ad-heap: an Efficient Heap Data Structure for Asymmetric Multicore Processors

Weifeng Liu and Brian Vinter

Niels Bohr Institute
University of Copenhagen
Denmark

{*weifeng, vinter*}*@nbi.dk*

March 1, 2014

Binary heap

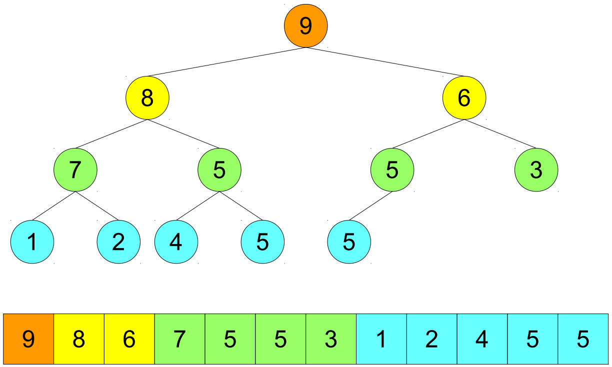
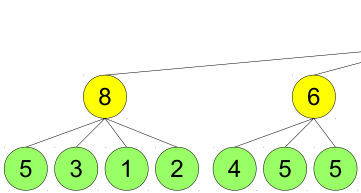


Figure: The layout of a binary heap (2-heap) of size 12.

Given a node at storage position i, its parent node is at $\lfloor (i-1)/2 \rfloor$, its child nodes are at $2i+1$ and $2i+2$.

d-heaps [Johnson, 1975]



9	8	6	7	5	5	3	1	2	4	5	5

Figure: The layout of a 4-heap of size 12.

For node i, its parent node is at $\lfloor (i-1)/d \rfloor$, its child nodes begin from $di+1$ and end up at $di+d$.

Cache-aligned *d*-heaps [LaMarca and Ladner, 1996]

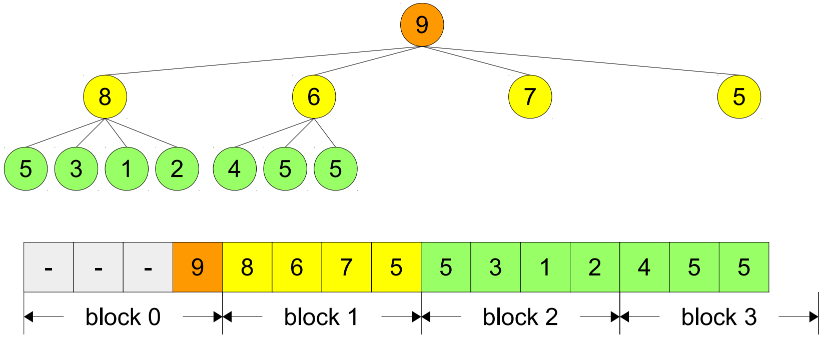


Figure: The layout of a cache-aligned 4-heap of size 12.

For node i, its parent node is at $\lfloor (i-1)/d \rfloor + offset$, its child nodes begin from $di + 1 + offset$ and end up at $di + d + offset$, where $offset = d - 1$ is the padded head size.

Operations on the *d*-heaps

insert

adds a new node at the end of the heap, increases the heap size to $n+1$, and takes $O(log_d n)$ worst-case time to reconstruct the heap property,

delete-max

copies the last node to the position of the root node, decreases the heap size to $n-1$, and takes $O(dlog_d n)$ worst-case time to reconstruct the heap property,

update-key

updates a node, keeps the heap size unchanged, and takes $O(dlog_d n)$ worst-case time to reconstruct the heap property.

Update-key operation on the root node (step 0)

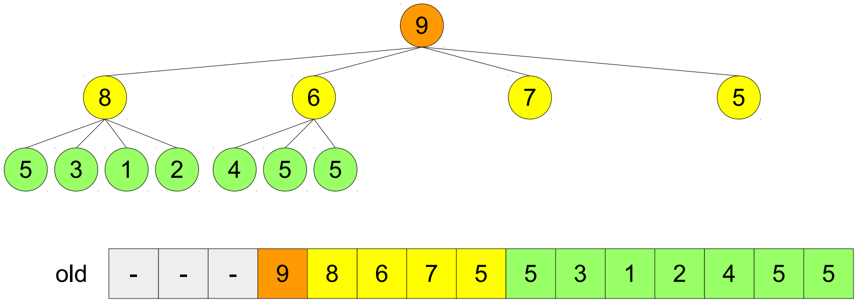
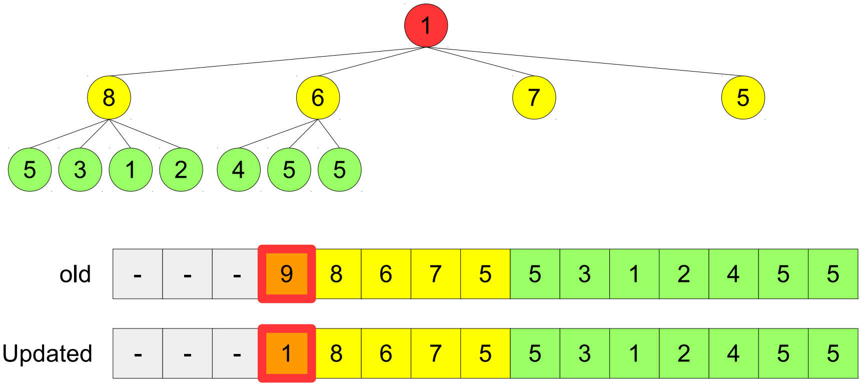


Figure: Initial status.

