG”)
7 segments of memory
- global, readonly, group, spill, private, arg, kernarg,
- Memory instructions can (optionally) specify a segment

Global Segment
- Visible to all HSA agents (including host CPU)

Group Segment
- Provides high-performance memory shared in the work-group.
- Group memory can be read and written by any work-item in the work-group
- HSAIL provides sync operations to control visibility of group memory
- Useful for expert programmers

Spill, Private, Arg Segments
- Represent different regions of a per-work-item stack
- Typically generated by compiler, not specified by programmer
- Compiler can use these to convey intent – ie spills
SEGMENTS AND MEMORY (2/2)

- Kernarg Segment
  - Programmer writes kernarg segment to pass arguments to a kernel

- Read-Only Segment
  - Remains constant during execution of kernel

- Flat Addressing
  - Each segment mapped into virtual address space
  - Flat addresses can map to segments based on virtual address
  - Instructions with no explicit segment use flat addressing
  - Very useful for high-level language support (ie classes, libraries)
  - Aligns well with OpenCL 2.0 “generic” addressing feature

```
ld_kernarg_u64  $d6, [%_arg0]
ld_u64  $d0,[$d6+24]  ; flat
```
REGISTERS

- Four classes of registers
  - C: 1-bit, Control Registers
  - S: 32-bit, Single-precision FP or Int
  - D: 64-bit, Double-precision FP or Long Int
  - Q: 128-bit, Packed data.

- Fixed number of registers:
  - 8 C
  - S, D, Q share a single pool of resources
    - $S + 2\times D + 4\times Q \leq 128$
    - Up to 128 S or 64 D or 32 Q (or a blend)

- Register allocation done in high-level compiler
  - Finalizer doesn’t have to perform expensive register allocation
SIMT EXECUTION MODEL

- HSAIL Presents a “SIMT” execution model to the programmer
  - “Single Instruction, Multiple Thread”
  - Programmer writes program for a single thread of execution
  - Each work-item appears to have its own program counter
  - Branch instructions look natural

- Hardware Implementation
  - Most hardware uses SIMD (Single-Instruction Multiple Data) vectors for efficiency
  - Actually one program counter for the entire SIMD instruction
  - Branches implemented with predication

- SIMT Advantages
  - Easier to program (branch code in particular)
  - Natural path for mainstream programming models
  - Scales across a wide variety of hardware (programmer doesn’t see vector width)
  - Cross-lane operations available for those who want peak performance
Hardware SIMD vector, composed of 1, 2, 4, 8, 16, 32, or 64 “lanes”

Lanes in wavefront can be “active” or “inactive”

Inactive lanes consume hardware resources but don’t do useful work

Tradeoffs

“Wavefront-aware” programming can be useful for peak performance
But results in less portable code (since wavefront width is encoded in algorithm)
CROSS-LANE OPERATIONS

- Example HSAIL operation: “countlane”
  - Dest set to the number of work-items in current wavefront that have non-zero source

```hsail
countlane_u32 $s0, $s6
```

- Example HSAIL operation: “countuplane”
  - Dest set to count of earlier work-items that are active for this instruction
  - Useful for compaction algorithms

```hsail
countuplane_u32 $s0
```
HSAIL MODES

◆ Working group strived to limit optional modes and features in HSAIL
  ◆ Minimize differences between HSA target machines
  ◆ Better for compiler vendors and application developers
  ◆ Two modes survived

◆ Machine Models
  ◆ Small: 32-bit pointers, 32-bit data
  ◆ Large: 64-bit pointers, 32-bit or 64-bit data
  ◆ Vendors can support one or both models

◆ “Base” and “Full” Profiles
  ◆ Two sets of requirements for FP accuracy, rounding, exception reporting
<table>
<thead>
<tr>
<th>Feature</th>
<th>Base</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing Modes</td>
<td>Small, Large</td>
<td>Small, Large</td>
</tr>
<tr>
<td>Load/store conversion of all floating point types</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>(f16, f32, f64)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All 32-bit HSAIL operations according to the</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>declared profile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F16 support (IEEE 754 or better)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>F64 support</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Precision for add/sub/mul</td>
<td>1/2 ULP</td>
<td>1/2 ULP</td>
</tr>
<tr>
<td>Precision for div</td>
<td>2.5 ULP</td>
<td>1/2 ULP</td>
</tr>
<tr>
<td>Precision for sqrt</td>
<td>1 ULP</td>
<td>1/2 ULP</td>
</tr>
<tr>
<td>HSAIL Rounding: Near</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HSAIL Rounding: Up</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>HSAIL Rounding: Down</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>HSAIL Rounding: Zero</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Subnormal floating-point</td>
<td>Flush-to-zero</td>
<td>Supported</td>
</tr>
<tr>
<td>Propagate NaN Payloads</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>FMA</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Arithmetic Exception reporting</td>
<td>DETECT</td>
<td>DETECT or BREAK</td>
</tr>
<tr>
<td>Debug trap</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
class Player {
    private Team team;
    private int scores;
    private float pctOfTeamScores;

