HSA-ENABLED DSP AND ACCELERATORS

JOHN GLOSSNER, GENERAL PROCESSOR TECHNOLOGIES, INC

PAUL BLINZER, ADVANCED MICRO DEVICE, INC

JARMO TAKALA, TAMPERE UNIVERSITY OF TECHNOLOGY, FINLAND

HETEROGENEOUS PROCESSORS HAVE PROLIFERATED — MAKE THEM BETTER

Heterogeneous SOCs have arrived, a tremendous advance over previous platforms

- Today SOCs combine many programmable blocks with high bandwidth access to memory
  - CPU cores, GPU cores, audio and other domain specific processors and accelerators
  - Each one with their own ISA, micro-architecture and language tool chain
- But: require specialized APIs or languages to target the processors

How do we make them even better?

- Easier to program
- Easier to optimize
- Higher performance
- Lower power

THE GOALS OF THE HETEROGENEOUS SYSTEM ARCHITECTURE

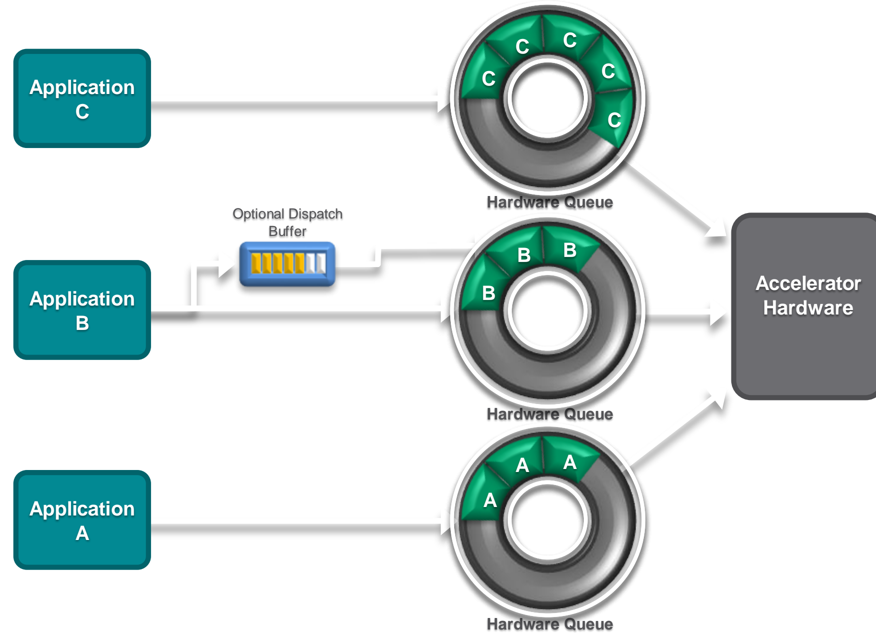
- Create a platform architecture accessible for all accelerator types
  - Focus on compute, allowing other accelerator types to participate
  - HSA unites accelerators architecturally, benefiting platform integration and system software
- Attract mainstream programmers
  - By making it easier writing data parallel code
  - Native high-level language support of a broader set beyond traditional accelerator languages
  - Support for Task Parallel & Nested Data Parallel Runtimes
  - Rich debugging and performance analysis support
- It is not dictating a specific platform or component microarchitecture
  - It defines the “what needs to be supported”, not the “how it needs to be supported”
  - Focus is on providing a robust data parallel application execution infrastructure that use familiar ways to write, reuse and maintain the software and use familiar language tool chains

THE PILLARS OF HSA

- To bring accelerators forward as a first class processor within the system
  - **Unified process address space across all processors (Shared Virtual Memory)**
  - Well-defined relaxed consistency memory model suited for many high level languages
  - Memory coherency between the CPU and HSA agents to simplify “data collaboration”
    - But supporting coarse-grain access for specialized task accelerators
  - Architected signal and event notification mechanisms between processers
  - Architected User mode dispatch/scheduling (eliminates “drivers” from dispatch path)
  - QoS through pre-emption and context switching*
  - HSAIL: data parallel machine language target for runtime/compiler tool chains
    - OpenCL, C/C++ and other high level language tool chains
  - Open specifications and open source reference software and tool chains available

© Copyright 2012-2015 HSA Foundation. All Rights Reserved.