Update-key operation on the root node (step 1)



old	-	-	-	9	8	6	7	5	5	3	1	2	4	5	5
Updated	-	-	-	1	8	6	7	5	5	3	1	2	4	5	5

Figure: Update the value of the root node. Then the heap property on the level-1 and level-2 might be broken.

Update-key operation on the root node (step 2)

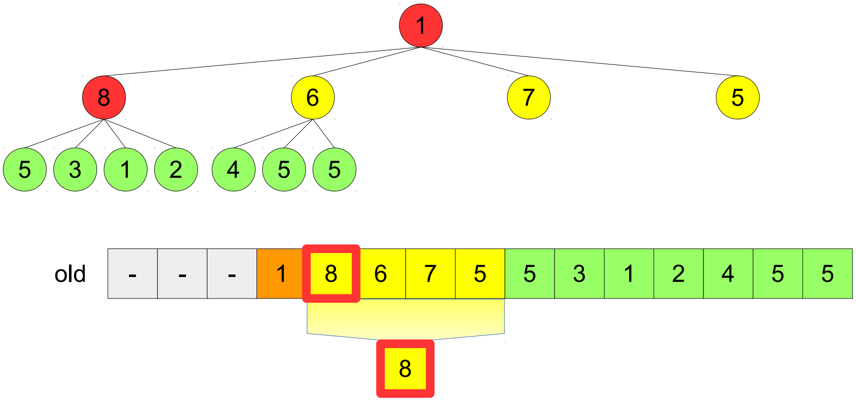


Figure: Find the maximum child node of the updated parent node.

Update-key operation on the root node (step 3)

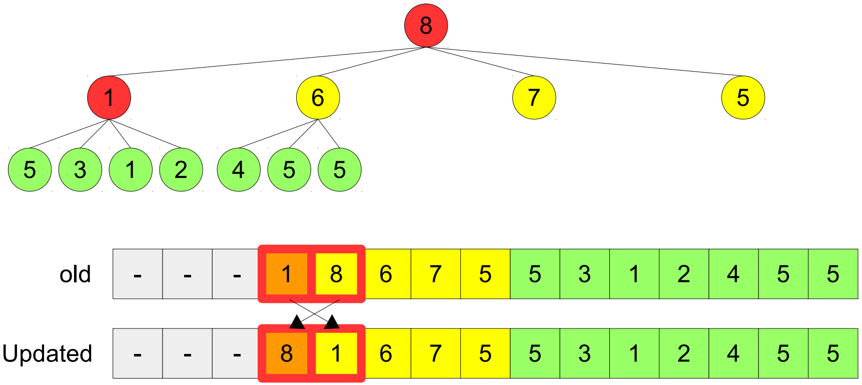


Figure: Compare, and swap if the max child node is larger than its parent node. Then the heap property on the level-2 and level-3 might be broken.

Update-key operation on the root node (step 4)

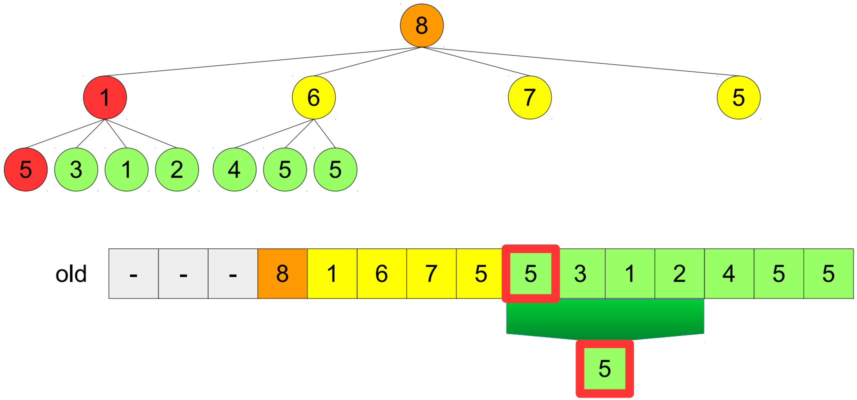


Figure: Find the maximum child node of the updated parent node.

Update-key operation on the root node (step 5)

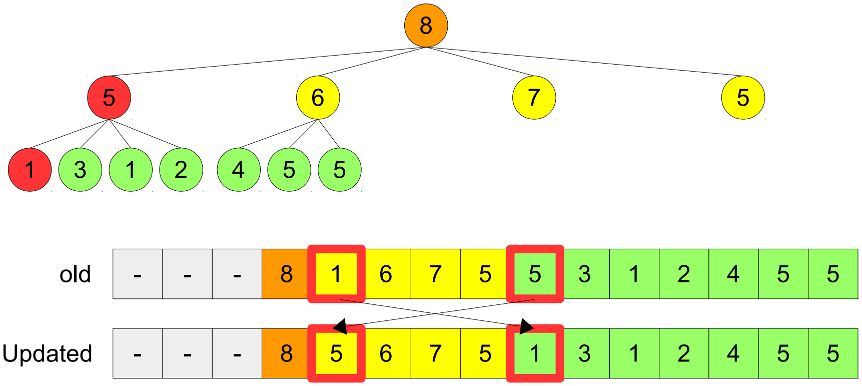


Figure: Compare, and swap if the max child node is larger than its parent node. Then no more child node, heap property reconstruction is done.

