RadixBoost: A Hardware Acceleration Structure for Scalable Radix Sort on Graphic Processors

Xingyu Liu, Shikai Li, Kuan Fang, Yufei Ni, Zonghui Li, Yangdong Deng
Institute of Microelectronics
Tsinghua University
{liuxingyu11, lsk11, fk10, niyf13, lizonghui11}@mails.tsinghua.edu.cn, dengyd@tsinghua.edu.cn

***Abstract*—In this paper, we propose RadixBoost, a hardware acceleration structure for scalable 32-bit integer radix sort on GPU. The whole structure is integrated into a GPU microarchitecture as a special functional unit and can be started by new instructions. Our design enables a significantly faster sorting procedure for general purpose GPU computing. The RadixBoost architecture was validated by an FPGA prototype integrated in FPGA-based GPU microarchitecture simulator, Fastlanes. An ASIC evaluation of RadixBoost was also performed. Our results proved that RadixBoost outperformed its GPU software equivalent by a factor of over 6 with an 1% and 3% increase in area and power respectively in cutting-edge Fermi GPU.**

***Keywords*—radix sort, hardware acceleration, GPU, FPGA, ASIC evaluation, prefix sum**

I. Introduction

As a fundamental tool in the area of computer science, sorting is widely used in many different applications. With the rise of graphics processor based computing, sorting is becoming even more essential since it is the basic building block of a large number of data parallel algorithms. Many GPU applications have their performance limited by the sorting process. For example, the sorting process consumes up to 42% of the total time in GPU based parallel kd-tree construction algorithms for real-time ray tracing [1].

To accelerate the sorting process, a number of recent research works sought to use custom hardware for a higher level of efficiency over pure software approaches [2]–[4]. However, previous works still suffer from relatively insufficient programming flexibility. Moreover, the overall impact of the sorting hardware on GPUs have not been systematically studied. In this work, we proposed a hardware accelerator, RadixBoost, to speed up 32-bit integer radix sort. The proposed hardware can be integrated into a GPU microarchitecture as a special functional unit and initialized with new instructions. The major contributions of our work are as follows.

- We proposed a hardware architecture for RadixBoost, which consists of modules connected in an optimized topology. Scheduling of modules are also designed to achieve coalescing in GPU global memory accessing.
- We implemented the FPGA prototype for RadixBoost and integrated it into FPGA-based GPU microarchitecture simulation framework, Fastlanes [5]. Our PTX compilation tool-chain is modified for the new instructions that starts the hardware-accelerated sorting.
- An ASIC evaluation for RadixBoost is performed and the timing, area and power results are compared with Nvidia's cutting-edge Fermi GPU.
- To the best of the authors' knowledge, RadixBoost is the first work to analysis sorting hardware acceleration in a complete GPGPU computing system. Experiment results showed that RadixBoost outperformed CUDA library Thrust [6] by a factor of over 6 with an area and power budget of 1% and 3% respectively of those of Nvidia GeForce GTX 580.

II. Parallel Scalable Radix Sort Algorithm

The proposed RadixBoost accelerator is based on the parallel radix sort algorithm introduced by [7] and [8]. Parallel prefix sum algorithms, both fixed-length and scalable, are used in the sorting process. We describe the algorithms as follows.

A. Parallel Radix Sort

Assume the input is an unsorted array of L elements of a width of B-bits. The elements are represented as d-digit number with a radix of r. The radix sort consists of d passes of counting sorts. In the ith pass, the array is sorted with respect to the ith least significant digit. The counting sort is divided into 3 phases, *Counting*, *Ranking* and *Scattering*.

1) Counting: The unsorted array is partitioned into T l-element tiles. Two arrays, $localcount$ and $localoffset$ are obtained in this phase. $localoffset$ has a length of Tl and is a combination of T l-element segments. The ith segment of $localoffset$ stores each element's local order within ith tile among elements with same value. $localcount$ has a length of Tr and is a combination of T r-element segments. The ith segment of $localcount$ is the histogram of the ith tile. The *Counting* phase is illustrated in algorithm 1.