HSA COMMAND AND DISPATCH FLOW



SW view:

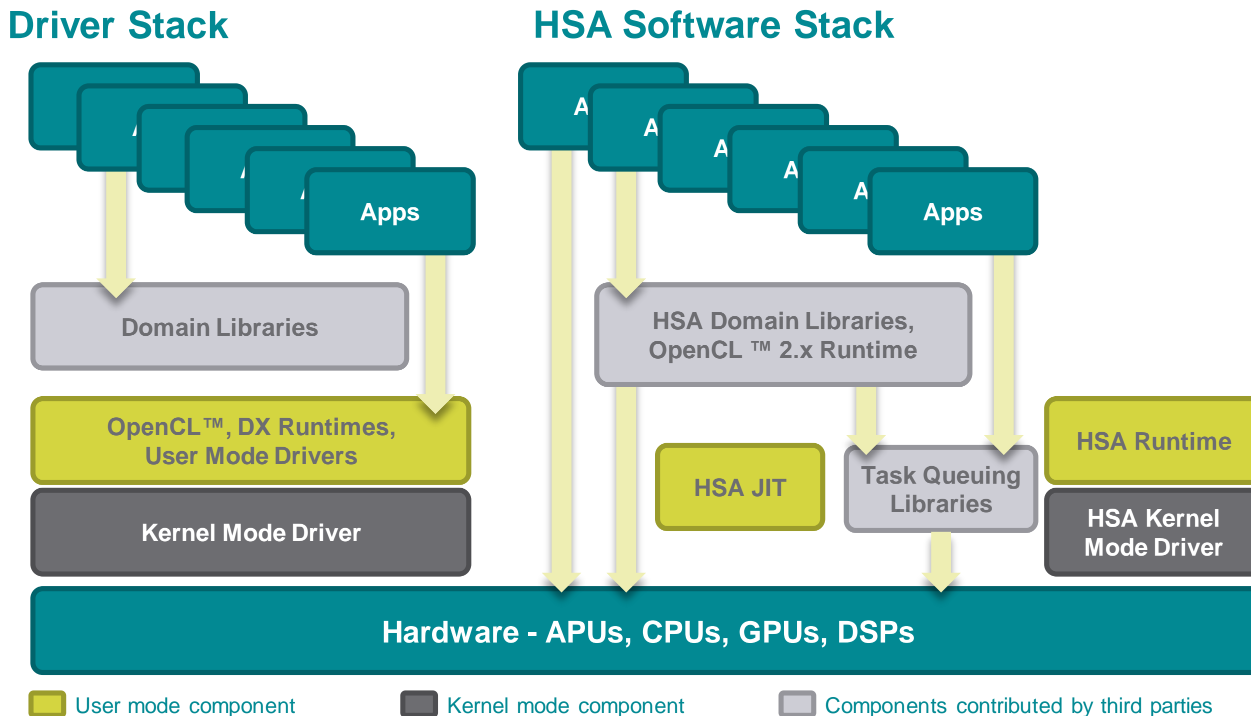
- User-mode dispatches to HW
- No Kernel Driver overhead
- Low dispatch times
- CPU & Accelerator dispatch APIs

HW view:

- HW / microcode controlled
- HW scheduling
- Architected Queuing Language (AQL)
- HW-managed protection

© Copyright 2012-2015 HSA Foundation. All Rights Reserved.

