HSA (HETEROGENEOUS SYSTEM ARCHITECTURE) FROM A SOFTWARE PERSPECTIVE

OCT 2013

GARY FROST
AMD SOFTWARE FELLOW
HSA FOUNDATION

- Founded in June 2012
- Developing a new platform for heterogeneous systems
- www.hsafoundation.com
- Specifications under development in working groups
- Our first specification, HSA Programmers Reference Manual is already published and available on our web site
- Additional specifications for System Architecture, Runtime Software and Tools are in process
<table>
<thead>
<tr>
<th>Founders</th>
<th>AMD</th>
<th>ARM</th>
<th>Imagination</th>
<th>MediaTek</th>
<th>Qualcomm</th>
<th>Samsung</th>
<th>Texas Instruments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Promoters</td>
<td>LG Electronics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supporters</td>
<td>Arteris</td>
<td>codeplay</td>
<td>FABRIC Engine</td>
<td>Multicore Ware</td>
<td>Sandia National Laboratories</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contributors</td>
<td>ANALOG DEVICES</td>
<td>apical</td>
<td>BROADCOM</td>
<td>CEVA</td>
<td>DMP</td>
<td>ITRI</td>
<td>Swarm64</td>
</tr>
<tr>
<td>Academic</td>
<td>NTHU Programming Language Lab</td>
<td>NTHU System Software Lab</td>
<td>University of Bristol</td>
<td>University of Pennsylvania Informatics</td>
<td>ILLINOIS Computer Science</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Associates</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SOCS HAVE PROLIFERATED — MAKE THEM BETTER

- SOCs have arrived and are a tremendous advance over previous platforms
- SOCs combine CPU cores, GPU cores and other accelerators, with high bandwidth access to memory
- How can we make them even better?
  - Easier to program
  - Easier to optimize
  - Higher performance
  - Lower power
- HSA unites accelerators architecturally
- Early focus on the GPU compute accelerator, but HSA goes well beyond the GPU
INFLECTIONS IN PROCESSOR DESIGN

Single-Core Era

Enabled by:
✓ Moore’s Law
✓ Voltage Scaling

Constrained by:
✗ Power
✗ Complexity

Assembly ➔ C/C++ ➔ Java …

Multi-Core Era

Enabled by:
✓ Moore’s Law
✓ SMP architecture

Constrained by:
✗ Power
✗ Parallel SW
✗ Scalability

 pthreads ➔ OpenMP / TBB …

Heterogeneous Systems Era

Enabled by:
✓ Abundant data parallelism
✓ Power efficient GPUs

Temporarily Constrained by:
✗ Programming models
✗ Comm. overhead

Shader ➔ CUDA ➔ OpenCL ➔ C++ and Java

Single-thread Performance

we are here

Time

Throughput Performance

we are here

Time (# of processors)

Modern Application Performance

we are here

Time (Data-parallel exploitation)
HIGH LEVEL FEATURES OF HSA

- Features currently being defined in the HSA Working Groups**
  - Unified addressing across all processors
  - Operation into pageable system memory
  - Full memory coherency
  - User mode dispatch
  - Architected queuing language
  - High level language support for GPU compute processors
  - Preemption and context switching

** All features subject to change, pending completion and ratification of specifications in the HSA Working Groups
HSA — AN OPEN PLATFORM

- Open Architecture, membership open to all
  - HSA Programmers Reference Manual
  - HSA System Architecture
  - HSA Runtime

- Delivered via royalty free standards
  - Royalty Free IP, Specifications and APIs

- ISA agnostic for both CPU and GPU

- Membership from all areas of computing
  - Hardware companies
  - Operating Systems
  - Tools and Middleware
HSA MEMORY MODEL

- Defines visibility ordering between all threads in the HSA System

- Designed to be compatible with C++11, Java, OpenCL and .NET Memory Models

- Relaxed consistency memory model for parallel compute performance

- Visibility controlled by:
  - Load.Acquire
  - Store.Release
  - Barriers
HSA QUEUING MODEL

- User mode queuing for low latency dispatch
  - Application dispatches directly
  - No OS or driver in the dispatch path

- Architected Queuing Layer
  - Single compute dispatch path for all hardware
  - No driver translation, direct to hardware

- Allows for dispatch to queue from any agent
  - CPU or GPU

- GPU self enqueue enables lots of solutions
  - Recursion
  - Tree traversal
  - Wavefront reforming
**HSA INTERMEDIATE LAYER — HSAIL**

- **HSAIL** is a virtual ISA for parallel programs
  - Finalized to ISA by a JIT compiler or “Finalizer”
  - ISA independent by design for CPU & GPU

- Explicitly parallel
  - Designed for data parallel programming

- Support for exceptions, virtual functions, and other high level language features

- Lower level than OpenCL SPIR
  - Fits naturally in the OpenCL compilation stack

- Suitable to support additional high level languages and programming models:
  - Java, C++, OpenMP, etc
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”)
SEGMENTS AND MEMORY (1/2)