Update-key operation on the root node (step 6)

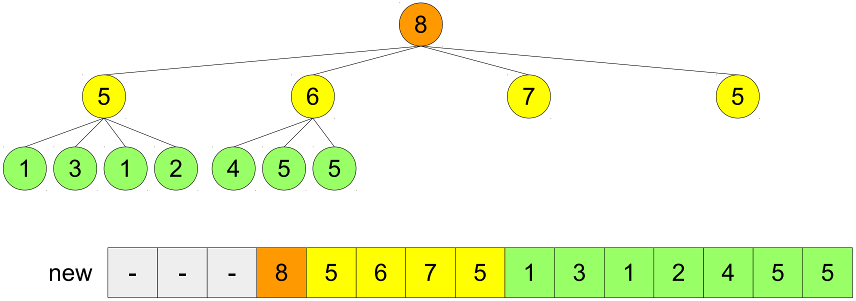
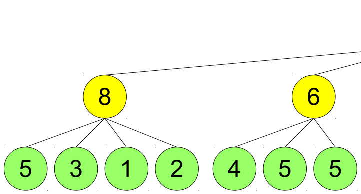


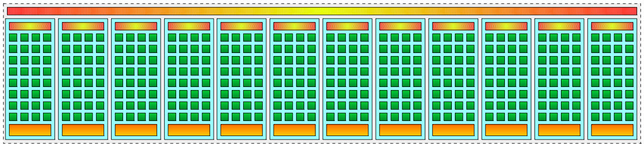
Figure: Final status.

Unroll the above *update-key* operation

- Step 1: update the root node
- Step 2: *find-maxchild*
- Step 3: *compare-and-swap*
- Step 4: *find-maxchild*
- Step 5: *compare-and-swap*
- Step 6: heap property satisfied, return

Running *d*-heaps on GPUs?

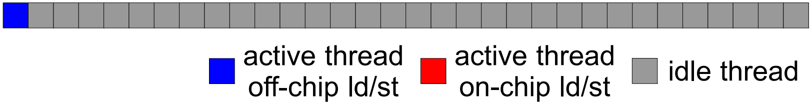




The above *update-key* operation on GPUs

Given a 32-heap running in a thread-block (or work-group) of size 32 threads (or work-items).

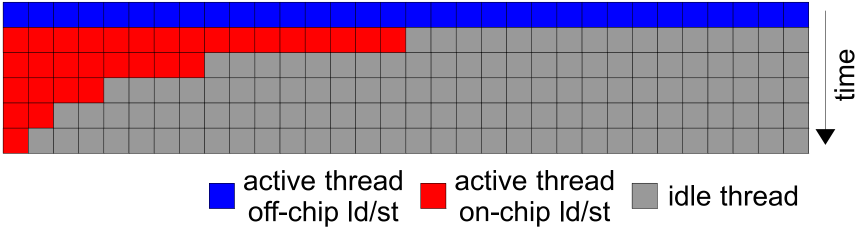
- Step 1: update the root node



The above *update-key* operation on GPUs

Given a 32-heap running in a thread-block (or work-group) of size 32 threads (or work-items).

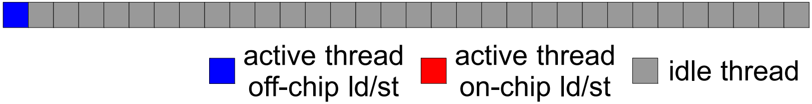
- Step 1: update the root node
- Step 2: *find-maxchild* (parallel reduction)



The above *update-key* operation on GPUs

Given a 32-heap running in a thread-block (or work-group) of size 32 threads (or work-items).

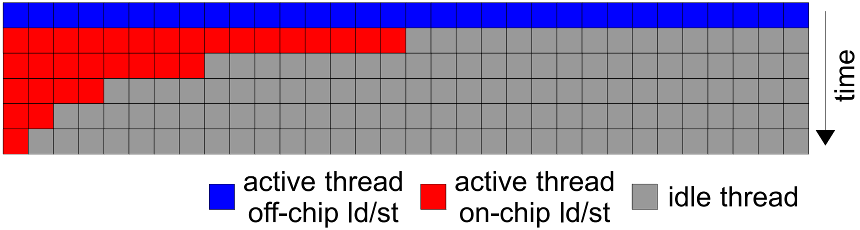
- Step 1: update the root node
- Step 2: *find-maxchild* (parallel reduction)
- Step 3: *compare-and-swap*



The above *update-key* operation on GPUs

Given a 32-heap running in a thread-block (or work-group) of size 32 threads (or work-items).

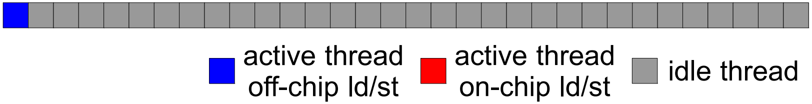
- Step 1: update the root node
- Step 2: *find-maxchild* (parallel reduction)
- Step 3: *compare-and-swap*
- Step 4: *find-maxchild* (parallel reduction)



The above *update-key* operation on GPUs

Given a 32-heap running in a thread-block (or work-group) of size 32 threads (or work-items).

