Implementing Radix Sort on Emu 1

Marco Minutoli*, Shannon K. Kuntz†, Antonino Tumeo*, Peter M. Kogge†
*Pacific Northwest National Laboratory, Richland, WA 99354, USA
E-mail: {marco.minutoli, antonino.tumeo}@pnnl.gov
†Emu Solutions, Inc., South Bend, IN 46617, USA
E-mail: {skuntz, pkogge}@emutechnology.com

Abstract—**This paper discusses the implementation of radix sort on Emu Solutions' Emu 1 system. Sorting is an important building block for a large class of data-intensive applications. Radix sort, in particular, is a good candidate for sorting large sequences of fixed-length integer keys (for example, indexes in a database), because it presents lower computational complexity with respect to comparison based sorting algorithms when the length of the keys is small with respect to the dataset size. In general, implementing scalable sorting algorithms for large datasets is challenging, due to the large amount of memory operations with low data locality to move keys to their new positions. The Emu architecture is a memory-centric design based around the concept of lightweight threads that migrate across nodelets, a combination of memory and multithreaded processors aimed at providing a scalable large-memory system. We show how the Emu 1 design provides scalability in performance and size to our radix sort implementation.**

I. Introduction

Sorting is a basic algorithm that plays a key role in a large variety of applications. Having an efficient sorting kernel is particularly important in the emerging class of data-intensive applications. In many cases, results from queries on very large databases need to be sorted on some criteria (e.g., key), and the results themselves could be of significant size. Among all the sorting algorithms, radix sort is a good candidate for sorting large sequences of fixed-length integer keys: it has lower computational complexity, and thus it is intrinsically faster, than comparison based sorting algorithms when the length of the keys is small with respect to the dataset size. Many applications that need to sort key-value pairs often employ radix sort.

Radix sort works by grouping keys by the individual digits that share the same significant position (radix), and then ordering the group of keys according to the values of the digits. Key grouping is usually done with counting or bucket sort, generating a histogram of the distribution of the keys with respect to the considered digits of a radix. The basic radix sort algorithm formulation is serial, but there are many efficient parallel implementations. However, as in any sorting algorithm, radix sort also exhibits significant data movement, often with poor locality, as keys are assigned to their new position. Many of the available implementations, targeting multi- and many-core Central Processing Units (CPUs) and Graphic Processing Units (GPUs), focus on exploiting data partitioning and locality to improve the performance.

This paper discusses the optimization process of the radix sort algorithm for the Emu Solutions' Emu 1 system. Emu is a memory-centric architecture designed to scale to very large (memory) systems. It employs near-memory processing concepts together with the concept of migrating threads to provide scalability. The system implements an extended Cilk-based programming model that allows implementing with relative ease algorithms running on the whole system. We discuss similarities and differences of implementing a scalable radix sort on Emu 1 with respect to other systems, and provide initial results obtained with a simulation infrastructure of the architecture.

The paper proceeds as follows. Section II briefly summarizes some previous works on radix sort. Section III provides an overview of the Emu architecture. Section IV discusses our radix sort implementation, tying it in with the Emu architecture features. Section V presents our experimental evaluation and, finally, Section VI concludes the paper.

II. Related Work

Many works focus on the optimization of radix sort on multicore processors, manycore processors, and GPUs. Sohn and Kodama [1] present a balanced parallel radix sort implementation. Their solution first computes the bin counts for all the processors, and then computes, for each processor, the number of keys, their source bin and their source processor. This computation is done with one all-to-all bin count transpose operation in every round. The partitioned parallel radix sort, proposed in [2], improves upon the load balanced implementation by reducing the multiple rounds of data redistribution to one. The paper [3] employs a most significant digit (MSD) radix sort (i.e., a radix sort that starts from the most significant digit, in contrast to the classic implementation which starts from the least significant digit) to evaluate scatter and gather primitives for GPUs.

The paper [4] presents both a merge and a radix sort for GPUs. The proposed implementations employ optimized data-parallel primitives, such as parallel scan, and hardware features such as the high-speed on-chip shared memory of GPU architectures. The radix sort implementation divides the initial input in tiles assigned to different threads blocks and sorted in the on-chip memory using series of 1-bit split operations [5]. Split operations are Single Instruction Multiple Data (SIMD) friendly: they are performed with a SIMD prefix sum and a SIMD subtraction. The proposed radix sort implementation reaches a saturated performance of around 200 Mkeys/s for floating point keys on a GeForce GTX 280.