- 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.

```
ld_global_u64 $d0, [$d6]
ld_group_u64 $d0, [$d6+24]
st_spill_f32 $s1, [$d6+4]
```
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 (i.e., 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*D + 4*Q <= 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
WAVEFRONTS

- 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

```c
if (cond) {
    operationA; // cond=True lanes active here
} else {
    operationB; // cond=False lanes active here
}
```

- Tradeoffs
  - “Wavefront-aware” programming can be useful for peak performance
  - But results in less portable code (since wavefront width is encoded in algorithm)
HSA ENABLEMENT OF LANGUAGES, FRAMEWORKS, LIBRARIES AND RUNTIMES
HSA AND OPENCL™

- HSA is an optimized platform architecture for OpenCL™
  - Not an alternative to OpenCL™

- OpenCL™ on HSA will benefit from
  - Avoidance of wasteful copies
  - Low latency dispatch
  - Improved memory model
  - Pointers shared between CPU and GPU

- OpenCL™ 2.0 shows considerable alignment with HSA
  - Many HSA member companies are also active with Khronos in the OpenCL™ working group
HSA AND OPENMP®

- OpenMP®
  - Established
  - Portable
  - Scalable (desktop to supercomputer)
  - Simple
  - Flexible

- HSA enablement brings :-
  - GPU performance
  - Energy efficiency
  …to established developer community
BOLT: A C++ PARALLEL PRIMITIVES LIBRARY FOR HSA

- Allow C++ developers to leverage the power efficiency of GPU computing
  - Common routines such as scan, sort, reduce, transform
  - More advanced routines like heterogeneous pipelines
  - Bolt library works with OpenCL and C++ AMP

- Enjoy the unique advantages of the HSA platform
  - Move the computation not the data

- A single source code base for the CPU and GPU!
  - Developers can focus on core algorithms

[GitHub Link](https://github.com/HSA-Libraries/Bolt)
HSA ENABLEMENT OF JAVA™

- Why Java™?
  - 9 Million Developers
  - 1 Billion Java downloads per year
  - 97% Enterprise desktops run Java
  - 100% of blue ray players ship with Java
    

- Java™ 8 language/libraries include concurrency features
  - primitives (threads, locks, monitors, atomic ops)
  - libraries (fork/join, thread pools, executors, futures)
  - support for ‘lambda’ based Stream API’s

- JIT (Just In Time) architecture ideal for generating and executing HSAIL.
  - Project ‘Sumatra’ targets GPU JIT generation/execution in the 2015 Java™ 9 timeframe.
Aparapi API for expressing data parallel workloads

- Developer uses common Java™ patterns and idioms
- Java source compiled to (bytecode) using standard compiler (javac)

Aparapi runtime capable of converting bytecode to OpenCL™

- Execution on OpenCL™ 1.1+ capable devices (GPUs/APUs)
- OR
- Execute via a Java thread pool if OpenCL™ is not available

Open Source project

- ~20 contributors
- >7000 downloads
- ~150 visits per day
• AMD/Oracle sponsored Open Source (OpenJDK) project
• Targeted at Java 9 (2015 release)
• Allow developers to efficiently represent data parallel algorithms in Java using Stream API + Lambda expressions
• Sumatra is not pushing new ‘programming model’
• Instead we ‘repurpose’ Java 8’s new Stream API/Lambda to enable both CPU or GPU computing
• A Sumatra enabled Java Virtual Machine will dispatch ‘selected’ constructs to HSA enabled devices at runtime.
• Developers already refactoring JDK to use stream+lambda
  – So anyone using existing JDK should see GPU acceleration without any code changes.

http://openjdk.java.net/projects/sumatra/
https://wikis.oracle.com/display/HotSpotInternals/Sumatra
http://mail.openjdk.java.net/pipermail/sumatra-dev/
HSA ENABLEMENT OF JAVA

Java 7 – OpenCL enabled Aparapi

- AMD initiated Open Source project
- APIs for data parallel algorithms
  GPU accelerate Java application:
  - No need to learn OpenCL
- Active community captured mindshare
  ~20 contributors
  >7000 downloads
  ~150 visits per day

Java 8 – HSA enabled Aparapi

- Initial release
- Supports Java 8 Lambdas
- Dispatch code to HSA enabled devices at runtime via HSAIL

Java 9 – HSA enabled Java (Sumatra)

- Adds native GPU compute support to Java Virtual Machine (JVM)
- Developer uses JDK provided Lambda + Stream API
- JVM uses GRAAL compiler to generate HSAIL
- JVM decides at runtime to execute on either CPU or GPU depending on workload characteristics.