Algorithm 1: counting phase of parallel radix sort

```
Input: $Array$:list, $L, l, r$:integer
Output: $localcount$, $localoffset$:list
begin
    $T$=ceil($L/l$);
    foreach $i$ in [0,$T$) in parallel do
        foreach $k$ in [0,$l$) in parallel do
            $tile[i][k] = (i \cdot l + k < L)?Array[i \cdot l + k] : 0;$
            foreach $j$ in [0,$r$) in parallel do
                $mask[i][j][k] = (i \cdot l + k < L)?(tile[i][k] == j) : 0;$
        foreach $j$ in [0,$r$) in parallel do
            $sum[i][j]$ = fixedLengthPrefixSum($mask[i][j]$);
            $localcount[i \cdot r + j] = sum[i][j];$
        foreach $k$ in [0,$l$) in parallel do
            $localoffset[i \cdot l + k] = mask[i][tile[i][k]][k];$
```

2) Ranking: Array $localcount$ is reorganized to form an array $globaloffset$ of length Tr. Another prefix sum is performed to $globaloffset$. Then $globaloffset$ stores the global position of the first element with each unique value in each tile. The *Ranking* phase is illustrated in algorithm 2.

Algorithm 2: ranking phase of parallel radix sort

```
   Input: $localcount$:list, $L, l, r$:integer
   Output: $globaloffset$:list
1  begin
2      $T$=ceil($L/l$);
3      foreach $j$ in [$0,r$) in parallel do
4          foreach $i$ in [$0,T$) in parallel do
5              $globaloffset[j \cdot T + i] = localcount[i \cdot r + j]$;
6      scalablePrefixSum($globaloffset$);
```

3) Scattering: For each element in the unsorted array, we sum up the corresponding element in $localoffset$ and $globaloffset$ to obtain the final order of the element. Then this element is written to the final position in the sorted array. The *Scattering* phase is illustrated in algorithm 3.

Algorithm 3: scattering phase of parallel radix sort

```
   Input: $globaloffset, localoffset, Array$:list, $L, l, r$:integer
   Output: $sortedArray$:list
1  begin
2      $T$=ceil($L/l$);
3      foreach $i$ in [$0,T$) in parallel do
4          foreach $k$ in [$0,l$) in parallel do
5              if $i \cdot l + k < L$ then
6                  $address[i][k] = globaloffset[Array[i \cdot l + k] \cdot T + i] + localoffset[i \cdot l + k]$;
7                  $sortedArray[address[i][k]] = Array[i \cdot l + k]$;
```

B. Parallel Scalable and Fixed-length Prefix Sum

As elaborated in sub-section 2.1, the prefix sum operation is of special importance in the radix sort process. For an array with an arbitrary length, we chop it into shorter tiles with a fixed length and perform fixed-length parallel prefix sum operations to each tile. A batch of tiles can be handled in parallel. Then these partial results are merged after all elements in the same tile are added by a specific base offset value. The base offset values are updated every batch of processing.

The designed fixed-length parallel prefix sum operation follows a computing flow as proposed by [9]. The computing flow consists of both an up-sweep phase and a down-sweep phase that offer sufficient data parallelism. The two phases are implemented as cascaded pipeline as illustrated in Figure 1.

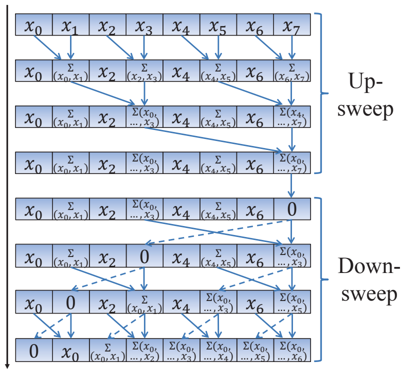

Fig. 1: Fixed-length parallel prefix sum of an 8-entry array

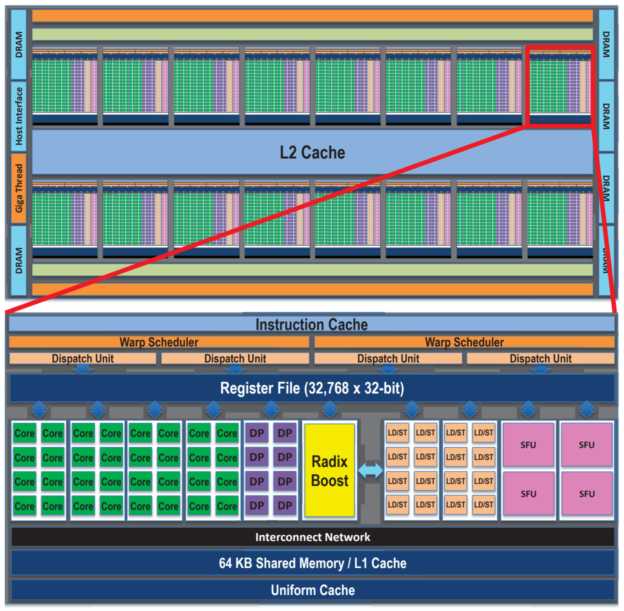

Fig. 2: RadixBoost integrated into Fermi GPU

III. Acceleration Structure Implementation

As depicted in Figure 2, RadixBoost is integrated into a GPU microarchitecture similar to NVIDIA's Fermi. One such accelerator is deployed in every streaming multiprocessor. RadixBoost has clock synchronization FIFOs and a dedicated bus connected with the load/store unit to exchange data with shared memory. The whole architecture was implemented in a GPU microarchitecture simulation framework, FastLanes [5].