- Step 1: update the root node
- Step 2: *find-maxchild* (parallel reduction)
- Step 3: *compare-and-swap*
- Step 4: *find-maxchild* (parallel reduction)
- Step 5: *compare-and-swap*



The above *update-key* operation on GPUs

Given a 32-heap running in a thread-block (or work-group) of size 32 threads (or work-items).

- Step 1: update the root node
- Step 2: *find-maxchild* (parallel reduction)
- Step 3: *compare-and-swap*
- Step 4: *find-maxchild* (parallel reduction)
- Step 5: *compare-and-swap*
- Step 6: heap property satisfied, return

Pros and Cons

Pros – why we want GPUs?

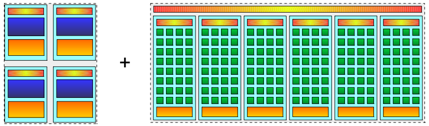
- Run much faster *find-maxchild* using parallel reduction
- Load continuous child nodes with few memory transactions (coalesced memory access)
- Shallow heap can accelerate *insert* operation

Cons – why we hate them?

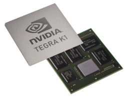
- Run slow *compare-and-swap* using only one single weak thread
- Other threads have to wait for a long time due to single-thread high-latency off-chip memory access

Emerging Asymmetric Multicore Processors (AMPs)

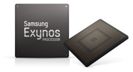
AMD APUs



CPU
GPU


Nvidia Tegra

Intel Sandy/Ivy Bridge, Haswell

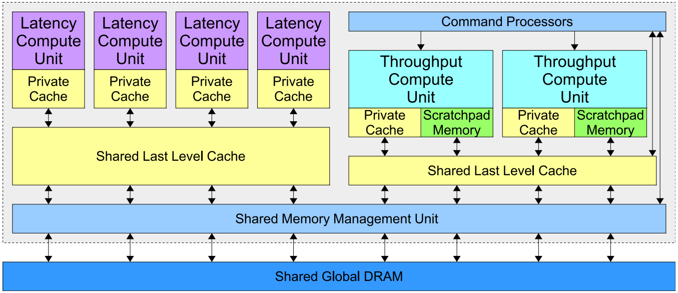

Samsung Exynos

Qualcomm Snapdragon

The block diagram of an AMP used in this work

The chip consists of four major parts:

- a group of Latency Compute Units (LCUs) with caches,
- a group of Throughput Compute Units (TCUs) with shared command processors, scratchpad memory and caches,
- a shared memory management unit, and
- a shared global DRAM



Heterogeneous System Architecture (HSA): a step forward

Main features in the current HSA design:

- the two types of compute units share unified memory address space
    - no data transfer through PCIe link
    - large pageable memory for the TCUs
    - much more efficient LCU-TCU interactions due to coherency
- fast LCU-TCU synchronization mechanism
    - user-mode queueing system
    - shared memory signal object
    - much lighter driver overhead

Leveraging the AMPs?

A direct way is to exploit task, data and pipeline parallelism in the two types of cores.

But, we still have two questions:

- Whether or not the AMPs can expose fine-grained parallelism in fundamental data structure and algorithm design?
- Can new designs outperform their conventional counterparts plus the coarse-grained (task, data and pipeline) parallelization?

ad-heap data structure

We propose *ad*-heap (*a*symmetric *d*-heap), a new heap data structure that can obtain performance benefits from both of the two types of cores.

The *ad*-heap data structure introduces a new component – a bridge structure, located in the originally empty head part of the *d*-heap. The bridge consists of one node counter and one sequence of size $2h$, where h is the height of the heap.

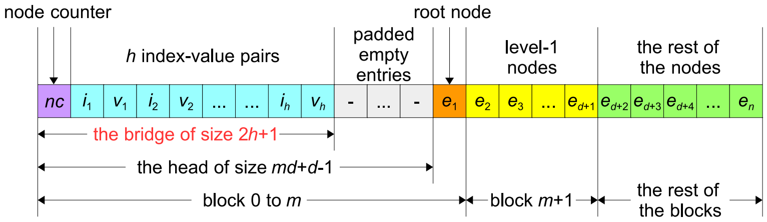


Figure: The layout of the *ad*-heap data structure.

Update-key operation on the root node (step 0)

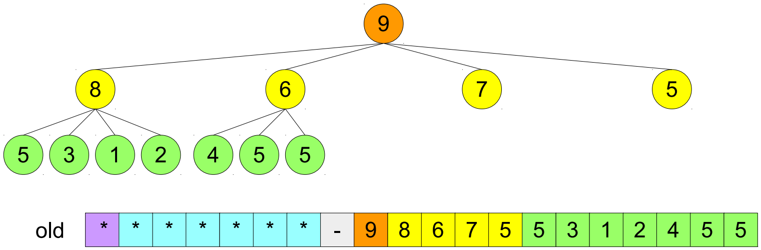


Figure: Initial status.

