The First Self-Contained Hardware Implementation of the Parallel Radix Sort

Nathan V. Morrical[1], Patsy Cadareanu[1], Walter Lau Neto[1], and Max D. Austin[1]

[1]The University of Utah, Salt Lake City, UT, USA

As chip density reaches its limits, many programmers are switching to concurrent programming models to meet high performance computing demands. These concurrent models achieve massive performance gains by distributing similar computation to local compute modules, which typically work in a *Single Instruction Multiple Data* (SIMD) fashion. However, performance suffers when thread divergence is high, since individual threads need to execute different instructions which are often not parallelizable. Even with newer generation *Graphical Processing Units* (GPUs) like the Nvidia *Volta* and *Turing* architectures, thread divergence tends to cause uncoalesced memory access patterns, minimizing potential memory bandwidth.

These complications can be resolved by aggregating similar computation together, which is achievable through a sort. As a result, many database systems, computer graphics data structures, and linear algebra systems all depend on efficient sorting as a fundamental building block. However even when parallelized, sorting is computationally demanding and becomes the speed-limiting factor. For example, more than 50% of the parallel BVH construction algorithm in [1] is spent sorting numbers.

To improve sorting performance, this paper presents a hardware-accelerated parallel-radix sorter capable of sorting an arbitrary number of elements and bits in linear time. Parallel radix sort was chosen due to its non-comparative sorting algorithm which allows for improved parallelization. It is also the fastest GPU parallel sorting algorithm to date [2] and a variant is used by Nvidia's *Thrust* library to perform sorting. The parallel-radix sort is composed of three repeated stages.

The first stage is **predication**. This works as follows: For each element in parallel, copy the bit at the current iteration. The predication for each element equals the extracted bit compared with the ascending/ descending flag.

The second stage is the **prefix sum**: For each prediction element in parallel, take the sum of all elements before and including the current prediction element. Figure 1 shows an example of a prefix-sum iteration on 8 elements of 3-bits for a better visualization.

The final stage is **compaction**: If the predication corresponding to an element is 1, move that element to the left. Otherwise, move that element to the right. For an example showing the radix-sort at work for 4 elements of 3-bits, see Figure 2.

These three stages are repeated for the total number of bits to sort.

Our implementation of the parallel radix sorting algorithm is broken up into 4 modules. Three modules are used for each of the three major stages of the sorting algorithm, processing N input elements. The last module defines a finite state machine which iterates over the three stages K times, where K is the number of bits to sort. Figure 3 shows the block diagram for this design.

The predication module is composed of a set of MUXes, one per each element. The select line for these MUXes controls which bit to extract from each element. The Verilog pseudo-code used for this step is included in Figure 4. The prefix sum module is composed of several prefix iteration modules (seen in Figure 5), and works similar to a Kogge Stone adder. Figure 6 implements this module in pseudo-code. The compaction module is composed of several parallel MUX's, one per element. The select line for the compaction MUXes is driven using a procedural address computation, as seen in Figure 7. Finally, the finite state machine is composed of a register containing the current iteration, and the instantiation of the predication, prefix sum, and compaction modules. The pseudo-code for this module is included in Figure 8.

Our implementation can be compared against a similar, although more complicated design proposed by Liu *et al.* [3] which requires external memory. To our knowledge, no such device has been fabricated, packaged, and tested at this time. We believe our implementation is the first to undergo complete logical and physical synthesis following the TSMC 180 nm technology. In addition, the proposed radix sorter will sort a specified number of bits in either ascending or descending order, and requires no external memory, making it completely self-contained.

In application, several instances of our radix sorter would be used in combination to sort separate contiguous sections of a larger sequence of numbers.This would resolve the uncoalesced access patterns of a typical parallel radix sorter by aggregating subsequences in a local memory array, at which point coalesced rearrangement can occur at a higher level.