We plan to provide HSA Enabled Aparapi (Java 8) as a bridge technology between OpenCL based Aparapi (Java 7) and HSA Enabled Sumatra (Java 9).
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;
    }
} // Setters omitted for brevity

Player[] allPlayers = ... // Code to initialize array of Players omitted

Device.hsa().forEach(allPlayers, p -> {
    // HSA enabled Aparapi
    int teamScores = p.getTeam().getScores();
    float pctOfTeamScores = (float)p.getScores()/(float) teamScores;
    p.setPctOfTeamScores(pctOfTeamScores);
});
01: version 0:95: $full : $large;
02: // We pass underlying array of Players to the kernel
03: kernel &run (
04:     kernarg_u64 %_arg0                        // Array of players passed as arg
05: ){
06:     ld_kernarg_u64 $d6, [%_arg0];           // Move arg to an HSAIL register
07:     workitemabsid_u32 $s2, 0;             // Read the work-item global id (gid)
08:     cvt_u64_s32 $d2, $s2;                   // Convert gid to long
09:     mul_u64 $d2, $d2, 8;                   // Stride for sizeof(int) elements
10:     add_u64 $d2, $d2, 24;                 // Skip array object header (24 bytes)
11:     add_u64 $d2, $d2, $d6;                // $d2 now points to players[$d2]
12:     ld_global_u64 $d6, [$d2];            // Load Player p from players[$d2]
13:     ld_global_u64 $d0, [$d6 + 120];       // p.getTeam() inlined
14:     ld_global_u64 $d6, [$d6 + 40];         // p.getScores() inlined
15:     cvt_f32_s32 $s3, $s3;                  // cast to (float)
16:     cvt_f32_s32 $s0, [$d0 + 24];          // Team getScores() inlined
17:     div_f32 $s16, $s16, $s17;             // p.getScores() / teamScores inlined
18:     st_global_f32 $s16, [$d6 + 100];      // p.setPctOfTeamScores() inlined
19:     ret;
20: }
21:
A Java developer implementing Nbody would probably...

```java
// Create a class to represent each body
class Body{
    float x, y, z, m, vx, vy, vz;
    // Include method to update position and display
    void updateAndShow(Screen screen, Body[] bodies){
        // omitted vars for accumulating forces
        for (Body other:bodies){
            // accumulate forces between other and this
        }
        // update vx, vy, vz, x, y and z from accumulated data
        screen.paint(x, y, z);
    }
}
```

// Assuming bodies[] is an initialized array of Body
// We can update and display each one in turn
for (Body b: bodies)
    b.updateAndShow(screen, bodies);

Java does not guarantee contiguous allocation of objects in arrays

Only arrays of primitives (long, float etc) are allocated contiguously

Non HSA enabled Java GPU frameworks force developers to either

- Abandon Object Oriented solutions and revert to parallel primitive arrays
- Or...
  - Add scatter/gather (costly copies) behind the scenes

```java
// Create and populate parallel arrays of primitives
float x[], y[], z[], m[], vx[], vy[], vz[];
// Treat x[n],y[n],z[n] etc as the state of Body[n]
Kernel k = new Kernel()
    .void run()
    .for (int j=0; j<bodies j++)
        // accum forces between (x,y,z)[j] and (x,y,z)[i]
    .for (int j=0; j<bodies j++)
        // update vx[j],vy[j],vz[j],x[j],y[j] and z[j]
    );
k.execute(bodies);
```
HSA ENABLEMENT ALLOWS NATURAL JAVA REPRESENTATIONS

- HSA version of Aparapi and Sumatra can deal with Java objects

```java
class Body{
    float x, y, z, m, vx, vy, vz;
    void updateAndShow(Screen screen, Body[] bodies){
        // hidden vars for accumulating forces
        for (Body other:bodies){
            // accumulate forces between other and this
        }
        // update vx, vy, vz, x, y and z from accumulated data
        screen.paint(x, y, z);
    }
}
```

- Then loop over the array, updating and displaying the bodies.

```java
Device.hsa().forEach(bodies, b -> { //HSA enabled Aparapi solution
    b.updateAndShow(screen, bodies);
});
```
NBody implemented as an array of Objects.
On early access HSA enabled hardware and software.

- 7.9 x perf (1.35 x power)
- 10.6 x perf (1.44 x power)
- 12.3 x perf (1.48 x power)
HSA ENABLEMENT OF JVM CAN ACCELERATE OTHER JVM BASED LANGUAGES

Java 9 – 3Q2015
HSA enabled Java (Sumatra)

- Adds native GPU compute support to Java Virtual Machine (JVM)
- Developer uses JDK provided Lambda + Stream API
- JVM uses GRAAL compiler to generate HSAIL
- JVM decides at runtime to execute on either CPU or GPU depending on workload characteristics.

Java 9 + 2016?
HSA enablement of other JVM based languages/frameworks

- Developer uses their preferred Truffle based language (R, Javascript, Python, Runby etc)
- JVM uses Truffle + GRAAL compiler to generate HSAIL
- HSA acceleration beyond Java
TAKEAWAYS

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

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

- Java Enablement
  - Can access Objects on Java’s heap thanks to ‘Shared Virtual Memory’
  - Leverages Java 8 Lambda and Stream APIs intended for multicore
  - Gateway to enabling other JVM based languages.
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 enablement via 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/)
  - [http://aparapi.googlecode.com/](http://aparapi.googlecode.com/)