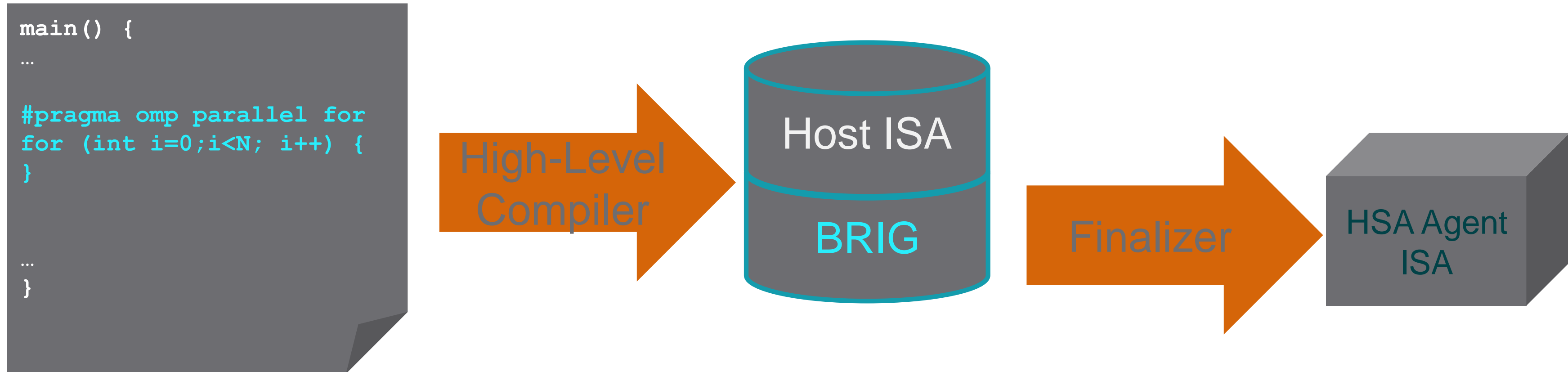
HSA - EVOLUTION OF THE SOFTWARE STACK



© Copyright 2012-2015 HSA Foundation. All Rights Reserved.

WHAT IS HSAIL?

- Intermediate language for parallel compute in HSA
  - Generated by a “High Level Compiler” (GCC, LLVM, Java VM, etc)
  - Expresses parallel regions of code
  - Binary format of HSAIL is called “BRIG”
  - *Goal: Bring parallel acceleration to mainstream programming languages*



© Copyright 2012-2015 HSA Foundation. All Rights Reserved.