A. Hardware Architecture

The hardware architecture of RadixBoost is illustrated in Figure 3. A full-featured RadixBoost contains 4 PSUs, 4 DSs, a ODQ, a set of parallel MUXes and a set of parallel adders. A global controller and buffers are also assigned. We will explain the details of the components of RadixBoost as follows.

1) Prefix Sum Unit (PSU): PSU takes charge of fixed-length prefix sum operation in *Counting* and *Ranking* phases. It consists of several cascaded stages of registers and uses the computing flow of fixed-length prefix sum mentioned in Section II. For hardware simplicity, we set the tile length in both scalable prefix sum and scalable radix sort as l, same as the length of PSU input.

2) Digit Selector (DS): DS judges whether a specific digit of each element in the input array equals a certain value and set the corresponding position in PSU input array to 1 or 0 in *Counting* phase and to the original value in *Ranking* phase. It contains l pairs of compare unit (CMP) and multiplexer (MUX). A pair of CMP and MUX is illustrated in Figure 4(a).

3) Original Digit Queue (ODQ): ODQ consist of several stages of $l \times 2$ bits wide register. Each stage of the queue keeps track of the original last 2 bits of currently processed digit of elements in the input arrays. Its number of stages equals the

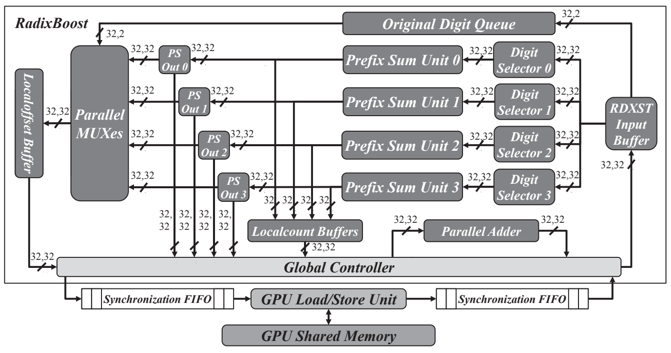

Fig. 3: RadixBoost architecture

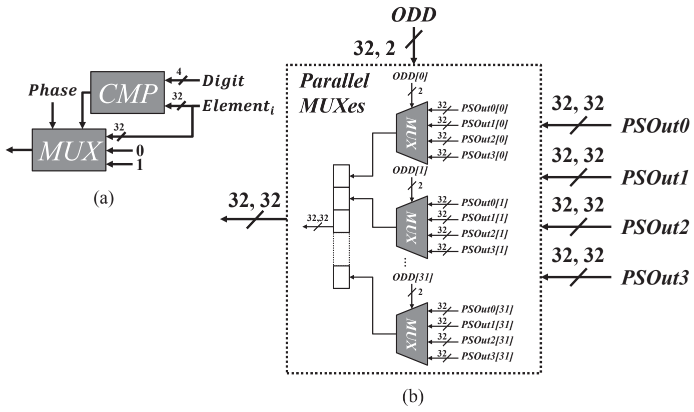

Fig. 4: (a) Example of a pair of CMP and MUX in Digit Selector (b) Structure of Parallel MUXes

sum of the number of stages of DS, PSU and PS output buffer.

4) Parallel MUXes: It consists of l 4-to-1 32-bit MUXes working in parallel. For each of the l entries, a MUX selects the value of the entry from the corresponding entries in 4 PSU outputs. The module is illustrated in Figure 4(b).

5) Parallel Adders: It consists of l 32-bit integer adders working in parallel in *Scattering* phase of scalable radix sort.

6) Buffers: All buffers have a width of 1024 bits. They fit GPU load/store unit's feature of reading or writing 32×32-bit integers during a shared memory access of a thread warp.

7) Global Controller: It handles data dispatch or collection and instruction issuing. It also schedules computing process to ensure coalesced global memory access in *Counting* and *Ranking* phase. Between global controller and GPU load/store unit, FIFOs are assigned for clock synchronization between two clock domains.