The paper [6] compares the previous approach for GPUs to a buffer based scheme for CPUs. Buffers in local storage (i.e., pinned to private caches of cores) collect elements belonging to the same radix for each core. Cores write these buffers to the shared memory only when they accumulate enough elements. Such an implementation reaches up to 240 Mkeys/seconds on a Core i7 (Nehalem architecture) processor.

[7] adds to the analysis the Xeon Phi accelerator, previously known during development as the Intel Many Integrated Cores (MIC) architecture. The paper applies the previously described GPU approach to Knights Ferry (non commercialized version of the Xeon Phi). The authors show saturated performance respectively at about 325 Mkeys/s for the GeForce GTX 280, 240 Mkey/s for the Core i7 and 560 Mkeys/s on Knights Ferry.

Merrill and Grimshaw [8] present a number of radix sort implementations for GPUs. They all exploit an efficient parallel prefix scan "runtime" that includes kernel fusion, multi-scan for performing multiple related and concurrent prefix scans, and a flexible algorithm serialization that removes unnecessary synchronization and communication within the various phases. The best implementation was originally integrated into the NVIDIA Thrust Library, and it has been shown to reach a sustained performance over 1000 Mkeys/s with a GeForce GTX 480 (Fermi architecture).

Wassenberg and Sanders [9] present a radix sort algorithm that exploits a microarchitecture-aware counting sort. The implementation is based on the use of virtual memory: it generates as many containers as there are digits, and it inserts each value in the appropriate container. It also employs a form of write combining by accumulating in buffers values in the correct relative ordering before writing them to memory with non temporal stores that bypass all the data caches. On a dual Xeon W5580 (Nehalem architecture, 4 cores, 3.2 GHz) this solution reaches 657 Mkeys/s. However, the extensive use of virtual memory limits the maximum size of datasets.

Finally, in [10], Morari et *al.* present an efficient radix sort implementation for the Tilera TILE64Pro manycore processor, discussing a set of optimizations that include aggressive unrolling, the exploitation of the network on-chip, and write buffers. The implementation reaches a throughput of up to 138 MKeys/s, which is competitive with other systems when taking into consideration power consumption.

III. Emu architecture

The Emu architecture is a memory-centric design for scalable large memory systems. An Emu architecture system is composed of multiple nodes. A node hosts several *Stationary Cores* (SC), one or more *nodelets*, some non-volatile memory (NVM), and a *Migration Engine* that acts as the communication interface between nodelets and the system interconnect.

SCs are conventional cores that run a conventional operating system. Programs for the SCs are compiled to conventional Instruction Set Architectures (ISAs) with *stationary* threads that keep executing in the same node. The SCs also control the NVM within the nodes. The NVM could be, for example,

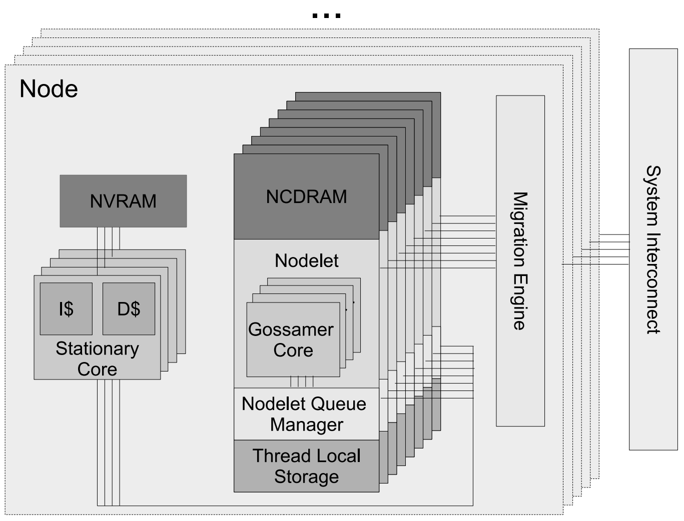

Fig. 1: Emu system level architecture

a set of Solid State Disks (SSDs) that implements a distributed high performance file system.

