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

Weifeng Liu, Brian Vinter
Niels Bohr Institute
University of Copenhagen
Copenhagen, Denmark
{weifeng, vinter}@nbi.dk

ABSTRACT

Heap is one of the most important fundamental data structures in computer science. Unfortunately, for a long time heaps did not obtain ideal performance gain from widely used throughput-oriented processors because of two reasons: (1) heap property decides that operations between any parent node and its child nodes must be executed sequentially, and (2) heaps, even d-heaps (d-ary heaps or d-way heaps), cannot supply enough wide data parallelism to these processors. Recent research proposed more versatile asymmetric multicore processors (AMPs) that consist of two types of cores (latency-oriented cores with high single-thread performance and throughput-oriented cores with wide vector processing capability), unified memory address space and faster synchronization mechanism among cores with different ISAs.

To leverage the AMPs for the heap data structure, in this paper we propose *ad*-heap, an efficient heap data structure that introduces an implicit bridge structure and properly apportions workloads to the two types of cores. We implement a batch k-selection algorithm and conduct experiments on simulated AMP environments composed of real CPUs and GPUs. In our experiments on two representative platforms, the *ad*-heap obtains up to 1.5x and 3.6x speedup over the optimal AMP scheduling method that executes the fastest d-heaps on the standalone CPUs and GPUs in parallel.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Lists, stacks, and queues; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*

General Terms

Algorithms, Performance

Keywords

Heaps, Priority queues, d-heaps, ad-heap, GPGPU, Asymmetric multicore processor, HSA

GPGPU-7, March 01 2014, Salt Lake City, UT, USA

ACM 978-1-4503-2766-4/14/03 ...$15.00
http://dx.doi.org/10.1145/2576779.2576786.

1. INTRODUCTION

Heap (or priority queue) data structures are heavily used in many algorithms such as k-nearest neighbor (kNN) search, finding the minimum spanning tree and the shortest path problems. Compared to the most basic binary heap, d-heaps [19, 38], in particular implicit d-heaps proposed by LaMarca and Ladner [23], have better practical performance on modern processors. However, as throughput-oriented processors (e.g. GPUs) bring higher and higher peak performance and bandwidth, heap data structures did not reap benefit from this trend because their very limited degree of data parallelism cannot saturate wide SIMD units.

Recently, more and more programs can obtain performance improvements from heterogeneous computing which combines multiple different symmetric multicore processors (e.g. CPUs and throughput-oriented accelerators) into one system. At the same time, asymmetric multicore processors (AMPs) were proposed and received a lot of attention. The AMPs normally consist of two types of cores (latency-oriented cores with high single-thread performance and throughput-oriented cores with wide vector processing capability) and unified memory address space with cache coherence. Compared with standalone symmetric multicore processors and loosely-coupled heterogeneous systems, the AMPs promised higher overall performance, energy efficiency and flexibility to broader applications with single-ISA [3, 21, 35] and multi-ISA [10] configurations. Those expected benefits come from three aspects: (1) the two types of cores can execute tasks of various characteristics in parallel, (2) unified memory address space saves the cost of memory copy or address mapping between separate address spaces, and (3) tightly-coupled design reduces context switching overhead.

To leverage the AMPs, previous research has concentrated on various coarse-grained methods that exploit task, data and pipeline parallelism in the AMPs. However, it is still an open question whether or not the new features of the emerging AMPs can expose fine-grained parallelism in fundamental data structure and algorithm design. And can new designs outperform their conventional counterparts plus the coarse-grained parallelization is a further question.

In this paper, we propose a new heap data structure called *ad*-heap (*a*symmetric d-heap). The *ad*-heap introduces an implicit bridge structure — a new component that records deferred random memory transactions and makes the two types of cores in the AMPs focus on their most efficient memory behaviors. Thus overall bandwidth utilization and instruction throughput can be significantly improved.

We evaluate performance of the *ad*-heap by using a batch k-selection algorithm on two simulated AMP platforms composed of real CPUs and GPUs. The experimental results show that compared with the optimal AMP scheduling method that executes the fastest d-heaps on the standalone CPUs and GPUs in parallel, the *ad*-heap achieves up to 1.5x and 3.6x speedup on the two platforms, respectively.

2. PRELIMINARIES

2.1 Implicit *d*-heaps

Given a heap of size n, where $n \neq 0$, a d-heap data structure [19, 38] lets each parent node has d child nodes, where $d > 2$ normally. To satisify cache-line alignment and reduce cache miss rate, the whole heap can be stored in an implicit space of size $n + d - 1$, where the extra $d - 1$ entries are padded in front of the root node and kept empty [23]. Here we call the padded space "head" of the heap. Figure 1 shows an example of the implicit max-d-heaps while $n = 12$ and $d = 4$. Notice that each group of the child nodes starts from an aligned cache block.

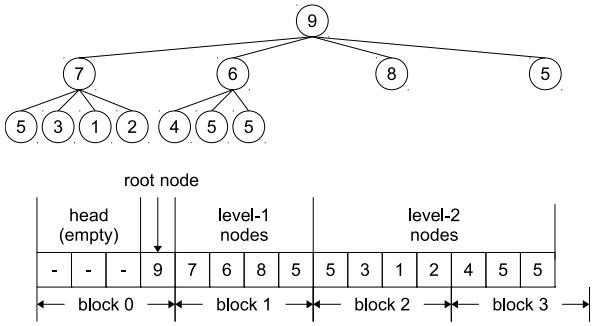


Figure 1: The layout of a 4-heap of size 12.

Because of the padded head, each node has to add an $offset = d - 1$ to its index in the implicit array. Given a node of index i, its array index becomes $i + offset$. Its parent node's (if $i \neq 0$) array index is $\lfloor (i-1)/d \rfloor + offset$. If any, its first child node is located in $di + 1 + offset$ and the last child node is in array index $di + d + offset$.