Figure 9 presents the results after running logic synthesis with the Design Compiler. The total area of the chip is 314,741 μm^2 with the compaction module taking up the most area at 35,504 μm^2. The total power dissipation simulated for the chip is 4.75 mW, and the critical path delay is simulated at 2.35 ns. Figure 10 shows the results after running physical synthesis, *i.e., place and route* (PnR), with Innovus which considers wire capacitances and resistances. The final total area of the chip after PnR is 1,169,641 μm^2, the total power dissipation is 35.84 mW, and the arrival time is 4.59 ns. Figure 11 shows the final die where all metrics were extracted for both logic and physical synthesis.

Acknowledgements
The authors would like to acknowledge Edouard Giacomin for his assistance throughout this project.

References
[1] Karras, Tero. "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees." In Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics, pp. 33-37. Eurographics Association, 2012.
[2] M. C. Delorme, T. S. Abdelrahman, and C. Zhao, "Parallel Radix Sort on the AMD Fusion Accelerated Processing Unit," in Proceedings of International Conference on Parallel Processing, pp. 339–348, 2013.
[3] Liu, Xingyu, Shikai Li, Kuan Fang, Yufei Ni, Zonghui Li, and Yangdong Deng. "RadixBoost: A hardware acceleration structure for scalable radix sort on graphic processors." In Circuits and Systems (ISCAS), 2015 IEEE International Symposium on, pp. 1174-1177. IEEE, 2015.

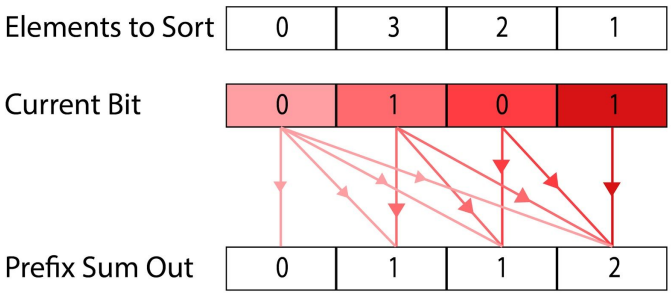

Figure 1: Example showing the prefix-sum iteration of the radix-sort on 8 elements of 3-bits.

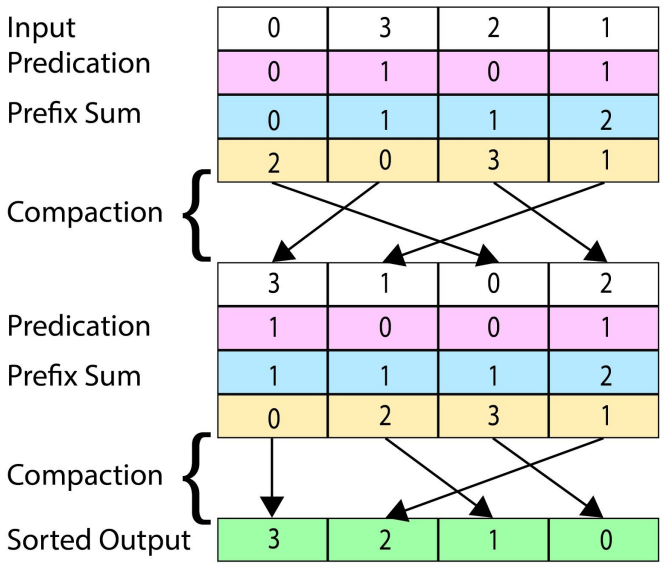

Figure 2: Example showing the radix-sort at work for 4 elements of 3-bits.

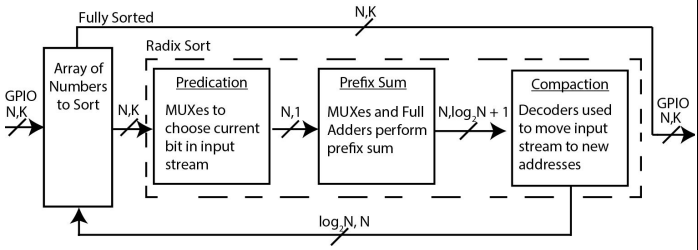

Figure 3: Hardware block diagram of our radix sorter.