Nodelets are the basic building block and constitute the near-memory processing part of the system. A nodelet tightly couples processing elements, named Gossamer Cores, with a Narrow Channel DRAM (NCDRAM) system. The GCs are 64-way multithreaded general-purpose cores that execute extremely lightweight threads that are able to migrate across nodelets in any phisical location of the system. A nodelet also hosts a *Nodelet Queue Manager* (NQM), and a *Local Thread Storage* (LTS). NCDRAM employs a bus width of 8 bits (in contrast to a conventional bus of 64 bits). This allows reading a single word of 8 bytes when the 8-element burst of DDR memory completes, rather than the 64 bytes of conventional DIMMs, fitting behaviors of applications with irregular workloads and very poor locality. When a thread needs to migrate to another nodelet, the GC writes its context to the LTS and notifies the NQM. The NQM works with the Migration Engine to move a thread to the proper location, where it is stored in the destination nodelet's LTS. Communication between SCs in different nodes uses standard networking protocols over the system interconnect. The SCs may also create threads to perform communication on their behalf. An interface adjacent to the SCs allows creating and sending threads to the Migration Engine.

The Emu architecture exploits near data processing to provide more effective ways to implement atomic and in-place memory operations with respect to other multithreaded architectures. Additionally, it may enable future Emu architecture designs where massive number of thread contexts could be directly stored into the attached memory rather than in a dedicated storage, providing more opportunities for latency tolerance.

A. Address Space

The Emu architecture supports a Partitioned Global Address Space (PGAS), with memory contributed by each nodelet. Unlike other PGAS systems, the Emu architecture supports alternative naming schemes for addresses, so that consecutive addresses can be mapped to pages interleaved across nodelets. This enables programmers to define the striping of data structures across the nodelets in the systems. The Emu Application Programming Interface (API) enables configurations with up to 16 different segments using a striping factor that is a power of 2 greater or equal to the size of an 8 byte word. Address ranges may also be denoted as *Replicated* so that a thread may use the same address on different nodelets to access a local copy of the data. GC program code is replicated into the same address range on each nodelet, so that a thread may use the same PC on any nodelet to fetch the next instruction without needing the knowledge of where it is executing.

B. Migrating threads

When a thread accesses a memory location, it first checks to determine whether the address is local. If it is not, the thread's state is packaged by hardware and sent to the correct nodelet. If the destination is not on the current node, the address is used to route the thread over the inter-node network. The thread is then unpacked and stored in the LTS on the nodelet holding the targeted location. A GC then loads the thread for execution and the thread continues execution without any knowledge that it has moved. Note that the GCs are anonymous and any GC that sees a particular nodelet memory is equivalent to any other that sees the same memory.

The GCs' ISA has a number of unique features to support lightweight multithreading. A thread can efficiently create an independent child thread to execute in parallel with itself using a single instruction spawn. A rich set of atomic instructions is provided to perform read-modify-writes on memory without any possibility of intervening access by other threads or SCs. In order to reduce the thread state carried with migrating threads, the number of registers in use is tracked and instructions are provided to specify when registers are no longer needed.

C. Programming model

An application begins execution on one or more of the SCs and creates a thread state to access off-node memory or start a computation with migrating threads. Each thread may itself perform additional spawns to start additional operations in parallel. An SC has preloaded copies of any needed thread code to all nodelets where it may be executed.

The Emu architecture uses the Cilk programming model to express threads. Cilk is an extension of C that supports three keyword extensions: `cilk_spawn`, `cilk_sync`, and `cilk_for`. The cilk_spawn is a prefix to a conventional function call that converts it into a non-blocking call that proceeds execution independently of the code following the call (called the *continuation*). This is equivalent to starting a new thread to do the function call, and corresponds almost directly to the Emu architecture's spawn instruction. The cilk_sync indicates that the current thread cannot continue in parallel with its spawned children and must wait for all child threads in the enclosing scope to complete. There is an implicit cilk_sync at the end of each block that contains a cilk_spawn. The cilk_for comes from Cilk Plus7. It replaces a conventional for loop but there is no assumed order to the loop evaluations, so that a separate thread could execute each iteration of such loop. Notionally, the loop work is divided into a tree of cilk_spawns, where each spawn uses the body of the loop as an anonymous function to execute a subset of the loop iterations.