Given an established non-empty max-d-heap, we can execute three typical heap operations:

- *insert* operation 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* operation copies the last node to the position of the root node, decreases the heap size to $n-1$, and takes $O(d \log_d n)$ worst-case time to reconstruct the heap property, and
- *update-key* operation updates a node, keeps the heap size unchanged, and takes $O(d \log_d n)$ worst-case time (if the root node is updated) to reconstruct the heap property.

The above heap operations depend on two more basic operations:

- *find-maxchild* operation takes $O(d)$ time to find the maximum child node for a given parent node, and
- *compare-and-swap* operation takes constant time to compare values of a child node and its parent node, then swap their values if the child node is larger.

2.2 Asymmetric Multicore Processors

Compared to symmetric multicore processors, the AMPs offer more flexibilities in architecture design space, thus many AMP architectures have been proposed. To leverage mature CPU and GPU architectures, we use a CPU-GPU integrated AMP model for evaluating the *ad*-heap proposed in this paper. Representatives of this model include AMD Accelerated Processing Units (APUs) [8, 2], Intel Ivy Bridge multi-CPU and GPU system-on-chip [13], Echelon heterogeneous GPU architecture [20] proposed by nVidia, and many mobile processors (e.g. nVidia Tegra [28], Qualcomm Snapdragon [30] and Samsung Exynos [32]).

Figure 2 shows a block diagram of the AMP chip used in this paper. The chip consists of four major parts: (1) a group of Latency Compute Units (LCUs) with hardware-controlled caches, (2) a group of Throughput Compute Units (TCUs) with shared command processors, software-controlled scratchpad memory and hardware-controlled caches, (3) shared memory management unit, and (4) shared global DRAM. For simplicity, only four LCUs and two TCUs are drawn in the Figure 2.

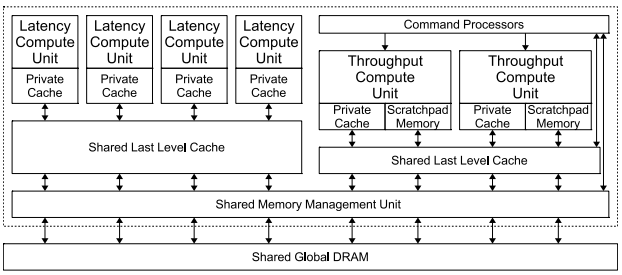


Figure 2: The block diagram of an AMP.

The LCUs can be seen as CPU cores that have higher single-thread performance due to out-of-order execution, branch prediction and large amounts of caches. The TCUs can be seen as GPU cores that execute massively parallel lightweight threads on SIMD units for higher aggregate throughput. The two types of compute units have completely different ISAs and separate cache sub-systems. Each compute unit has its own set of instruction issue units, while all TCUs share one set of command processors.

Compared to the loosely-coupled CPU-GPU heterogeneous systems, the emerging CPU-GPU integrated AMPs make expected differences in hardware architecture and programming model.

From the perspective of the AMP hardware, the two types of compute units share single unified address space instead of using separate address spaces (i.e. system memory space and GPU device memory space). The benefits include avoiding data transfer through connection interfaces (e.g. PCIe link) and letting TCUs access more memory by paging memory to and from disk. Further, the consistent pageable shared virtual memory can be fully or partially coherent, meaning that much more efficient LCU-TCU interactions are possible due to eliminated heavyweight synchronization (i.e. flushing and GPU cache invalidation).

From the perspective of the programming model, synchronization mechanism among compute units is redefined. Recently, several CPU-GPU fast synchronization approaches [9, 22, 25] have been proposed. In this paper, we implement the *ad*-heap operations through the synchronization mechanism designed by the HSA (Heterogeneous System Architecture) Foundation. According to the current HSA design [22], each compute unit executes its task and sends a signal object of size 64 Byte to a low-latency shared memory queue when it has completed the task. Thus with HSA, LCUs and TCUs can queue tasks to each other and to themselves. Further, the communications can be dispatched in the user mode of the operating systems, thus the traditional "GPU kernel launch" method (through the operating system kernel services and the GPU drivers) is avoided and the LCU-TCU communication latency is significantly reduced. Figure 3 shows an example of the shared memory queue.

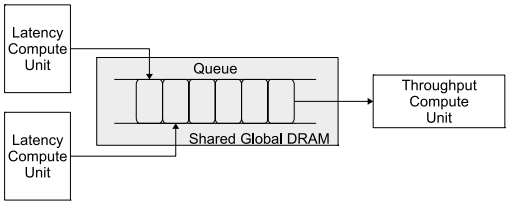


Figure 3: A shared memory queue.

3. *AD*-HEAP DESIGN

3.1 Performance Considerations

We first conduct analysis on the degree of parallelism of the d-heap operations. We can see that the *insert* operation does not have any data parallelism because the heap property is reconstructed in a bottom-up order and only the unparallelizable *compare-and-swap* operations are required. On the other hand, the *delete-max* operation reconstructs the heap property in a top-down order that does not have any data parallelism either, but executes multiple ($\log_d n$ in the worst case) lower-level parallelizable *find-maxchild* operations. For the *update-key* operation, the position and the new value of the key decides whether the bottom-up or the top-down order is executed in the heap property reconstruction. Therefore, in this paper we mainly consider accelerating the heap property reconstruction in the top-down order. After all, the *insert* operation can be efficiently executed in serial because the heap should be very shallow if the d is large.

Without loss of generality, we focus on an *update-key* operation that updates the root node of a non-empty max-d-heap. To reconstruct the heap property in the top-down order, the *update-key* operation alternately executes the *find-maxchild* operations and the *compare-and-swap* operations until the heap property is satisfied or the last changed parent node does not have any child node. Notice that the *swap* operation can be simplified because the child node does not need to be updated in the procedure. Actually its value can be kept in thread register and be reused until the final round. Algorithms 1 and 2 show pseudo codes of the *update-key* operation and the *find-maxchild* operation, respectively.