Update-key operation on the root node (step 1)

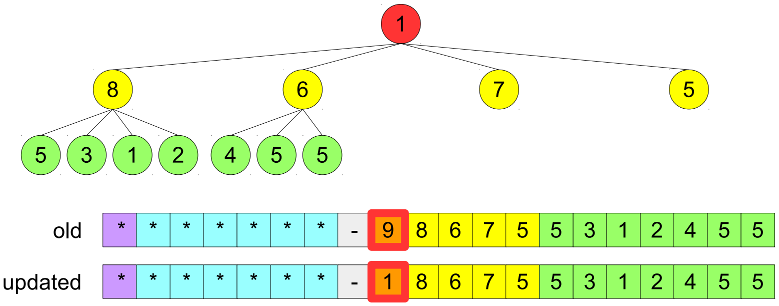


Figure: An LCU updates the value of the root node. Then the heap property on the level-1 and level-2 might be broken, so we issue an LCU $\rightarrow$ TCU call.

Update-key operation on the root node (step 2)

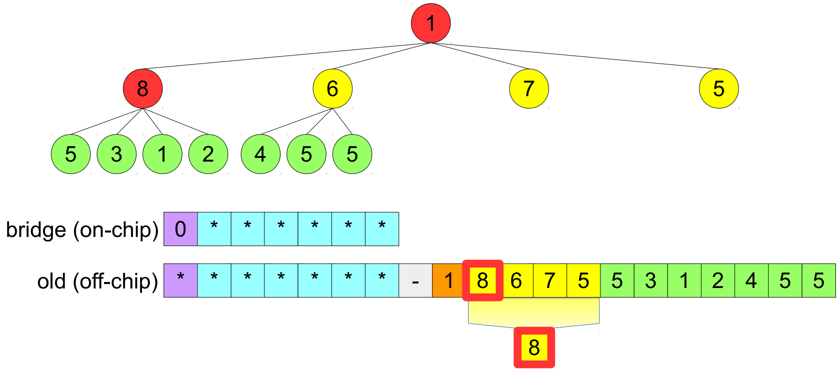


Figure: An invoked TCU initializes the bridge in its scratchpad memory, finds the maximum child node of the updated parent node.

Update-key operation on the root node (step 3)

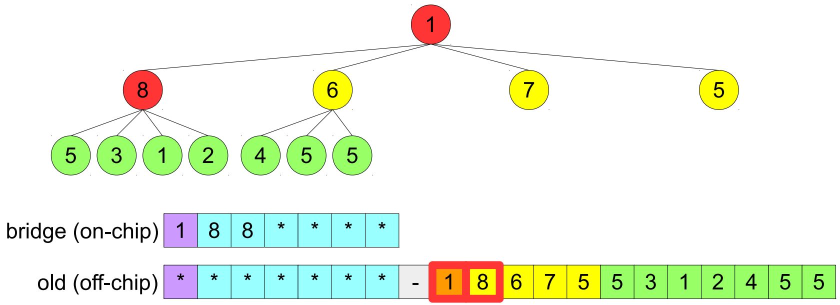


Figure: The TCU compares, and updates node counter, saves the parent node position and the max child node value to the on-chip bridge, if the max child node is larger than its parent node. Then the heap property on the level-2 and level-3 might be broken.

Update-key operation on the root node (step 4)

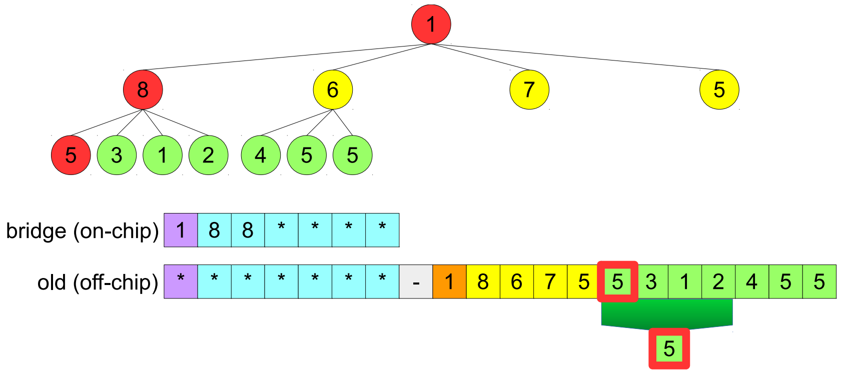


Figure: The TCU finds the maximum child node of the updated parent node.

Update-key operation on the root node (step 5)

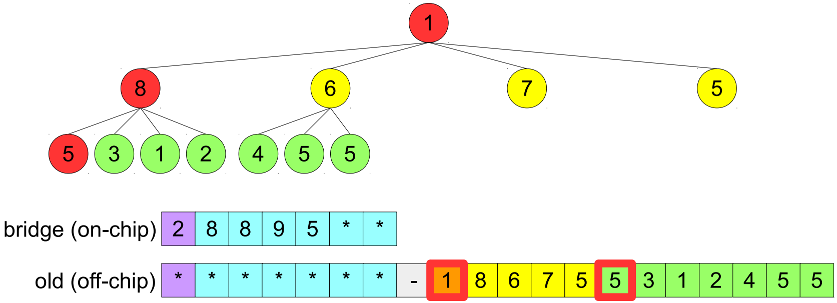


Figure: The TCU compares, and updates node counter, saves the parent node position and the max child node value to the bridge, if the max child node is larger than its parent node.