In addition to the Cilk-based keywords, the Emu API includes several intrinsic functions to access unique features of the architecture. Among them, there are atomic memory operations, including compare-and-swap and in-place memory operations (e.g., fetch-and-add), and migrating thread control functions, such as restart, restore, save state.

Although the Emu API offers intrinsics to exercise finer control over the system, the Emu architecture is substantially programmed like any NUMA shared-memory system. In systems employing modern multi-processors with advanced cache subsystems, a developer has to carefully optimize for locality, and provide full utilization of heavy cores. With the Emu architecture she or he must still consider how the data is laid out on the nodelets and the nodes, but only so to obtain a balance between memory operations that theadlets perform locally and the number of migrations (i.e., the injection rate of the interconnect among nodelets and nodes). Concurrency through multithreading provides the latency tolerance while maintaining the system fully saturated.

IV. Radix sort implementation

Radix sort works by dividing the bits of each key in k groups of l bits called radixes. Starting from the least significant l-bit digit[1], the algorithm iteratively:

- computes the histogram of occurrences of each l-bit digit;
- computes the offset of each key with respect to to the buckets determined during histogram calculation
- reorders the keys based on the offset

After k passes, the algorithm produces a total ordering of the keys. Radix sort computational complexity is $O(nk)$, where k is the number of radixes for fixed-length keys and n the number of elements in the datasets. Comparison-based sorting algorithms are lower- bounded by computational complexity of $O(n \log n)$, thus radix sort is faster for a large n. The number of radixes can vary from 1 to the number of bits used to represent the keys to be sorted. There is a trade-off between the number of passes and the size of the histogram $(2l)$.

The typical parallel radix sort implementation works by partitioning the initial dataset of N keys into M blocks $blocks_t$. There is a destination array for the progressively sorted keys (i.e., the sorting is not in place). Each block is assigned to

[1]Radix sort starting from the most significant digits - MSD - is not stable, in the sense that keys may not maintain relative ordering as they are progressively rearranged radix by radix, and is usually used to sort keys in lexicographic order

Algorithm 1 Parallel radix sort main loop.

```
for $pass = 0 \ldots P$ do
    $hist_t \leftarrow$ DO_LOCAL_HISTOGRAM($block_t, pass$)
    $offset_t \leftarrow$ COMPUTE_OFFSET($hist_t$)
    DISTRIBUTE_KEYS($offset_t, block_t, pass$)
    $block_t \leftarrow block\_sorted_t$
end for
```

a core/processor/thread t, which computes a local histogram for the block (DO_LOCAL_HISTOGRAM()). Starting from the local histograms, a global histogram is computed (usually through a parallel prefix sum operation) for the whole dataset. This allows computing a global offset (destination locations in the output array) for groups of keys with the same digit (COMPUTE_OFFSET()). Because radix sort from the LSD maintains the relative ordering of the keys, each core/processor can then move keys from its own input block to the position identified by the computed offset (DISTRIBUTE_KEYS()). Algorithm 1 shows the main loop of the parallel radix sort.

Our radix sort implementation for the Emu architecture follows similar principles. For each radix, we partition the datasets in blocks. Each block is processed by a different thread. The Emu architecture's threads are spawned through the cilk_spawn construct, by employing an appropriate grain size. Each thread computes a local histogram for its block. The dataset is strided using Emu architecture's custom memory allocation primitives so that blocks do not cross nodelets, minimizing unnecessary thread migrations. Computing the local histogram presents high locality. Implementations for conventional CPUs or GPUs usually use radix lengths that provide histograms fitting either in caches (for CPUs), or in on-chip shared memory (for GPUs). In the Emu architecture we instead look at eliminating thread migrations, trying to reach the maximum memory throughput by exploiting multithreading. Threads then compute offsets for each element in the block by executing a parallel prefix sum. Differently from classical parallel prefix sum implementations, where threads are progressively halved at each iteration in a tree-like fashion, on the Emu architecture we can implement an efficient shared memory prefix sum. The implementation works asynchronously, so that each thread spawns a child thread that propagates its value to the next position. Finally, each thread distributes the keys from its assigned block to a target location in the output buffer (scatter). This last phase of the algorithm is the most irregular, and in implementations for other architectures usually is a specific focus for optimizations. For example, a number of approaches accumulates multiple keys (usually, the number necessary to fill a cache line) in a "local" buffer pinned to the private cache before writing it to the global destination buffer, exploiting the property that keys maintains relative ordering. In the Emu architecture implementation, instead, there obviously are no cache optimizations, and threads transparently migrate if the target location is in another nodelet. Because we employ a multithreaded architecture, multithreading allows tolerating latencies for threads that have to migrate across the system. There obviously are tradeoffs in the number of threads with respect to the size of the datasets and, consequently, the size of each block.