Imagine the whole operation is executed on a wide SIMD processor (e.g. GPU), the *find-maxchild* operation can be efficiently accelerated by the SIMD units through a *streaming reduction* scheme within much faster $O(\log d)$ time instead of original $O(d)$ time. And because of wider memory controllers, one group of w continuous SIMD threads (a warp in the nVidia GPUs or a wavefront in the AMD GPUs) can load w aligned continuous entries from the off-chip memory to the on-chip scratchpad memory (the shared memory in the CUDA terminology or the local memory in the OpenCL terminology) by one off-chip memory transaction (coalesced memory access). Thus to load d child nodes from the off-chip memory, only d/w memory transactions are required.

A similar idea has been implemented on the CPU vector units. Furtak et al. [15] accelerated d-heap *find-maxchild* operations by utilizing x86 SSE instructions. The results showed 15% - 31% execution time reduction, on average, in a mixed benchmark composed of the *delete-max* operations and *insert* operations while $d = 8$ or 16. However, the vector units in the CPU cannot supply as much SIMD processing capability as in the GPU. Further, according to the previous research [4], moving vector operations from the CPU to the integrated GPU can obtain both performance improvement and energy efficiency. Therefore, in this paper we focus on utilizing GPU-style vector processing but not SSE/AVX instructions.

However, other operations, in particular the *compare-and-*

Algorithm 1 Update the root node of a non-empty max-d-heap.

```
1: function UPDATE-KEY(*heap, d, n, newv)
2:     offset ← d − 1                          ▷ offset of the implicit storage
3:     i ← 0                                   ▷ the root node index
4:     v ← newv                                ▷ the root node value
5:     while di + 1 < n do                     ▷ if the first child is existed
6:         ⟨maxi, maxv⟩ ← FIND-MAXCHILD(*heap, d, n, i)
7:         if maxv > v then                    ▷ compare
8:             heap[i + offset] ← maxv         ▷ swap
9:             i ← maxi
10:        else                                ▷ the heap property is satisfied
11:            break
12:        end if
13:    end while
14:    heap[i + offset] ← v
15:    return
16: end function
```

Algorithm 2 Find the maximum child node of a given parent node.

```
1: function FIND-MAXCHILD(*heap, d, n, i)
2:     offset ← d − 1
3:     starti ← di + 1                         ▷ the first child index
4:     stopi ← MIN(n − 1, di + d)              ▷ the last child index
5:     maxi ← starti
6:     maxv ← heap[maxi + offset]
7:     for i = starti + 1 to stopi do
8:         if heap[i + offset] > maxv then
9:             maxi ← i
10:            maxv ← heap[maxi + offset]
11:        end if
12:    end for
13:    return ⟨maxi, maxv⟩
14: end function
```

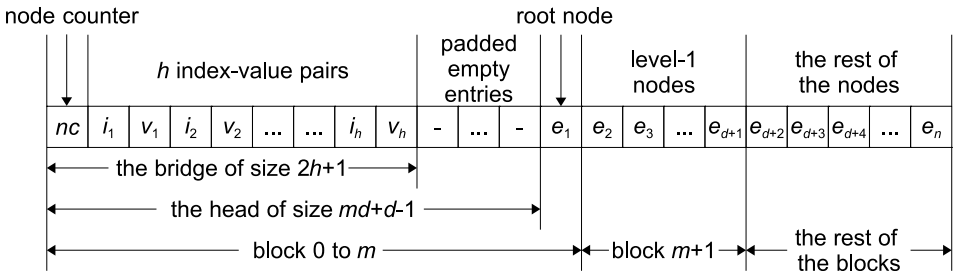


Figure 4: The layout of the *ad*-heap data structure.

swap operations, cannot obtain benefit from the SIMD units because they only need one single thread, which is far from saturating the SIMD units. And the off-chip memory bandwidth is also wasted because one expensive off-chip memory transaction only stores one entry (lines 8 and 14 in the Algorithm 1). Further, the rest threads are waiting for the single thread's time-consuming off-chip transaction to finish. Even though the single thread store has a chance to trigger a cache write hit, the very limited cache in the throughput-oriented processors can easily be polluted by the massively concurrent threads. Thus the single thread task should always be avoided.

Therefore, to maximize the performance of the d-heap operations, we consider two design objectives: (1) maximizing throughput of the large amount of the SIMD units for faster *find-maxchild* operations, and (2) minimizing negative impact from the single-thread *compare-and-swap* operations.

3.2 *ad*-heap Data Structure

Because the TCUs are designed for the wide SIMD operations and the LCUs are good at high performance single-thread tasks, the AMPs have a chance to become ideal platforms for operations with different characteristics of parallelism. 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 in the AMPs.

Compared to the d-heaps, the ad-heap data structure introduces an important new component — an implicit bridge structure. The bridge structure is located in the originally empty head part of the implicit d-heap. It consists of one node counter and one sequence of size $2h$, where h is the height of the heap. The sequence stores the index-value pairs of the nodes to be updated in different levels of the heap, thus at most h nodes are required. If the space requirement of the bridge is larger than the original head part of size $d-1$, the head part can be easily extended to $md+d-1$ to guarantee that each group of the child nodes starts from an aligned cache block, where m is a natural number and equal to $\lceil 2(h+1)/d \rceil - 1$. Figure 4 shows the layout of the ad-heap data structure.

3.3 *ad*-heap Operations

The corresponding operations of the ad-heap data structure are redesigned as well. Again, for simplicity and without loss of generality, we only consider the *update-key* operation described in the sub-section 3.1.