Update-key operation on the root node (step 6)

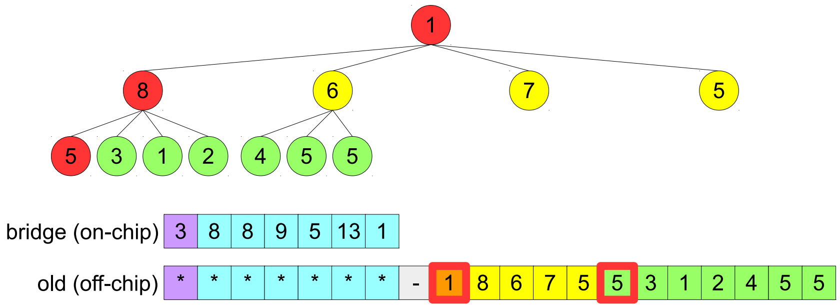


Figure: The TCU updates node counter and saves the child node position and the parent node value to the bridge, due to no more child node and the heap property reconstruction is done.

Update-key operation on the root node (step 7)

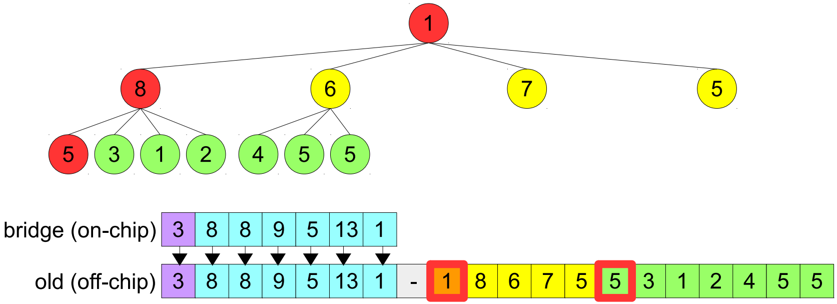


Figure: The TCU dumps the bridge from the scratchpad memory to the global memory. Then we issue an TCU → LCU call.

Update-key operation on the root node (step 8)

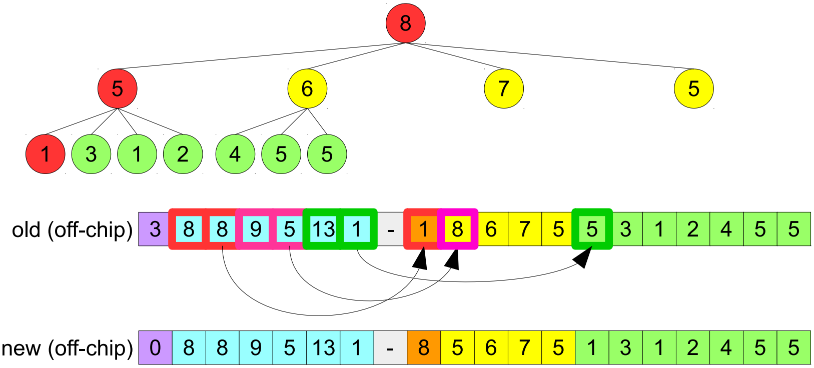


Figure: An invoked LCU reads each key-value pair, saves the value to its final position. Then all entries are up-to-date.

Update-key operation on the root node (step 9)

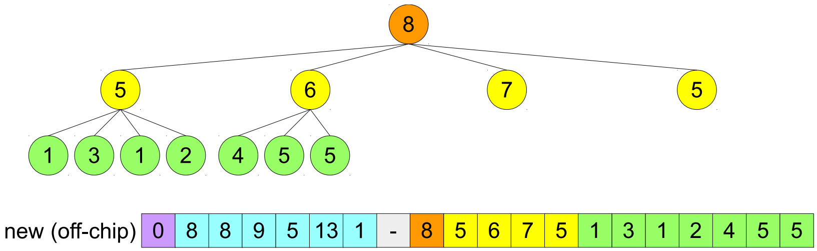


Figure: Final status.

Largely reduced TCU off-chip cost

Using the *ad*-heap, the number of the TCU off-chip memory access needs $hd/w + (2h+1)/w$ transactions, instead of $h(d/w+1)$ in the *d*-heap, where h is the heap height and w is warp size.

For example, given a 7-level 32-heap and set w to 32, the *d*-heap needs 14 off-chip memory transactions while the *ad*-heap only needs 8.

ad-heap simulator (before the HSA tools are ready)

The simulator pre-executes a *d*-heap based workload, counts the numbers of all kinds of operations, then run the same workload on real CPU-GPU systems.

The AMP queueing system is simulated by DKit C++ Library and Boost C++ Libraries.

Testbeds

System	Machine 1	Machine 2
CPU	AMD A6-1450 APU	Intel Core i7-3770
CPU cores	4 cores/1.0 GHz	4 cores/3.4 GHz
GPU	AMD Radeon HD 8250	nVidia GeForce GTX 680
GPU SIMD units	128 Radeon cores	1536 CUDA cores
ad-heap simulator	C++ and OpenCL	C++ and CUDA

Table: The Machines Used in Our Experiments

Benchmark and Datasets

Benchmark: a heap-based batch k-selection algorithm that finds the kth smallest entry from each of the sub-lists in parallel. One of its applications is batch kNN search in large-scale concurrent queries.