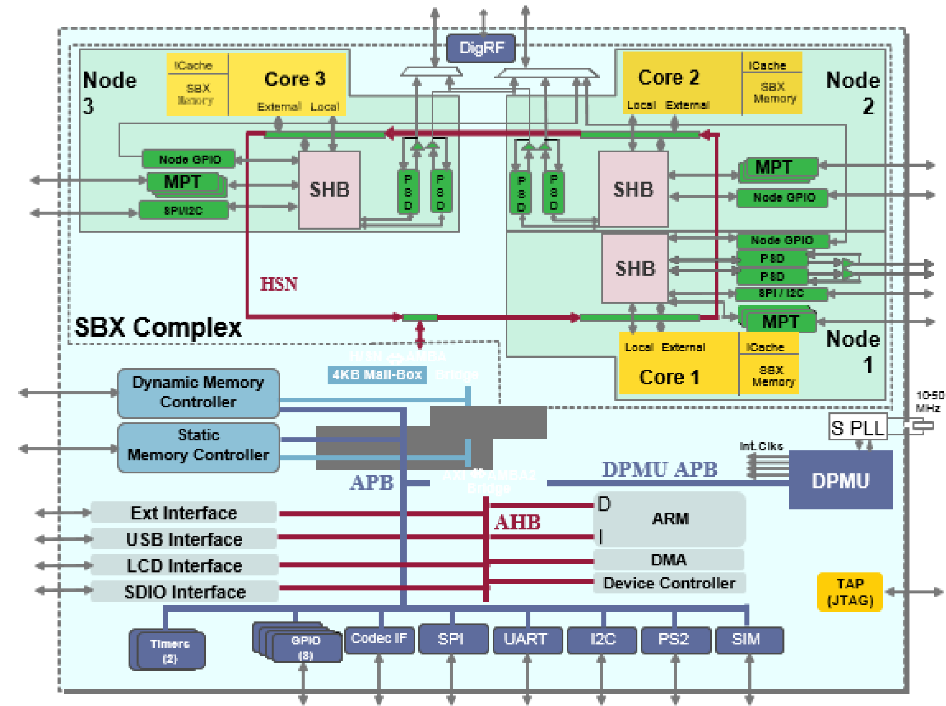
HSAIL: HSA INTERMEDIATE LAYER

- A Virtual Explicitly Parallel ISA
  - ~135 Opcodes
  - RISC Register-based Load/Store
  - Branches & Function Calls
  - Atomic Operations
  - Arithmetic
    - IEEE 754 Floating Point
    - including 16-bit Integer (32/64-bit)
    - DSP fixed point
    - Packed / SIMD
      - f16x2, f16x4, f16x8,f32x2, f32x4, f64x2
      - signed/unsigned 8x4, 8x8, 8x16, 16x2,
      - 16x4, 16x8, 32x2, 32x4, 64x2

- Wavefronts
  - 1, 2, 4, 8, 16, 32, or 64 SIMD lanes
  - Lanes can be active or inactive
- Memory
  - Shared Virtual Memory
- Exceptions

SB3500

- 3 Sandblaster 2.0 DSPs
  - 600MHz, 4-way threaded
  - 32KB I-Cache, 256KB D-memory
  - 256b Vector Unit, 16 x 16b MACs
  - 9600 MMAC/s
- HSN interconnect
  - ring
  - 2.4GBs/link
- ARM, 65nm LP
- Full Production



MAPPING HSAIL ONTO SB3500

- HSA
  - HSA agent
  - Compute Unit
  - Processing Element
  - Work-Item
  - Work-Group
  - Wavefront
  - Packed / SIMD

- SB3500
  - SB3500 DSP Complex
    - 3 DSP cores
  - SB3500 DSP Core
  - SB3500 Thread Unit
  - Software Thread
  - Multi-threaded App
    - FIR filter, FFT, etc.
  - Vector

MAPPING HSAIL TO DSP - DOT PRODUCT

HSA FOUNDATION

- The dot product of two vectors is

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

OPENCL DOT PRODUCT

```
#define WORK_ITEM_COUNT 256

__kernel void vector_dot_product(
    __global const double* vec1,
    __global const double* vec2,
    __global double* result,
    uint vec_len
    )
{
    // save the work-item id for future use
    uint id = get_global_id(0);
    // array of accumulators in local memory
    // shared by all work-items
    __local float acc_array[LOCAL_GROUP_XDIM];
    // private accumulator
    float acc = 0;

    // find a sum of (vec_len/WORK_ITEM_COUNT) products
    for (i = id; i < vec_len; i += WORK_ITEM_COUNT) {
        acc += vec1[i]*vec2[i];
    }

    // store the result into the accumulator array
    acc_array[id] = acc;

    // wait until all work-items finish modifying
    // the accumulator array.
    barrier(CLK_LOCAL_MEM_FENCE);

    // perform the horizontal reduction of acc_array;
    // this will take log2(WORK_ITEM_COUNT) iterations
    for (reduce_distance = (WORK_ITEM_COUNT >> 1);
            reduce_distance > 0;
            reduce_distance >>= 1) {
        if (id + dist < WORK_ITEM_COUNT)
            acc += acc_array[id + dist];
            acc_array[id] = acc;
        }
        // wait until all work-items finish modifying
        // the accumulator array.
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if (id == 0) {
        // work-item 0 stores the result
        *result = acc;
    }
}

/* vi: set ts=4 sw=4 et filetype=c: */
```