Before the *update-key* operation starts, the bridge is constructed in the on-chip scratchpad memory of a TCU and the node counter is initialized to zero. Then in each iteration (lines 6–12 of the Algorithm 1), a group of lightweight SIMD threads in the TCU simultaneously execute the *find-maxchild* operation (i.e. in parallel load at most d child nodes to the scratchpad memory and run the *streaming reduction* scheme to find the index and the value of the maximum child node). After each *find-maxchild* and *compare* operation, if a *swap* operation is needed, one of the SIMD threads adds a new index-value pair (index of the current parent node and value of the maximum child node) to the bridge and updates the node counter. If the current level is not the last level, the new value of the child node can be stored in a register and be reused as the parent node of the next level. Otherwise, the single SIMD thread stores the new indices and values of both of the parent node and the child node to the bridge. Because the on-chip scratchpad memory is normally two orders of magnitude faster than the off-chip memory, the cost of the single-thread operations is negligible. When all iterations are finished, at most $2h+1$ SIMD threads store the bridge from the on-chip scratchpad memory to the continuous off-chip memory by $\lceil (2h+1)/w \rceil$ off-chip memory transactions. The single program multiple data (SPMD) pseudo code is shown in Algorithm 3. Because the *streaming reduction* is a widely used building block, here we do not give out parallel pseudo code of the *find-maxchild* operation. After the bridge is dumped, a signal object is transferred to the TCU-LCU queue.

Triggered by the synchronization signal from the queue, one of the LCUs sequentially loads the entries from the bridge and stores them to the real heap space in linear time. Notice that no data transfer, address mapping or explicit coherence maintaining is required due to the unified memory space with cache coherence. And because the entries in the bridge are located in continuous memory space, the LCU cache system can be efficiently utilized. When all entries are updated, the whole *update-key* operation is completed. The pseudo code of the LCU workload in the *update-key* operation is shown in Algorithm 4.

Refer to the command queue in the OpenCL specification and the architected queueing language (AQL) in the HSA design, we list the pseudo code of the *update-key* operation in Algorithm 5. Notice that the main difference between the current OpenCL-style queue and the emerging HSA-style queue is that the former is always triggered by an LCU and the latter can be triggered by an LCU or a TCU with very low communication cost.

We can see that although the overall time complexity is not reduced, the two types of compute units more focus on the off-chip memory behaviors that they are good at. We can calculate that the number of the TCU off-chip memory access needs $hd/w + (2h+1)/w$ transactions instead of

Algorithm 3 The SPMD TCU workload in the *update-key* operation of the *ad*-heap.

```
1: function TCU-WORKLOAD(*heap, d, n, h, newv)
2:     tid ← GET-THREAD-LOCALID()
3:     i ← 0
4:     v ← newv
5:     *bridge ← SCRATCHPAD-MALLOC(2h + 1)
6:     if tid = 0 then
7:         bridge[0] ← 0                ▷ initialize the node counter
8:     end if
9:     while di + 1 < n do
10:        ⟨maxi, maxv⟩ ← FIND-MAXCHILD(*heap, d, n, i)
11:        if maxv > v then
12:            if tid = 0 then          ▷ insert a index-value pair
13:                bridge[2 * bridge[0] + 1] ← i
14:                bridge[2 * bridge[0] + 2] ← maxv
15:                bridge[0] ← bridge[0] + 1
16:            end if
17:            i ← maxi
18:        else
19:            break
20:        end if
21:    end while
22:    if tid = 0 then                  ▷ insert the last index-value pair
23:        bridge[2 * bridge[0] + 1] ← i
24:        bridge[2 * bridge[0] + 2] ← v
25:        bridge[0] ← bridge[0] + 1
26:    end if
27:    if tid < 2h + 1 then             ▷ dump the bridge to off-chip
28:        heap[tid] ← bridge[tid]
29:    end if
30:    return
31: end function
```

$h(d/w + 1)$ in the d-heap. 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. Since the cost of the off-chip memory access dominates execution time, the practical TCU performance can be improved significantly. Further, from the LCU perspective, all read transactions are from the bridge in continuous cache blocks and all write transactions only trigger non-time-critical cache write misses to random positions. Therefore the LCU workload performance can also be expected to be good.

3.4 *ad*-heap Simulator

Because the HSA programming tools for the AMP hardware described in this paper are not currently available yet, we conduct experiments on simulated AMP platforms composed of real standalone CPUs and GPUs. The *ad*-heap simulator has two stages:

(1) **Pre-execution stage**. For a given input list and a size d, we first count the number of the *update-key* operations and the numbers of the subsequent *find-maxchild* and *compare-and-swap* operations by pre-executing the work through the d-heap on the CPU. We write N_u, N_f, N_c and N_s to denote the numbers of the *update-key* operations, *find-maxchild* operations, *compare* operations and *swap* operations, respectively. Although the N_f and the N_c are numerically equivalent, we use two variables for the sake of clarity.

(2) **Simulation stage**. Then we execute exactly the same amount of work with the *ad*-heap on the CPU and the GPU.

Algorithm 4 The LCU workload in the *update-key* operation of the *ad*-heap.

```
1: function LCU-WORKLOAD(*heap, d, n, h)
2:     m ← ⌈2(h + 1)/d⌉ − 1
3:     offset ← md + d − 1
4:     *bridge ← *heap
5:     for i = 0 to bridge[0] − 1 do
6:         index ← bridge[2 * i + 1]
7:         value ← bridge[2 * i + 2]
8:         heap[index + offset] ← value
9:     end for
10:    return
11: end function
```

Algorithm 5 The control process of the *update-key* operation.

```
1: function UPDATE-KEY(*heap, d, n, h, newv)
2:     QLtoT ← CREATE-QUEUE()
3:     QTtoL ← CREATE-QUEUE()
4:     Tpkt ← TCU-WORKLOAD(*heap, d, n, h, newv)
5:     Lpkt ← LCU-WORKLOAD(*heap, d, n, h)
6:     QUEUE_DISPATCH_FROM_LCU(QLtoT, Tpkt)
7:     QUEUE_DISPATCH_FROM_TCU(Tpkt, QTtoL, Lpkt)
8:     return
9: end function
```

The work can be split into three parts:

- The CPU part reads the entries in N_u bridges (back from the GPU) and writes $N_u(N_s + 1)$ values to the corresponding entry indices. This part takes T_{cc} time on the CPU.