We set sizes of the list sets to 2^{25} and 2^{28} on the two machines, respectively, data type to 32-bit integer (randomly generated), size of each sub-list to the same length l (from 2^{11} to 2^{21}), and k to $0.1l$.

Performance results of the Machine 1

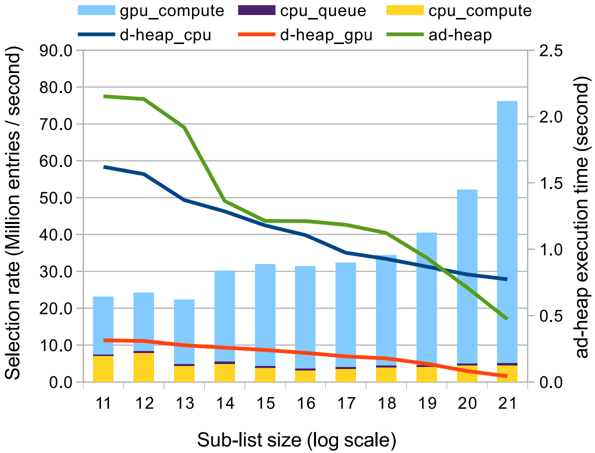


Figure: $d = 8$

Performance results of the Machine 1

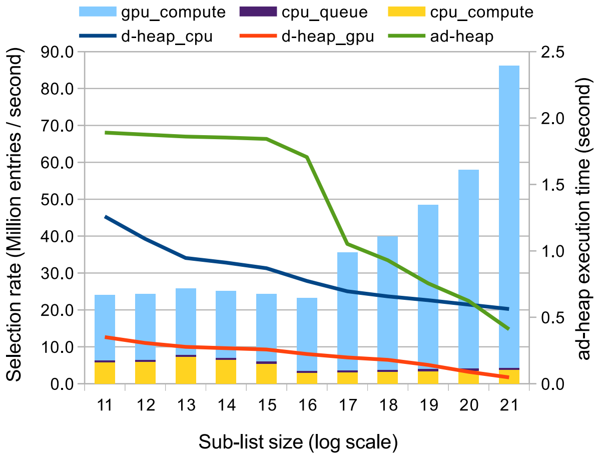


Figure: $d = 16$

Performance results of the Machine 1

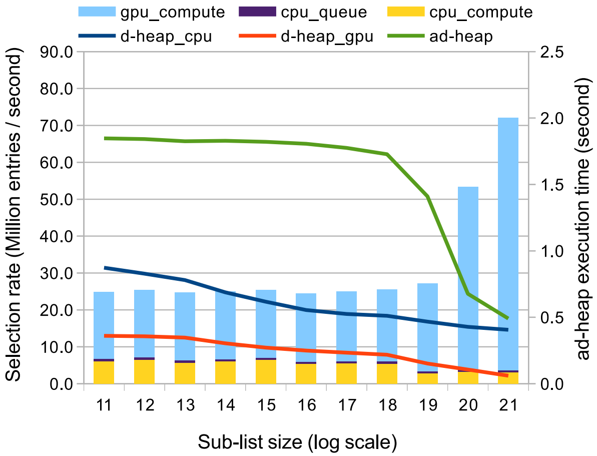


Figure: $d = 32$

Performance results of the Machine 1

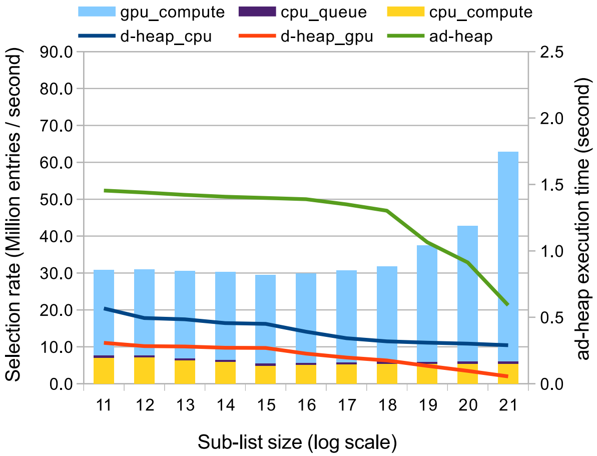


Figure: $d = 64$

Performance results of the Machine 1

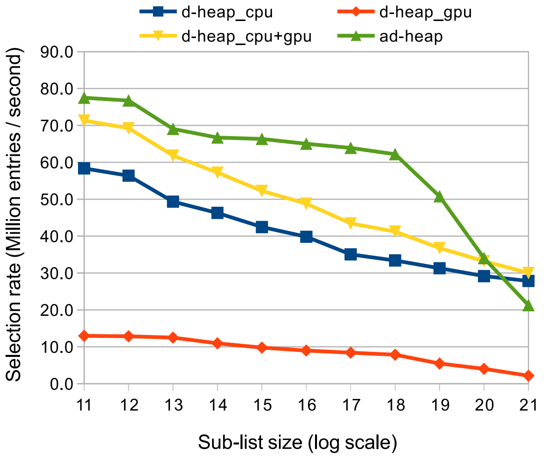


Figure: *aggregated results*

Performance results of the Machine 2

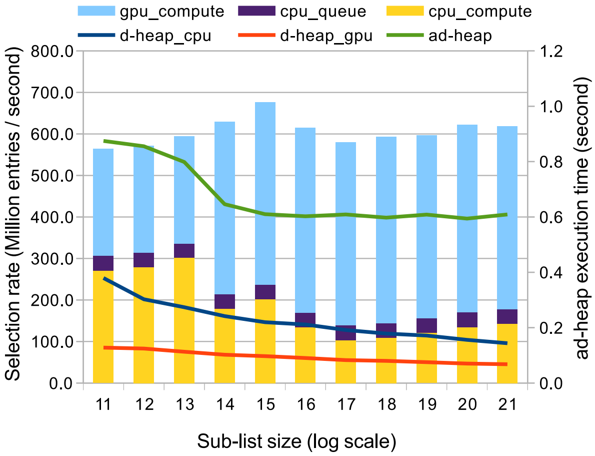


Figure: $d = 8$

Performance results of the Machine 2