C/C++ DOT PRODUCT

```
// Typical DSP Inner Loop
float out[N];
float in[N+T-1];
float taps[T];
for( i = 0; i < N; i++ )
   v = 0;
   for( j = 0; j < T; j++ )
     v += in[i+j]*taps[j];
   out[i] = v;

// Typical Graphics Inner Loop
float out[N][4];
float in[N][4];
float matrix[4][4];
for( i = 0; i < N; i++ )
    for( j = 0; j < 4; j++ )
    v = 0;
         for( k = 0; k < 4; k++ )
              v += in[i][j] * matrix[j][k];
          out[i][j] = v;
```

HSAIL DOT PRODUCT (1)

```
kernel &vector_dot_product(
   kernarg_u32 %vec1,
   kernarg_u32 %vec2,
   kernarg_u32 %result,
   kernarg_u32 %vec_len)
{
   // array of accumulators in local memory
   // shared by all work-items
   group_f32        &acc_array[WORK_ITEM_COUNT];
   // s0 = sizeof(float)*id
   workitemid_u32    $s0,0;
   mul_u32           $s0,$s0,4;

   // s2 = vec1
   ld_kernarg_u32    $s2,[%vec1];
   // s3 = vec2
   ld_kernarg_u32    $s3,[%vec2];

   // set accumulator to 0
   mov_f32           %s4,0.0f

   // s5 = sizeof(float)*vec_len
   ld_kernarg_u32    %s5,[%vec_len]
   mul_u32           $s5,$s5,4;

// set outer loop counter in s6 to sizeof(float)*id
   mov_u32           $s6,$s0;
   cmp_b1_u32_ge     $c0,$s5,$s6;
   cbr               $c0,@update_acc_array;

   // accumulation loop

@next_element:
   // load the next vec1 element into s7
   ld_global_f32     %s7,[%s2];
   add_u32           $s2,$s2,4;
   // load the next vec2 element into s8
   ld_global_f32     %s8,[%s3];
   add_u32           $s3,$s3,4;
   // multiply and accumulate a pair of vector elements
   // acc += vec1[i]*vec2[i]
   fma_f32           $s4,$s7,$s8,$s4
   // increment the loop variable and exit the loop when it 0
   add_u32           $s6,$s6,WORK_ITEM_COUNT
   cmp_b1_u32_lt     $c0,$s5,$s6;
   cbr               $c0,@next_element;
```

HSAIL DOT PRODUCT (2)

```
@update_acc_array:
   // store the result into the accumulator array
   // acc_array[id] = acc;
   lda_group_u32      $s10,[%acc_array]
   add_u32            $s10,$s10,$s0
   st_group_f32       $s4,[$s10]

   // wait until all work-items finish modifying
   // the accumulator array.
   barrier;

   // horizontal reduction loop

   // reduction loop initialization
   mov_u32            $s11,(WORK_ITEM_COUNT/2)
   cmp_b1_u32_eq      $c0,$s11,$s11,0
   cbr                $c0,@store_result

@reduce_next:
   // skip reduction on some work-items
   mad_u32            $s12,$s11,4,$s0
   cmp_b1_u32_lt      $c0,$s12,WORK_ITEM_COUNT
   cbr                $c0,@reduce_barrier

// reduce two elements from the accumulator array
   ld_group_f32       $s13,[$s12]
   add_f32            $s4,$s4,$s13
   st_group_f32       $s4,[$s10]

@reduce_barrier:
   // wait until all work-items finish modifying
   // the accumulator array.
   barrier;

   // continue until only one value remains in
   // the accumulator array
   shr_u32            $s11,$s11,1
   cmp_b1_u32_ne      $c0,$s11,$s11,0
   cbr                $c0,@reduce_next

@store_result:
   // work-item 0 stores the result
   cmp_b1_u32_ne      $c0,$s0,0
   cbr                $c0,@exit
   ld_kernarg_u32     $s11,[%result];
   st_global_f32      $s4,[$s11]

@exit:
   ret;
```