In general, the implementation does not significantly differ from a conventional radix sort implementation for a shared-memory system. The Emu 1 design allows implementing the algorithm without considering the distributed, non uniform, memory organization. The only significant precaution has been making so that blocks are not striped across nodelets for the local histogram calculation, the most regular part of the algorithm, so to not induce migrations. On the other hand, the cacheless near-memory Emu 1 architecture does not mandate any particular optimization for the most irregular part of the algorithm, the key distribution, in contrast with other approaches.

V. Experimental Evaluation

We performed a preliminary evaluation of our radix sort implementation for the Emu architecture by executing the algorithm on an in-house simulation environment. This environment provides both an untimed functional model and a SystemC based architectural timing model. The simulator currently executes LLVM intermediate representation instructions. We explored tradeoffs in terms of number of nodelets and (migrating) threads with the timed model. The simulation targets the specifics of an Emu 1 memory server (which could become available in 2016). The Emu 1 system will include up to 8 nodelets in a single node. In Emu 1, a node can sustain up to 2.6B memory accesses per second, and has an injection rate into the network of up to 17.5 GB/s. We execute experiments with up to 128 nodelets, i.e., up to 16 nodes. 16 nodes correspond to a tray in a rack. A full Emu 1 rack is projected to have 1024 nodelets, corresponding to 128 nodes (8 trays).

Figure 2 shows the design space exploration of our radix sort implementation with a dataset of 64 MKeys (512 MB of data, with 64-bit integer keys) on Emu 1 systems employing from 8 to 32 nodelets. For each configuration, we progressively increase the number of threads (8, 16, 32 and 64) initially spawned for eache nodelet, and present the throughput reached in terms of Millions of Keys sorted per second (MKeys/s). As previously described, our implementation is characterized by a regular phase (local histogram calculations), a prefix sum, which although not regular can be efficiently implemented in parallel with minimal number of migrations across nodelets, and an irregular phase (key distribution). We can start the regular phase with an equal partitioning of the datasets across the nodelets, spawn threads on the specific nodelets, and have the same number of operations per thread. The latency for accessing remote memory locations impacts the throughput only on the last phase. We can see that with this dataset size, on a small system (1 node, 8 nodelets) performance keeps increasing (getting near to 60 MKeys/s) as the number of threads increases. The increase in the number of threads allows increasing the memory reference rate of the system, and the migration engine is able to sustain any movement of threads from one nodelets to the other. With a single node, the network interconnect is not used. Moving to a higher number of nodelets, we get an increase in performance up to 32 threads.