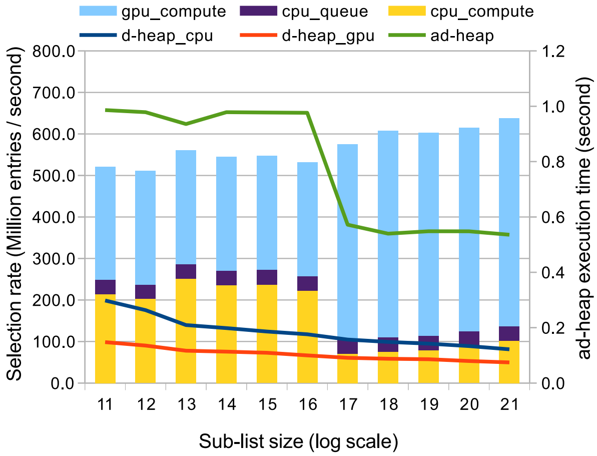


Figure: $d = 16$

Performance results of the Machine 2

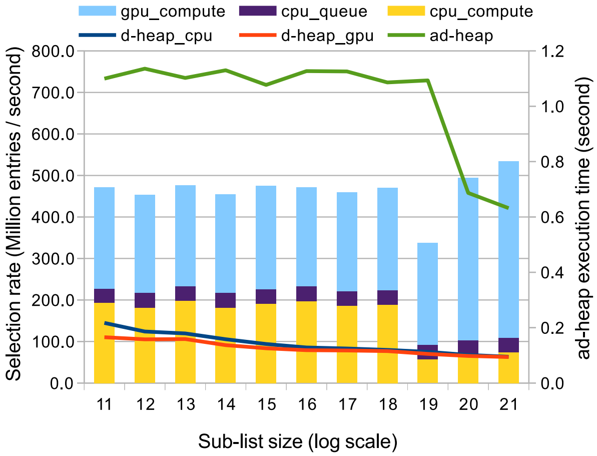


Figure: $d = 32$

Performance results of the Machine 2

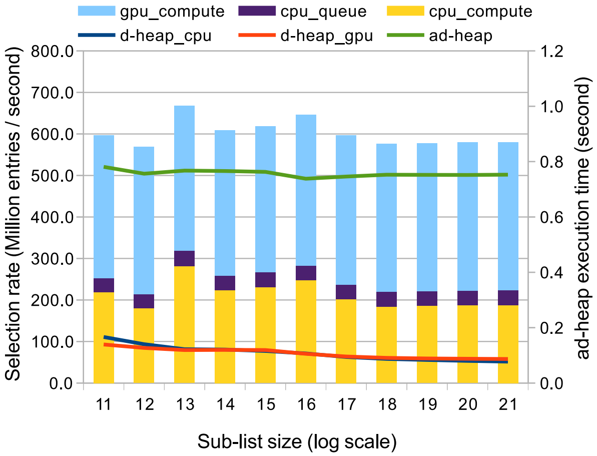


Figure: $d = 64$

Performance results of the Machine 2

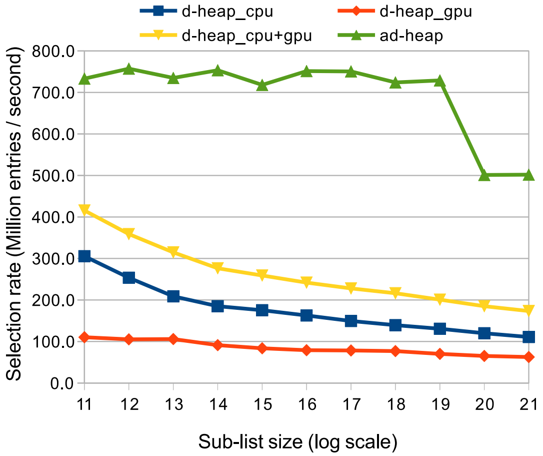


Figure: *aggregated results*

Conclusion

We proposed *ad-heap*, a new efficient heap data structure for the AMPs, and obtained up to 1.5x and 3.6x performance of the optimal scheduling method on two representative machines, respectively.

The performance numbers also showed that redesigning data structure and algorithm is necessary for exposing higher computational power of the AMPs.

We are looking forward to running *ad*-heap on real HSA programming tools but not simulators.

Thanks!

Questions?