Algorithm 1 *PREDICATION*

```
function PREDICATION
    Input:elements:reg list, is_descending:reg, bit_index:reg list
    Output:predication:reg list
    for each element address j in parallel do
        predication[j] ← (elements[j][bit_index] == is_descending)
    end for
end function
```

Figure 4: Verilog pseudo-code of the predication module.

Algorithm 2 *PREFIX_ITERATION*

```
function PREFIX_ITERATION
    Input:a:reg list, offset:integer
    Output:b:reg list
    for each address j in parallel do
        if j + 1 > offset then
            b[j] ← a[j] + a[j - offset]
        else
            b[j] ← a[j]
        end if
    end for
end function
```

Figure 5: Verilog pseudo-code of the helper function "prefix-iteration" called in the prefix-sum module, as seen in Figure 6.

Algorithm 3 *PREFIX_SUM*

```
function PREFIX_SUM
    Input:predication:reg list
    Output:prefix:reg list
    for each address bit i do
        Generate wires for layer i
    end for
    // Generate several prefix iteration modules, connecting wires together
    for j = 1 to the total number of address bits do
        PREFIX_ITERATION(wires[j − 1], wires[j], 2^(j−1))
    end for
    wires[0] ← predication
    prefix ← wires[number of address bits - 1]
end function
```

Figure 6: Verilog pseudo-code of the prefix-sum module.

Algorithm 4 *COMPACTION*

```
function COMPACTION
    Input:elements:reg list, predication:reg list, prefix:reg list
    Output:reordered_elements:reg list
    for each element address i in parallel do
        curr_prefix ← prefix[i]
        other_prefix ← prefix[total number of elements - 2]
        curr_pred ← predication[i]
        end_pred ← predication[total number of elements - 1]
        if curr_pred is 1 then
            addr ← curr_prefix - 1
        else
            addr ← i - curr_prefix + (end_pred + other_prefix)
        end if
        reordered_elements[addr] ← elements[i]
    end for
end function
```

Figure 7: Verilog pseudo-code of the compaction module.

Algorithm 5 *RADIX_SORTER*

```
function RADIX_SORTER
    reg list elements //elements to sort
    reg list predication
    reg list prefix_sum
    reg list reordered_elements
    for each bit idx in elements to sort do
        predication ← PREDICATION(elements, idx)
        prefix_sum ← PREFIX_SUM(predication)
        reordered_elements ← COMPACTION(elements, predication, prefix_sum)
        elements ← reordered_elements
    end for
end function
```

Figure 8: Verilog pseudo-code of the compaction module.

Block	Area (um^2)	Total Power (mW)	Arrival Time (ns)	WNS
Predication	3,632.11	0.30	0.99	Combinational
Prefix Sum	584.31	9.56E10^3	0.34	Combinational
Compaction	35,504.45	1.91	0.35	Combinational
Top (Radix Sorter)	314,741.46	4.75	2.35	1.56

Figure 9: The results of logic synthesis.

Block	Area (um^2)	Total Power (mW)	Arrival Time (ns)
Radix Sorter (Post P&R)	1,169,641.33	35.84	4.59

Figure 10: The results of physical synthesis.

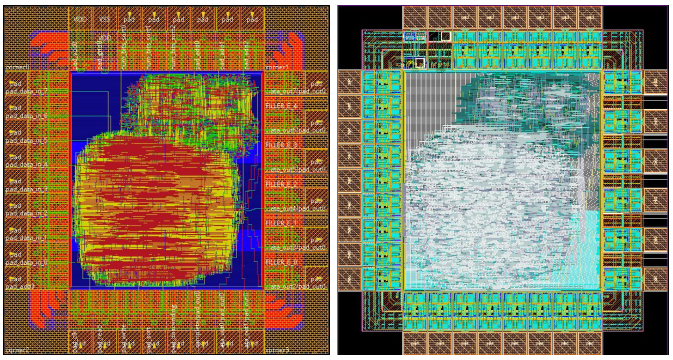

Figure 11: Physical layout and logical synthesis layout simulation.