SB3500 DOT PRODUCT

```
/* int sb3500_dot_product(const short* vec1, const short* vec2, int
       vec_len) */

FNDEFN_BEGIN(sb3500_dot_product)

   // r15 = vec_len/16 - 1
   shri      %r15,%r11,4
   addi      %r15,%r15,-1
   ctsr      %r15,MACH_LC0

   // load first 16 elements of vec1 into vr0
   lr        %vr0,%r8,0
   || lir    %r14,0

   // load first 16 elements of vec1 into vr0 & set ac0 to 0
   lr        %vr1,%r9,0
   || ctsr   %r14,MACH_AC0

NEXT_16_SAMPLES:
   // multiply and reduce the current vr0 anv vr1
   // load next 16 elements from vec1 and vec2
   lru       %vr0,%r8,32
   || rmulred  %ac0,%vr0,%vr1,%ac0
   lru       %vr1,%r9,32
   ||  loop    0,%lc0,NEXT_16_SAMPLES

   // move result into register r8
   cfsr      %r8,MACH_AC0

   // exit
   ji        %jt0,8

FNDEFN_END(sb3500_dot_product)
```

SB3500 INSTRUCTIONS NOT IN HSAIL

HSA FOUNDATION

- SIMD reduce with/without Mask
  - sum of all elements in a SIMD register
  - find the maximum element value in an SIMD register
  - find the minimum element value in an SIMD register
- Multiply-reduce
  - Or just the horizontal reduction
- Rotate a pair of vector registers by a single element
  - The element with the highest index from one register is shifted into the lowest index of another register.
- Complex vector multiply
  - de-rotation
- SIMD formats for bit operations
  - Entropy encoding
  - Bit correlations
  - Encryption
- Special Purpose Instructions
  - FFT
  - Viterbi/Turbo/LDPC
  - Galois Field

RESULTS(1)

- HSAIL Includes Most Instructions Inherent in DSP Applications
  - Even when absent, some algorithms can be changed
- DSP’s tend to use compound operations that HSAIL specifies independently
  - Vector load with update
  - Decrement and Branch
  - Vector MAC
    - No vector FMA in HSAIL (really only need scalar->vector and vector->vector)
  - Rotate Vector Element
    - ld_global_b128 inefficient (throw away 7 elements)
  - Horizontal Reductions
    - Matrix * Vector operations
    - Long Vector Dot Products

RESULTS(2)

- Peep hole optimization may be difficult but possible
  - Optimizing HSAIL compiler should try to maximize the distance between Load and Use
  - May be hard to find some patterns (register operands)
  - Easier to break apart compound instructions versus coalescing them
  - Finalizer is available to identify and coalesce instructions on the platform
  - Standard may see extensions to better accommodate DSP
- NOT SHOWN: To achieve full throughput you need to unroll the loops
  - In excessive cases eventually may cause register pressure

CONCLUSIONS

HSA FOUNDATION

- HSAIL is Scalable Heterogeneous System Methodology
  - Solves H/W Integration and S/W Programming of Heterogeneous Systems
  - A RISC-based Virtual Machine
- Many DSP Algorithms Can Be Efficiently Described in HSAIL
  - Includes SIMD Fixed Point Types / F16 Vector Floating Point
  - Restructured Algorithms Can Improve Performance
- Some DSP Operations May Be More Efficiently Described With Compound Operations
  - Common to many DSPs
  - Lower Power (reduced loads)
  - Easier For Compiler to Break Apart Compound Operations
    - Additional Unrolling of Loops May Cause Register Pressure

MEMBERS DRIVING HSA

HSA FOUNDATION

Founders

Promoters

Supporters

Contributors

Academic

ANY QUESTIONS?

- Of course there are, so go ahead ☺