SUMATRA REDUCTION ON HSA APU

ERIC CASPOLE

JVM LANGUAGE SUMMIT 2014

OVERVIEW

- Intro to Stream API functions we are offloading
- Stream API Pipeline details
- Background on the algorithm we are using
- Pseudocode walk through
- Performance gotchas
- Future plans

SUMATRA REDUCTION ON HSA APU

- Stream API offers IntStream.reduce(IntBinaryOp x)
  - Examples
    - Integer::sum
    - Integer::max
- Reduce returns single integer result from an IntStream

- With forEach operations, each element is independent with no return value
  - The Stream.forEach lambda was the first target for HSA offload
- With reduce operations, return value comes from combining the elements

- Reductions are parallelizable: s.parallel().reduce(Integer::sum)
  - Parallel CPU: ~2-32 cores, APU: hundreds or thousands of stream cores
- Offload some reduce operations to the HSA APU

REDIRECT PARALLEL REDUCE INTO OFFLOAD PATH

- Code is available in the Sumatra JDK and Graal repository
  - http://hg.openjdk.java.net/sumatra/sumatra-dev/jdk/
  - http://hg.openjdk.java.net/graal/graal/

- Small modifications to IntStream JDK classes to divert to Graal HSAIL code
  - Similar to forEach diversion which was first Sumatra offload target
  - Extended the decoding of the Stream pipeline used for forEach

- Currently no debug info or deoptimization supported for reduce
  - The kernel is built from a hand crafted Java String of HSAIL source
  - Not backed by Java bytecode at this time
  - Does not work with compressed oops yet

DECODING THE PIPELINE FOR THE OFFLOAD PATH

- Stream Pipeline uses inner classes for operations
- Can now offload forEach and reduce, which are "terminal operations"
- Terminal operation is a subclass of Sink
  - For forEach: looking for klassName.contains("ForEachOps$ForEachOp$")
  - For reduce: looking for "java.util.stream.ReduceOps$5ReducingSink"
  - For filter: looking for klassName.endsWith("ReferencePipeline$2$1")
- Using a lot of reflection/unsafe/hard-coded names we read in the code
- Very fragile if Stream API is revised/bug fixes etc

DECODING THE STREAM PIPELINE WISH LIST

- We hope there can be an official JDK API to decompose the Stream Pipeline
- See the operations in the pipeline
  - Is it filter/peek/map etc?
- Get the stream source array reference
  - Need to send this pointer to the offload kernel
  - Stream source might be ArrayList, Vector, or simple array
- See how many operations are in the pipeline
  - Currently we offload only 1 step pipelines but plan to do more
  - We had multi-step pipeline offload working in an earlier OpenCL based version
- Get the lambda function used in a pipeline step
  - This is what gets compiled into the HSA kernel
  - Currently using Graal to extract it from Consumer.apply()
- Get the captures coming into the lambda from the Consumer
  - These are arguments to the HSA kernel

DIVIDE AND CONQUER THE REDUCE INTO GROUPS

- These reduce operations can be divided and conquered
  - Accumulate final result in the last step
- Unit of dividing work on a GPU is the “group”
- Group is a unit of work in GPU parlance of HSA and OpenCL
  - Group size is usually multiples of 64 and configurable per task
  - 1 or more wavefronts in a group
  - A wavefront is the scheduling unit of stream cores running a kernel all in lock step
  - If grid size is not divisible by group size, there will be a partial final group
- Features of a group
  - All wavefronts of a group run at the same time
  - There can be synchronization among wavefronts in the group
  - Group memory shared inside group
  - Generally group memory is faster than global memory
    - Keep partial results here until group completes
    - Group memory is automatically reused once the current group completes

OVERVIEW OF OUR FIRST REDUCE IMPLEMENTATION

- First implementation is based on an algorithm from BOLT
  - BOLT is an open source C++ template library for OpenCL
  - In github: https://github.com/HSA-Libraries/Bolt

- The OpenCL source was compiled to HSAIL source text using internal tools
  - Tweaked by hand to be compatible with Java Sumatra system
  - Saved as a large String that gets compiled into a kernel by Graal
  - Becomes a Graal ExternalCompilationResult that does not support deoptimization

- This version supports Integer::sum, min, and max
  - Uses HSAIL atomics to accumulate final result to return to CPU

OVERVIEW OF OUR FIRST REDUCE IMPLEMENTATION

- Three arguments to kernel
  - Input int[] extracted from IntStream source
  - Length of int[] is explicitly passed to kernel
    - With HSA we are reading the array directly from the Java heap
    - Could read length from the array in the HSAIL, but it would be extra work per workitem
  - Java int[1] result (kernels have no “return value” in the normal sense)