B. Design Parameter Considerations

In the hardware implementation of RadixBoost, we chose a radix value of 16 (i.e., $r = 16$) and a tile length of 32 (i.e., $l = 32$). We explain the choice of such a configuration as follows.

Generally speaking, the speed-up of data-parallel prefix sum over a sequential implementation is proportional to $log_2 l/l$ and a larger l benefits. However, the length l is limited by two factors. First, the number of stages is proportional to $\log l$ and thus the resultant usage of hardware resources is proportional to $l \times \log l$, which grows rapidly with regard to l. Second, the interface of the shared memory in a streaming multiprocessor is 32×32 bits in our setup and we cannot get more than 32 operands in one cycle. In light of above analysis, we set l as 32 to fully explore the potential of parallelism and avoid wasting of hardware resources.

As discussed above, the number of passes of counting sort is the number of digit, which is $d = \lceil B/\log_2 r \rceil$. A larger r will result in fewer passes of counting sort. However, a larger r also increases the number of clock cycles in each pass, since the amount of operands processed in *Counting* and *Ranking* phase is approximately proportional to r. Based on our experiment, we set r to 16 for a proper tradeoff in the work reported in this paper.

C. Compilation Tool-Chain

FastLanes' compilation tool-chain is based on NVIDIA PTX virtual instructions set [10]. The source code is developed in CUDA and compiled into PTX assembly code, which is then translated into the machine code of FastLanes and processed by a series of compiler optimizations. We defined 3 new instructions, FPFXSM, SPFXSM and RDXST, to start the hardware accelerated fixed-length prefix sum, scalable prefix sum and scalable radix sort respectively. Our compilation tool-chain will expand SPFXSM instruction to several FPFXSM instructions and RDXST instruction to several SPFXSM and FPFXSM instructions based on algorithms mentioned in Section II. Additional instructions that handle shared memory allocation and management is also created during the instruction expansion to provide buffers for intermediate results. A CUDA programmer just needs to call corresponding functions and our compilation tool-chain will recognize the function and create machine codes accordingly.

IV. Experiment and Results

In this section, we describe the hardware implementation of RadixBoost. Since Nvidia GPU's microarchitecture is not publicly available, we use FastLanes [5] to simulate the Fermi architecture. FastLanes can deliver a simulated instruction throughput within 15% of real execution on a NVIDIA Fermi GPU. We provide an FPGA validation of RadixBoost's integration with simulation clock cycles. Then we perform an ASIC evaluation and estimation for area, power and timing.

A. FPGA Prototype

RadixBoost-equipped FastLanes was deployed on a VC707 development board featuring a Xilinx Virtex-7 XC7VX485 FPGA and 1GB DDR3 memory. We used the Ethernet interface on the evaluation board to download unsorted array to FPGA and collect sorting result for verification. As illustrated in Table I, the FPGA hardware usage of a RadixBoost is reasonable. One streaming multiprocessors (SM) with one RadixBoost is implemented on FPGA. The setup is based on FastLanes' off-chip context switching mechanism that allows simulating an arbitrary number of multiprocessors with implementation of only one set of GPU functional and timing model on-chip [5].

TABLE I: FPGA Hardware Resources Usage Result

	Slice Register	Slice LUT	Distributed RAM (Kb)	Block RAM (Kb)
FastLanes	366,402 (60.34%)	91,190 (30.04%)	2,080 (25.44%)	500 (48.54%)
FastLanes & RadixBoost	428,373 (70.55%)	151,922 (50.05%)	2,080 (25.44%)	500 (48.54%)

B. ASIC Evaluation

For the ASIC evaluation, we use TSMC 65nm process [11] and Synopsys design compiler [12]. RadixBoost was synthesized up to 830 MHz with a voltage of 1 V. We set the frequency to 800 MHz. The area and power consumption of a RadixBoost is $0.65mm^2$ and 0.5 W, respectively.

C. Performance

We scaled RadixBoost to TSMC 40nm process so that it could be compared with the Nvidia Fermi GPU GeForce GTX 580 at the same technology node [13]. TSMC suggested that an improvement of 40% in speed and a down scaling factor of at least 2 in area is available when migrating from 65nm to 40nm [14]. We estimated the resultant area to be $0.33mm^2$ and the maximum clock frequency to be 1100 MHz. We chose a clock frequency of 1000 MHz. The power is scaled similarly but with the additional consideration of the increase of clock frequency. We estimated the power to be 0.4 W. Since each of 16 streaming multiprocessors (SM) in Fermi GPU should be equipped with a RadixBoost ASIC, the total area and power budget is $5.3mm^2$ and 6.4 W, respectively 1.0% and 2.6% of the die size and dynamic power of GeForce GTX 580 [15].