- To simulate the LCU-TCU communication mechanism in the HSA design, the CPU part also need to execute signal object sends and receives. We use a lockless multi-producer single-consumer (MPSC) queue programming tool in the DKit C++ Library [6] (based on multithread components in the Boost C++ Libraries [1]) for simulating the AMP queueing system. To meet the HSA standard [18], our packet size is set to 64 Byte with two 4 Byte flags and seven 8 Byte flags. Further, packing and unpacking time is also included. Because each GPU core (and also each TCU) needs to execute multiple thread groups (thread blocks in the CUDA terminology or work groups in the OpenCL terminology) in parallel for memory latency hiding, we use 16 as a factor for the combined thread groups. Therefore, $2N_u/16$ push/pop operation pairs are executed for N_u LCU to TCU communications and the same amount of TCU to LCU communications. We record this time as T_{cq}.

- The GPU part executes N_f *find-maxchild* operations and N_c *compare* operations and writes N_u bridges from the on-chip scratchpad memory to the off-chip global shared memory. This part takes T_{gc} time on the GPU.

After simulation runs, we use overlapped work time on the CPU and the GPU as execution time of the *ad*-heap since the two types of cores are able to work in parallel. Thus the final execution time is the longer one of $T_{cc} + T_{cq}$ and T_{gc}.

Table 1: The Machines Used in Our Experiments

System	Machine 1	Machine 2
CPU	AMD A6-1450 APU	Intel Core i7-3770
CPU cores/clock rate/architecture	4 cores/1.0 GHz/Jaguar	4 cores/3.4 GHz/Ivy Bridge
CPU peak single precision throughput	32 GFLOPS	217.6 GFLOPS
CPU max thermal design power	8 W (shared)	77 W
System memory/channels/bandwidth	3.4 GB DDR3L-1066/1/8.5 GB/s (shared)	32 GB DDR3-1600/2/25.6 GB/s
GPU	AMD Radeon HD 8250 (intergrated)	nVidia GeForce GTX 680
GPU execution units/architecture	2 compute units/Graphics Core Next	8 multiprocessors/Kepler
GPU vector units/clock rate	128 Radeon cores/400 MHz	1536 CUDA cores/1006 MHz
GPU peak single precision throughput	102.4 GFLOPS	3090.4 GFLOPS
GPU scratchpad memory	128 KB (64 KB per compute unit)	384 KB (48 KB per multiprocessor)
GPU memory/bandwidth	0.6 GB DDR3L-1066/8.5 GB/s (shared)	2 GB GDDR5/192.2 GB/s
GPU max thermal design power	8 W (shared)	250 W
GPU driver version	13.11 Beta	304.116
Operating system	Ubuntu Linux 12.04	Ubuntu Linux 12.04
Compiler and library	g++ 4.6.3 and OpenCL 1.2	g++ 4.6.3 and CUDA 5.0
ad-heap simulator implementation	C++ and OpenCL	C++ and CUDA

Because of the features of the AMPs, costs of device/host memory copy and GPU kernel launch are not included in our timer. Notice that because we use both the CPU and the GPU separately, the simulated AMP platform is assumed to have accumulated off-chip memory bandwidths of the both processors. Moreover, we also assume that the GPU supports the device fission function defined in the OpenCL 1.2 specification and cores in the current GPU devices can be used as sub-devices which are more like the TCUs in the HSA design. Thus one CPU core and one GPU core can cooperate to deal with one *ad*-heap. The simulator is programmed in C++ and CUDA/OpenCL.

4. PERFORMANCE EVALUATION

4.1 Testbeds

To benchmark the performance of the d-heaps and the *ad*-heap, we use two representative machines: (1) a laptop system with an AMD A6-1450 APU, and (2) a desktop system with an Intel Core i7-3770 CPU and an nVidia GeForce GTX 680 discrete GPU. Detailed specifications are shown in Table 1.

4.2 Benchmark and Datasets

We use a heap-based batch k-selection algorithm as benchmark of the heap operations. Given a list set consists of a group of unordered sub-lists, the algorithm 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. In each sub-list, a max-heap of size k is constructed on the first k entries and its root node is compared with the rest of the entries in the sub-list. If a new entry is smaller, an *update-key* operation (i.e. the root node update and the heap property reconstruction) is triggered. After traversing all entries, the root node is the kth smallest entry and the heap contains the k smallest entries of the input sub-list.

In our *ad*-heap implementation, we execute heapify function (i.e. the first construction of the heap) on the GPU and the root node comparison operations (i.e. to decide whether an *update-key* operation is required) on the CPU. Besides the execution time described in the *ad*-heap simulator, the execution time of the above two operations are recorded in our timer as well.

According to capacity limitation of the GPU device memory, we set sizes of the list sets to 2^{28} and 2^{25} 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 $0.1l$.

4.3 Experimental Results

Primary Y-axis-aligned line graphs in Figures 5(a)–(e) and 6(a)–(e) show the selection rates of the d-heaps (on the CPUs and the GPUs) and the *ad*-heap (on the simulators) over the different sizes of the sub-lists and d values on the machine 1 and the machine 2, respectively. In all tests, all cores of the CPUs are utilized. We can see that for the performance of the d-heaps in all groups, the multicore CPUs are almost always faster than the GPUs, even when the larger d values significantly reduce throughputs of the CPUs. Thus, for the conventional d-heap data structure, the CPUs are still better choices in the heap-based k-selection problem. For the *ad*-heap, the fastest size d is always 32. On one hand, the smaller d values cannot fully utilize computation and bandwidth resources of the GPUs. On the other hand, the larger d values lead to much more data loading but do not bring the same order of magnitude shallower heaps.