- Grid size is a parameter to the kernel execute API
  - Called the global size or sometimes the range in OpenCL
  - Grid size indicates how many stream cores will work on this problem
  - For forEach kernels, the grid size equals the length of the input stream source array
  - This algorithm uses explicit length checks rather than length=grid size
  - Trade off which should get more work per wave rather than more very short waves

PART 1: REDUCE INPUT INTO WORKITEM LOCALS

COLOR CODED TO CODE SHOWN ON NEXT SLIDE

- Algorithm uses group size 256 by design
  - Uses group memory int[256] to build partial results
  - Could be made adjustable but 256 is working well so far
  - APU wavefront size is 64
- The grid size controls the looping in the kernel code
- Initialize a local variable with the element from the input[gid]
- Each workitem loops over the input array in grid size strides
  - Apply the reduce function with the new element against the local
  - Assign result to the local
  - Stride to next element
- Now the kernel has reduced the input array with results in per-workitem locals

PART 1 PSEUDOCODE

```
kernel void run(
    int* input_ptr,
    const int length,
    int* result,
)
{
    group int scratch[256] // scratch is group memory allocated by HSA runtime
                           // group size is 256
    int gx = get_workitemid();
    int gloId = gx;

    int accumulator;
    if(gloId < length){
       accumulator = input_iter[gx];
       gx += get_grid_size(0);
    }

    // Loop sequentially over chunks of input vector, reducing an arbitrary size input
    // length into a length related to the number of workgroups
    while (gx < length)
    {
        int element = input_iter[gx];
        accumulator = (*userFunctor)(accumulator, element);
        gx += get_grid_size(0);
    }
```

PART 2: THE FINAL ANSWER

- Next, local results are worked on in group memory
  - Group memory int[] of group size length allocated by the runtime
  - Store local variable result into this local memory int[group_id]
  - Apply reduce using lower half of the group (remember group size=256)
    - To (int[group_id], int[group_id + 128])
    - To (int[group_id], int[group_id + 64])
    - … To (int[group_id], int[group_id + 1])
  - Group barrier between each step to ensure all the workitems see the same values
- Now there is a partial result in each group memory int[0]
- Produce final result from group memory back to main memory
  - Each group’s workitem id 0 atomically emits its group memory result into the argument result int[1]
  - Other workitems in the group are predicated off and do nothing
  - Now the kernel is completed
- Host code returns the result int[1] as the result back up to IntStream.reduce

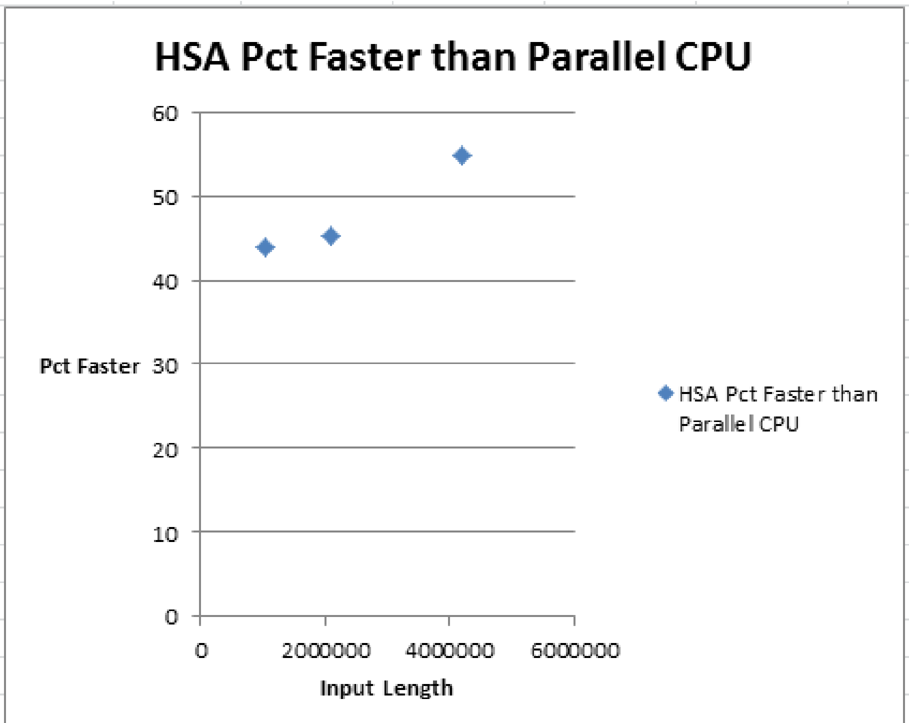
PART 2 PSEUDOCODE

```
// Initialize local data store
int local_index = get_local_id(0); // id inside this group
scratch[local_index] = accumulator;
barrier(CLK_LOCAL_MEM_FENCE);


// Tail stops the last workgroup from reading past the end of the input vector
uint tail = length - (get_group_id(0) * get_group_size(0));


// Parallel reduction within a given workgroup using local data store
// to share values between workitems - 256 is good to achieve high occupancy
_REDUCE_STEP(tail, local_index, 128);
_REDUCE_STEP(tail, local_index, 64);
_REDUCE_STEP(tail, local_index, 32);
_REDUCE_STEP(tail, local_index, 16);
_REDUCE_STEP(tail, local_index, 8);
_REDUCE_STEP(tail, local_index, 4);
_REDUCE_STEP(tail, local_index, 2);
_REDUCE_STEP(tail, local_index, 1);

 // Abort threads that are passed the end of the input vector
if( gloId >= length )
    return;


// Write only the single reduced value for the entire workgroup
if (local_index == 0) {
    atomic_add(&result[0], scratch[0]);
}
};
```

```
#define _REDUCE_STEP(_LENGTH, _IDX, _W) \
  if ((_IDX < _W) && ((_IDX + _W) < _LENGTH)) {\
  int mine = scratch[_IDX];\
  int other = scratch[_IDX + _W];\
  scratch[_IDX] = (*reduce)(mine, other); \
  }\
  barrier(CLK_LOCAL_MEM_FENCE);
```