    public Team getTeam() {return team;}
    public int getScores() {return scores;}
    public void setPctOfTeamScores(int pct) { pctOfTeamScores = pct; }
}

// “Team” class not shown

// Assume “allPlayers’ is an initialized array of Players..
Stream<Player> s = Arrays.stream(allPlayers).parallel();

s.forEach(p -> {
    int teamScores = p.getTeam().getScores();
    float pctOfTeamScores = (float)p.getScores()/(float) teamScores;
    p.setPctOfTeamScores(pctOfTeamScores);
});
HSAIL CODE EXAMPLE

01: version 0.95: $full : $large;
02: // static method HotSpotMethod<Main.lambda$2(Player)>
03: kernel &run ( 
04:  kernarg_u64 %_arg0       // Kernel signature for lambda method
05:  ) { 
06:  ld_kernarg_u64 $d6, [%_arg0]; // Move arg to an HSAIL register
07:  workitemabsid_u32 $s2, 0;   // Read the work-item global “X” coord
08: 
09:  cvt_u64_s32 $d2, $s2;       // Convert X gid to long
10:  mul_u64 $d2, $d2, 8;        // Adjust index for sizeof ref
11:  add_u64 $d2, $d2, 24;      // Adjust for actual elements start
12:  add_u64 $d2, $d2, $d6;     // Add to array ref ptr
13:  ld_global_u64 $d6, [$d2];  // Load from array element into reg
14:  @L0: 
15:  ld_global_u64 $d0, [$d6 + 120]; // p.getTeam() 
16:  mov_b64 $d3, $d0; 
17:  ld_global_s32 $s3, [$d6 + 40]; // p.getScores () 
18:  cvt_f32_s32 $s16, $s3; 
19:  ld_global_s32 $s0, [$d0 + 24]; // Team getScores() 
20:  cvt_f32_s32 $s17, $s0; 
21:  div_f32 $s16, $s16, $s17; // p.getScores() / teamScores 
22:  st_global_f32 $s16, [$d6 + 100]; // p.setPctOfTeamScores ()
23:  ret; 
24:  };
WHAT IS HSAIL?

- HSAIL is the intermediate language for parallel compute in HSA
  - Generated by a high level compiler (LLVM, gcc, Java VM, etc)
  - Low-level IR, close to machine ISA level
  - Compiled down to target ISA by an IHV “Finalizer”
  - Finalizer may execute at run time, install time, or build time

- Example: OpenCL™ Compilation Stack using HSAIL
# HSAIL AND SPIR

<table>
<thead>
<tr>
<th>Feature</th>
<th>HSAIL</th>
<th>SPIR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intended Users</strong></td>
<td>Compiler developers who want to control their own code generation.</td>
<td>Compiler developers who want a fast path to acceleration across a wide variety of devices.</td>
</tr>
<tr>
<td><strong>IR Level</strong></td>
<td>Low-level, just above the machine instruction set</td>
<td>High-level, just below LLVM-IR</td>
</tr>
<tr>
<td><strong>Back-end code generation</strong></td>
<td>Thin, fast, robust.</td>
<td>Flexible. Can include many optimizations and compiler transformation including register allocation.</td>
</tr>
<tr>
<td><strong>Where are compiler optimizations performed?</strong></td>
<td>Most done in high-level compiler, before HSAIL generation.</td>
<td>Most done in back-end code generator, between SPIR and device machine instruction set</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>Fixed-size register pool</td>
<td>Infinite</td>
</tr>
<tr>
<td><strong>SSA Form</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Binary format</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Code generator for LLVM</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Back-end device targets</strong></td>
<td>Modern GPU architectures supported by members of the HSA Foundation</td>
<td>Any OpenCL device including GPUs, CPUs, FPGAs</td>
</tr>
<tr>
<td><strong>Memory Model</strong></td>
<td>Relaxed consistency with acquire/release, barriers, and fine-grained barriers</td>
<td>Flexible. Can support the OpenCL 1.2 Memory Model</td>
</tr>
</tbody>
</table>
TAKEAWAYS