Secondary Y-axis-aligned stacked columns in Figures 5(a)–(e) and 6(a)–(e) show the execution time of the three parts (CPU compute, CPU queue and GPU compute) of the *ad*-heap simulators. On the machine 1, the execution time of the GPU compute is always longer than the total time of the CPU work, because the raw performance of the integrated GPU is relatively too low to accelerate the *find-maxchild* operations and the memory sub-system in the APU is not completely designed for the GPU memory behaviors. On the machine 2, the ratio of CPU time and GPU time is much more balanced (in particular, while $d = 32$) due to the much stronger discrete GPU.

Figures 5(f) and 6(f) show aggregated performance numbers include the best results in the former 5 groups and the optimal scheduling method that runs the fastest d-heaps on the CPUs and the GPUs in parallel, respectively. In these two sub-figures, we can see that the *ad*-heap obtains up to

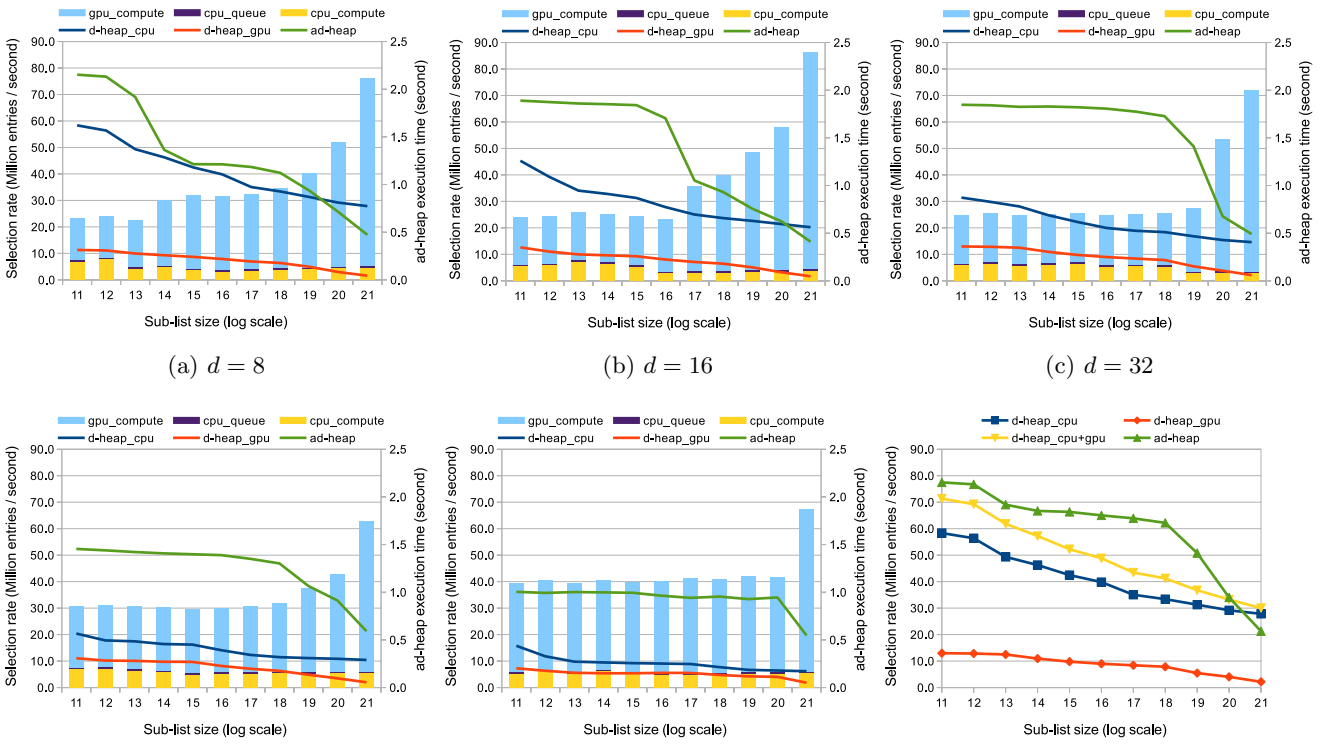


Figure 5: Selection rates and *ad*-heap execution time over different sizes of the sub-lists on the machine 1. The line-shape data series is aligned to the primary Y-axis. The stacked column-shape data series is aligned to the secondary Y-axis.

1.5x and 3.6x speedup over the optimal scheduling method when the d value is equal to 32 and the sub-list size is equal to 2^{18} and 2^{19}, respectively. Notice that the optimal scheduling method is also assumed to utilize accumulated off-chip memory bandwidths of the both processors.