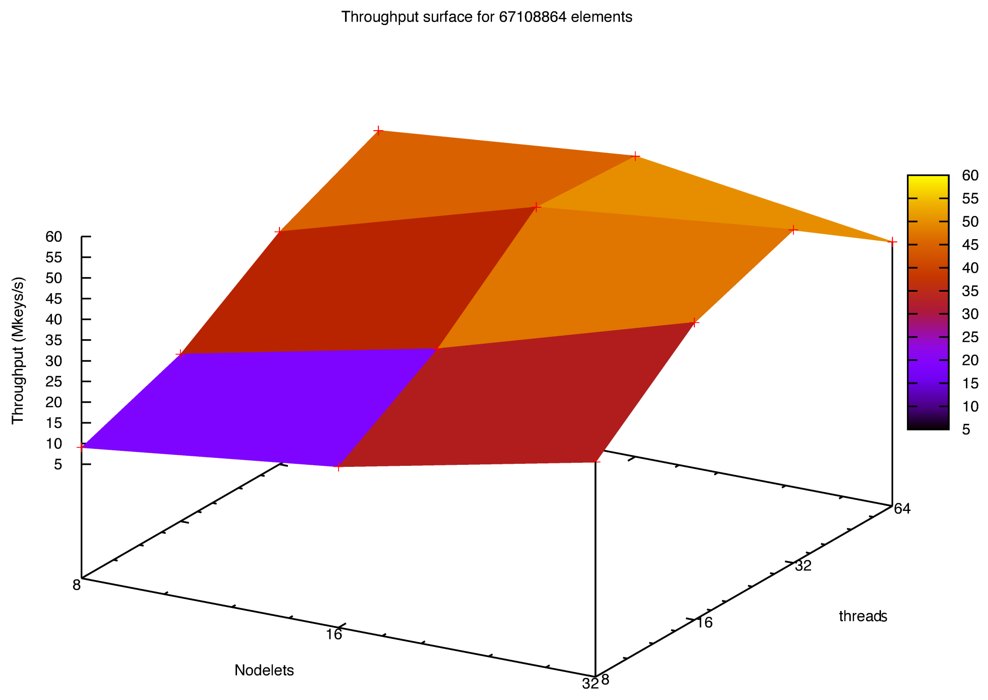

Fig. 2: Exploration of radix sort on a 64MKeys dataset using Emu 1 systems composed of 8, 16 and 32 nodelets with 8, 16, 32 and 64 threads

Increasing the number of nodelets obviously also increases the overall reference rate of the system. However, with to 64 threads per nodelet, the increase in performance appears more limited than with only 8 nodelets and the same number of threads. The network starts seeing significant utilization, and the work assigned to each thread is not able to make up for the slower network access rate. We see a similar behavior with 32 nodelets. With only 16 threads per node we obtain again an increase in performance with respect to configurations with lower number of nodelets but same number of threads per nodelet. Performance still increases as the number of threads per node gets to 32, but with a reduced slope. Moving to 64 threads per nodelet, we finally see a reduction in performance. Migrations become the constraining factor, and the work done by each thread is not sufficient to maximize the reference rate of threads that do not migrate (stay).

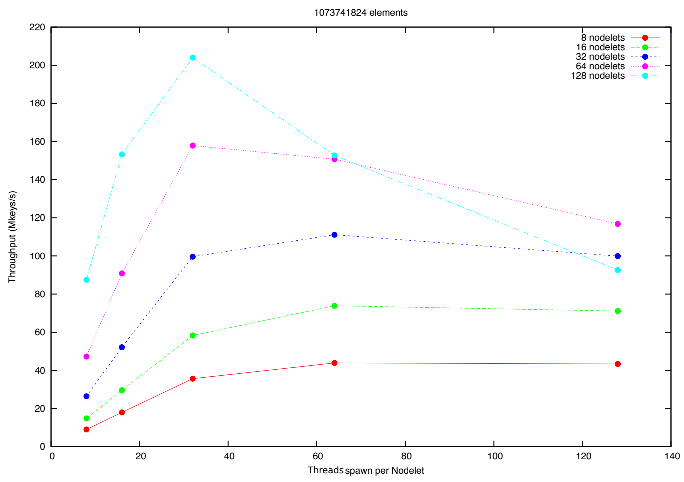

Fig. 3: Exploration of radix sort with a large dataset of 1B keys on Emu 1 systems of 8, 16, 32, 64, and 128 nodelets