◆ HSAIL Key Points
  ◆ Thin, robust, fast finalizer
  ◆ Portable (multiple HW vendors and parallel architectures)
  ◆ Complements OpenCL™
  ◆ Supports shared virtual memory and platform atomics

◆ HSA brings GPU computing to mainstream programming models
  ◆ Shared and coherent memory bridges “faraway accelerator” gap
  ◆ HSAIL provides the common IL for high-level languages to benefit from parallel computing

◆ Java Example
  ◆ Unmodified Java8 accelerated on the GPU!
  ◆ Can use pointer-containing data structures
TOOLS ARE AVAILABLE NOW

  - [https://hsafoundation.box.com/s/m6mrsjv8b7r50kqeyyal](https://hsafoundation.box.com/s/m6mrsjv8b7r50kqeyyal)

- Tools now at GitHub – HSA Foundation
  - libHSA Assembler and Disassembler
    - [https://github.com/HSAFoundation/HSAIL-Tools](https://github.com/HSAFoundation/HSAIL-Tools)
  - HSAIL Instruction Set Simulator
    - [https://github.com/HSAFoundation/HSAIL-Instruction-Set-Simulator](https://github.com/HSAFoundation/HSAIL-Instruction-Set-Simulator)

- Soon: LLVM Compilation stack which outputs HSAIL and BRIG

- Java compiler for HSAIL (preliminary)
  - [http://openjdk.java.net/projects/sumatra/](http://openjdk.java.net/projects/sumatra/)
  - [http://openjdk.java.net/projects/graal/](http://openjdk.java.net/projects/graal/)
OPPORTUNITIES WITH LLVM BASED COMPILATION

- C99
- C++ 11
- C++AMP
- Objective C
- OpenCL
- OpenMP
- KL
- OSL
- Render script
- UPC
- Rust

CLANG

LLVM
// Vector add
// A[0:N-1] = B[0:N-1] + C[0:N-1]

__kernel void vec_add(
    __global const float *a,
    __global const float *b,
    __global float *c,
    const unsigned int n)
{
    // Get our global thread ID
    int id = get_global_id(0);

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
version 1:0:$full:$small;
function &get_global_id(arg_u32 %ret_val)
(arg_u32 %arg_val0);
function &abort()();
kernel &__OpenCL_vec_add_kernel(
  kernarg_u32 %arg_val0,
  kernarg_u32 %arg_val1,
  kernarg_u32 %arg_val2,
  kernarg_u32 %arg_val3)
{
  __OpenCL_vec_add_kernel_entry:
    // BB#0: // %entry
    ld_kernarg_u32 $s0, [%arg_val3];
    workitemabsid_u32 $s1, 0;
    cmp_lt_b1_u32 $c0, $s1, $s0;
    ld_kernarg_u32 $s0, [%arg_val2];
    ld_kernarg_u32 $s2, [%arg_val1];
    ld_kernarg_u32 $s3, [%arg_val0];
    cbr $c0, @BB0_2;
    brn @BB0_1;
  @BB0_1: // %if.end
    ret;
  @BB0_2: // %if.then
    shl_u32 $s1, $s1, 2;
    add_u32 $s2, $s2, $s1;
    ld_global_f32 $s2, [$s2];
    add_u32 $s3, $s3, $s1;
    ld_global_f32 $s3, [$s3];
    add_f32 $s2, $s3, $s2;
    add_u32 $s0, $s0, $s1;
    st_global_f32 $s2, [$s0];
    brn @BB0_1;
};
SEGMENTS

Grid

Work-Group

Work-Items
(can also access Flat Address Space)

Group Segment

Global Segment

Private Segment

Private Segment

Agent

Flat Address Space

Global Segment within Flat Address Space
(Readonly & kernarg Segments have been combined with Global Segment in this example)

Group Segments within Flat Address Space

Private Segments within Flat Address Space
(Arg & Spill Segments are combined with Private Segments)