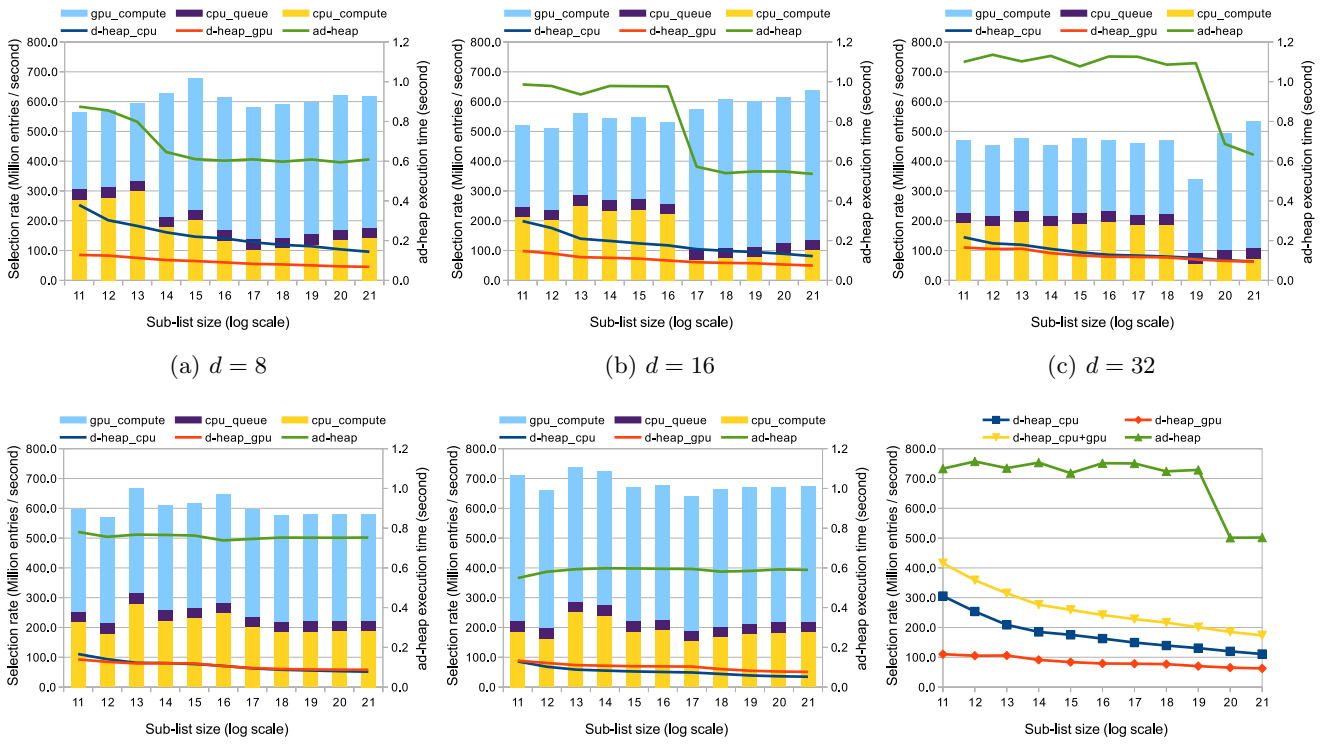
We can see that among all the candidates, only the *ad*-heap maintains relatively good performance stabilities while problem size grows. The performance numbers support our *ad*-heap design that gets benefits from main features of the two types of cores while the CPU d-heaps suffer with wider *find-maxchild* operations and the GPU d-heaps suffer with more single-thread *compare-and-swap* operations.

5. RELATED WORK

To the best of our knowledge, the *ad*-heap described in this paper is the first fundamental data structure that obtained good performance from fine-grained frequent interactions in the emerging AMPs. In contrast, the prior work has concentrated on exploiting coarse-grained parallelism or one-side computation in the AMPs. The current literature can be classified into four groups: (1) eliminating data transfer, (2) decomposing tasks and data, (3) pipelining, and (4) prefetching data.

Eliminating data transfer over PCIe bus is one of the most distinct advantages brought by the AMPs, thus its influence on performance and energy consumption has been relatively well studied. Research [11, 31, 37, 26] reported that various benchmarks can obtain performance improvements from the AMD APUs because of reduced data movement cost. Besides the performance benefits, research [34, 27] demonstrated that non-negligible power savings can be achieved by running programs on the APUs rather than the discrete GPUs because of shorter data path and the elimination of the PCIe bus and controller. Further, Daga and Nutter [12] showed that using the much larger system memory makes searches on very large B+ tree possible. Compared with the prior work, our *ad*-heap not only takes advantage of reduced data movement cost but also utilizes computational power of the both types of cores.

Decomposing tasks and data is also widely studied in heterogeneous system research. Research [21, 36] proposed scheduling approaches that map workloads onto the most appropriate core types in the single-ISA AMPs. In recent years, as GPU computing is becoming more and more important, scheduling on multi-ISA heterogeneous environments has been a hot topic. StarPU [5], Qilin [24], Glinda [33] and HDSS [7] are representatives that can simultaneously execute suitable compute programs for different data portions on CPUs and GPUs. As shown in the previous section, we found that 8-heap is the best choice for the CPU and 32-heap is the fastest on the GPU, thus the optimal scheduling method should execute the best d-heap operations on both types of cores in parallel. However, our results showed that the *ad*-heap is much faster than the optimal scheduling method. Thus scheduling is not always the best approach, although task or data parallelism is obvious.



(a) $d = 8$ (b) $d = 16$ (c) $d = 32$

(d) $d = 64$ (e) $d = 128$ (f) *aggregated results*

Figure 6: Selection rates and *ad*-heap execution time over different sizes of the sub-lists on the machine 2. The line-shape data series is aligned to the primary Y-axis. The stacked column-shape data series is aligned to the secondary Y-axis.

Pipelining is another widely used approach that divides a program into multiple stages and executes them on most suitable compute units in parallel. Heterogeneous environments further enable pipeline parallelism to minimize serial bottleneck in Amdahl's Law [17, 29, 14, 26]. Chen et al. [9] pipelined *map* and *reduce* stages on different compute units. Additionally, pipelining scheme can also expose wider design dimensions. Wang et al. [37] used CPU for relieving GPU workload after each previous iteration finished, thus overall execution time was largely reduced. He et al. [16] exposed data parallelism in pipeline parallelism by using both CPU and GPU for every high-level data parallel stage. Actually, in the *ad*-heap, the *find-maxchild* operation can be seen as a parallelizable stage of its higher-level operation *delete-max* or *update-key*. However, the *ad*-heap is different from the previous work because it utilizes advantages of the AMPs through frequent fine-grained interactions between the LCUs and the TCUs.