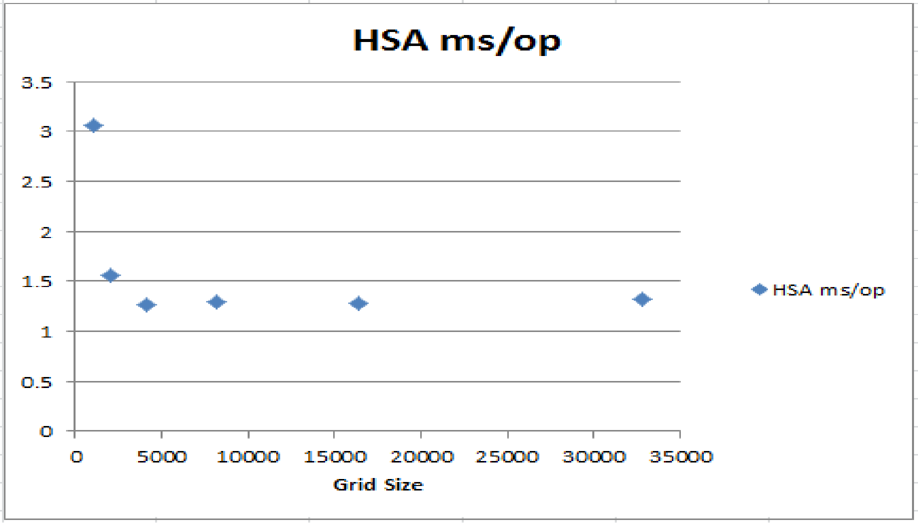
PERFORMANCE EXPERIMENT

- Integer::sum microbenchmark
- Kernel saved in Sumatra cache after compilation
- 2GHz CPU – representative of typical server clock rate, 4 CPU cores
- Grid size 16384 used here seems to be optimal for this scenario



PERFORMANCE GOTCHAS
TUNING GPU ALGORITHMS

- Grid size has an effect on performance
  - Let's say input int[] has length 4 million
  - Group size is 256
  - If grid size is 1024 for example, that is 4 groups/16 wavefronts of 64
  - Smaller grid size results in more looping to stride over the input
  - Berlin style APUs can run up to 40 wavefronts
  - Not all APU capability is utilized!
- For this algorithm bigger grid size is better until saturated



ALTERNATIVES

- This implementation depends on HSAIL atomics to produce the final result
- Other possibilities:
  - Run reduce kernel again on partial results
    - Might be worthwhile if partials array is really large
  - Return partial results array back to CPU to compute final answer
    - This is what the original BOLT code actually does
  - Borrow new algorithms from other GPU projects
- Trade off implementation/tuning complexity vs performance

FUTURE PLANS/CONCLUSION

- Plan to do regular compilation of reduce operations
  - Use Graal features such as replacements and snippets to insert group barriers etc
  - Avoid using hand crafted giant String of HSAIL
- Support other basic types besides int
- Compile user lambdas into reduce, not just built-ins like ::sum
- Support Object Stream reductions?
  - Need to think how to continue after deopt to interpreter
  - Current algorithm uses group memory
  - Extracting back to heap memory for Object fields will be tricky!

- Stream API Reductions are a promising workload to offload to HSA APU!

DISCLAIMER & ATTRIBUTION

ATTRIBUTION