Figure 3 shows the throughput (in Millions of sorted keys per second) for a significantly bigger dataset on larger systems. We sort an array of 1B 64-bit random integers (8 GB of data) . Because we do not perform in-place sorting, we need an output array of the same size (however, the output array becomes the input array for the next iteration). With the support data structures needed to compute the local histograms and calculate the global offsets, occupation nears or surpasses the memory available in modern accelerators. Single-GPU boards reach up to 12 GB nowadays, while some FPGA accelerators, such as the Convey Wolverine, can have up to 64 GB for each device. Note that the plot reports number of threads per nodelet on the X axis. The graph shows us that with systems up to 64 nodelets (4 nodes), using 32 threads provide around 80% of the peak throughput. Throughput reaches its maximum at 64 threads, and either stabilizes (single node) or slightly decreases after that, probably because of node interconnection contention. However, because of the large dataset, performance consistently increases when increasing the number of nodelets. With 64 and 128 nodelets, we reach peak throughput with 32 threads per nodelet. A higher number of nodelets implies that more memory operations execute in parallel (we have a higher overall memory reference rate). This benefits, in particular, the local histogram calculation phase. However, performance starts decreasing over 32 threads. Each thread has less work to do, and having the dataset distributed on a larger number of nodelets implies a higher number of migrations, potentially saturating not only the migration engine, but also the interconnect. Our peak throughput is around 210 MKey/s. As references, the best radix sort implementations for latest generation GPUs (Kepler) can reach up to 800 Mkeys/s with 64-bit integer keys[2]. With 32-bit keys, the best radix sort implementations reach around 560 Mkeys/s on the Intel Phi, 240 Mkeys/s on a 4-core 3.2 GHz Core i7 (Nehalem architecture) [7], and 138 Mkeys/s on a Tilera TILE64Pro [10]. These results, however, are obtained on much smaller datasets (up to 1M elements). Furthermore, our implementation exploits locality for the regular phase of the algorithm, but we do not make specific efforts to improve it in the irregular part.

We underline that, although sufficiently stable to perform a preliminary evaluation and obtain useful information on the scalability of the system and of the algorithm, the tools for Emu 1 are continuously evolving. Therefore, there are many opportunities to obtain further performance improvements as the architecture design evolves and the code is better tuned to the platform. Due to the maturity of the tools and the amount of time they have been available, our experiments are more focused at understanding the functionality and the behavior of the system, rather than achieving ultimate performance. We, however, think that this initial analysis provides interesting insights, and has particular value in understanding how the system and its programming model cooperate in enabling scalability for a memory-centric system, often a pain point for large scale machines dealing with memory-intensive workloads and for novel near memory designs.

VI. Conclusions

We have implemented radix sort for Emu Solutions' Emu 1, a memory-centric machine design which employs near memory processing components. The Emu architecture is based around the concept of lightweight threads that are able to migrate from one memory nodelet (combination of memory and simple multithreaded processors) to another to reach the data, rather than remotely loading the data. Multithreading provides latency tolerance during migrations. The design allows for large memory systems by combining multiple nodelets in nodes, and multiple nodes together through high performance network interconnect. The programming model exposes a global memory address space, with a configurable striping across nodelets, providing a shared memory abstraction. It uses Cilk to express parallelism and simple intrinsics for atomic memory operations.

We have described our radix sort implementation, highlighting similarities and differences with respect to other systems. The Emu architecture and programming model allow implementing the algorithm without caring about the distributed, non uniform, nature of the system, and limiting the number of optimizations required, especially for the most memory intensive, irregular parts of the algorithm. We have performed a design space exploration of the system with radix sort through a simulation infrastructure, and provided results that show promising scalability in size and performance for the system.

[2]Radix Sort implementation from the NVIDIA Thrust parallel methods library, available at: https://developer.nvidia.com/Thrust

References

[1] A. Sohn and Y. Kodama, "Load balanced parallel radix sort," in *ICS '98: international conference on Supercomputing*, 1998, pp. 305–312.

[2] S.-J. Lee, M. Jeon, D. Kim, and A. Sohn, "Partitioned parallel radix sort," *J. Parallel Distrib. Comput.*, vol. 62, pp. 656–668, April 2002.

[3] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *SC '07: ACM/IEEE conference on Supercomputing*, 2007, pp. 46:1–46:12.

[4] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *IPDPS '09: IEEE International Symposium on Parallel and Distributed Processing*, 2009, pp. 1–10.

[5] G. E. Blelloch, *Vector models for data-parallel computing*. Cambridge, MA, USA: MIT Press, 1990.

[6] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort," in *SIGMOD '10: ACM SIGMOD International conference on Management of data*, 2010, pp. 351–362.

[7] ——, "Fast Sort on CPUs, GPUs and Intel MIC Architectures," Intel Labs, Technical Report, 2010.

[8] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

[9] J. Wassenberg and P. Sanders, "Engineering a multi-core radix sort," in *EuroPar '11: international conference on Parallel processing - Part II*, 2011, pp. 160–169.

[10] A. Morari, A. Tumeo, O. Villa, S. Secchi, and M. Valero, "Efficient sorting on the tilera manycore architecture," in *SBAC-PAD 2012: IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, 2012, pp. 171–178.