Prefetching data can be considered with heterogeneity as well. Once GPU and CPU share one cache block, the idle integrated GPU compute units can be leveraged as prefetchers for improving single thread performance of the CPU [39, 40], and vice versa [41]. Further, Arora et al. [4] argued that stride-based prefetchers are likely to become significantly less relevant on the CPU while a GPU is integrated. If the two types of cores shared the last level cache, the *ad*-heap can naturally obtain benefits from heterogeneous prefetching, because the bridge and the nodes to be modified are already loaded to the on-chip cache by the TCUs, prior to writing back by the LCUs. Because of the legacy CPU and GPU architecture design, in this paper we choose focusing on an AMP environment with separate last level cache sub-systems. Conducting experiments on a shared last level cache AMP can be an interesting future work. Additionally, our approach is different from the previous work since we see both TCUs and LCUs as compute units as well but not just prefetchers.

6. CONCLUSIONS

In this paper, we proposed *ad*-heap, a new efficient heap data structure for the AMPs. We conducted empirical studies based on the theoretical analysis. The experimental results showed that the *ad*-heap can obtain up to 1.5x and 3.6x performance of the optimal scheduling method on two representative machines, respectively.

To the best of our knowledge, the *ad*-heap is the first fundamental data structure that efficiently leveraged the two different types of cores in the emerging AMPs through fine-grained frequent interactions between the LCUs and the TCUs. Further, the performance numbers also showed that redesigning data structure and algorithm is necessary for exposing higher computational power of the AMPs.

7. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments on this paper.

8. REFERENCES

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] AMD. *White Paper: Compute Cores*, jan 2014.

[3] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 298–309, 2005.

[4] M. Arora, S. Nath, S. Mazumdar, S. Baden, and D. Tullsen. Redefining the role of the cpu in the era of cpu-gpu integration. *Micro, IEEE*, 32(6):4–16, 2012.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, feb 2011.

[6] B. Beaty. *DKit: C++ Library of Atomic and Lockless Data Structures*, 2012.

[7] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, jan 2013.

[8] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *IEEE Micro*, 32(2):28–37, 2012.

[9] L. Chen, X. Huo, and G. Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 25:1–25:11, 2012.

[10] E. Chung, P. Milder, J. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 225–236, 2010.

[11] M. Daga, A. M. Aji, and W.-c. Feng. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, SAAHPC '11, pages 141–149, 2011.

[12] M. Daga and M. Nutter. Exploiting coarse-grained parallelism in b+ tree searches on an apu. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 240–247, 2012.

[13] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah. A 22nm ia multi-cpu and gpu system-on-chip. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 56–57, 2012.

[14] M. Deo and S. Keely. Parallel suffix array and least common prefix for the gpu. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 197–206, 2013.

[15] T. Furtak, J. N. Amaral, and R. Niewiadomski. Using simd registers and instructions to enable instruction-level parallelism in sorting algorithms. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 348–357, 2007.

[16] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, aug 2013.

[17] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[18] HSA Foundation. *HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG)*, 0.95 edition, may 2013.

[19] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3):53 – 57, 1975.

[20] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, 2011.

[21] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 64–, 2004.

[22] G. Kyriazis. Heterogeneous system architecture: A technical review. Technical report, AMD, aug 2013.

[23] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *J. Exp. Algorithmics*, 1, jan 1996.

[24] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, 2009.

[25] D. Lustig and M. Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 354–365, 2013.

[26] P. Mistry, Y. Ukidave, D. Schaa, and D. Kaeli. Valar: A benchmark suite to study the dynamic behavior of heterogeneous systems. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 54–65, 2013.

[27] N. Nishikawa, K. Iwai, and T. Kurokawa. Power efficiency evaluation of block ciphers on gpu-integrated multicore processor. In Y. Xiang, I. Stojmenovic, B. Apduhan, G. Wang, K. Nakano, and A. Zomaya, editors, *Algorithms and Architectures for Parallel Processing*, volume 7439 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin Heidelberg, 2012.

[28] nVidia. *NVIDIA Tegra 4 Family GPU Architecture*, 1.0 edition, feb 2013.

[29] J. Pienaar, S. Chakradhar, and A. Raghunathan. Automatic generation of software pipelines for heterogeneous parallel systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12, 2012.

[30] Qualcomm. *Qualcomm Snapdragon 800 Product Brief*, aug 2013.

[31] A. Sadrieh, S. Charissis, and A. Hill. An on-chip heterogeneous implementation of a general sparse linear solver. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 54–63, 2013.

[32] Samsung. *Enjoy the Ultimate WQXGA Solution with Exynos 5 Dual*, 2012.

[33] J. Shen, A. L. Varbanescu, H. Sips, M. Arntzen, and D. G. Simons. Glinda: A framework for accelerating imbalanced applications on heterogeneous platforms. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 14:1–14:10, 2013.

[34] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter. The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 103–112, 2012.

[35] K. Van Craeynest and L. Eeckhout. Understanding fundamental design choices in single-isa heterogeneous multicore architectures. *ACM Trans. Archit. Code Optim.*, 9(4):32:1–32:23, jan 2013.

[36] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 213–224, 2012.

[37] J. Wang, N. Rubin, H. Wu, and S. Yalamanchili. Accelerating simulation of agent-based models on heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 108–119, 2013.

[38] M. A. Weiss. *Data Structures and Algorithm Analysis*. Addison-Wesley, second edition, 1995.

[39] D. H. Woo, J. B. Fryman, A. D. Knies, and H.-H. S. Lee. Chameleon: Virtualizing idle acceleration cores of a heterogeneous multicore processor for caching and prefetching. *ACM Trans. Archit. Code Optim.*, 7(1):3:1–3:35, may 2010.

[40] D. H. Woo and H.-H. S. Lee. Compass: A programmable data prefetcher using idle gpu shaders. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 297–310, 2010.

[41] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. Cpu-assisted gpgpu on fused cpu-gpu architectures. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, 2012.