We tested RadixBoost-equipped FastLanes [5] with a set of 32-bit integer arrays with incremental lengths and elements of random values. We compared RadixBoost with CUDA parallel algorithm library, Thrust [6], on FastLanes with the same clock frequency. The sorting performance is measured in terms of the number of execution clock cycles. Figure 5 illustrates the performance results of RadixBoost and Thrust. The tested arrays are large enough to exceed the capacity of shared memory in a single SM, so the results can show the performance of the whole GPU microarchitecture.

Assuming the number of execution cycles to be the same when migrating RadixBoost from FPGA to a GPU chip, we scaled the execution time by taking into account the clock frequency of RadixBoost 40nm ASIC (1000 MHz) and SM of GeForce GTX 580 (1544 MHz) [15]. As a result, RadixBoost outperforms Thrust by a factor of 6.03. Our estimation is conservative, since our ASIC implementation of RadisBoost has a lower clock frequency than SM of GeForce GTX 580 GPU and thus actually takes fewer cycles in memory accessing than the FPGA implementation. Maximum memory bandwidth required by RadixBoost ASIC is 55.8GB/s, only 29% of the peak bandwidth of GDDR5 [15].

V. Conclusion

In this work, we proposed a hardware acceleration structure, RadixBoost, for scalable 32-bit interger radix sort on GPU. The proposed hardware architecture and corresponding

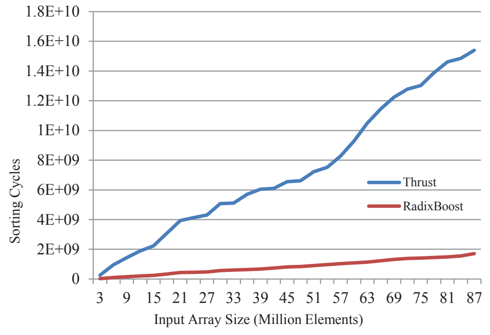

Fig. 5: Performance of RadixBoost and Thrust on FastLanes

compilation tool-chain was integrated into FPGA-based GPU microarchitecture simulation framework, FastLanes. We also provided an ASIC evaluation. Experiments showed that the RadixBoost enabled over 6-fold performance improvement over its software equivalent on cutting-edge Fermi GPU, requiring 1% and 3% increase in area and power, respectively.

References

[1] Z. Li, T. Wang, and Y. Deng, "Fully Parallel Kd-Tree Construction for Real-Time Ray Tracing," in *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2014.

[2] R. Marcelino, H. C. Neto, and J. M. P. Cardoso, "A Comparison of Three Representative Hardware Sorting Units," in *Proceedings of 35th Annual Conference of the IEEE Industrial Electronics Society*, 2009, pp. 2805–2810.

[3] D. Mihhailov, V. Sklyarov, I. Skliarova, and A. Sudnitson, "Application-Specific Hardware Accelerator for Implementing Recursive Sorting Algorithms," in *Proceedings of 2010 International Conference on Field-Programmable Technology*, 2010, pp. 269–272.

[4] D. Koch and J. Torresen, "FPGASort: A High Performance Sorting Architecture Exploiting Run-Time Reconfiguration on FPGAs for Large Problem Sorting," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011, pp. 45–54.

[5] K. Fang, Y. Ni, J. He, Z. Li, S. Mu, and Y. Deng, "FastLanes: An FPGA Accelerated GPU Microarchitecture Simulator," in *Proceedings of 2013 IEEE 31st International Conference on Computer Design*, Asheville, NC, 2013, pp. 241–248.

[6] Thrust, "Thrust," https://developer.nvidia.com/thrust, 2014.

[7] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," in *Proceedings of IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–10.

[8] 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*, 2013, pp. 339–348.

[9] M. Harris, S. Sengupta, and J. D. Owens, *Parallel Prefix Sum (Scan) with CUDA*. In GPU Gems 3, 2007.

[10] NVIDIA, "Parallel Thread Execution ISA Version 4.0," Retrieved on July 3, 2014 from http://docs.nvidia.com/cuda/parallel-thread-execution/index.html#axzz36YcuWqKF, 2014.

[11] TSMC, "65nm Technology," Technical Report. http://www.tsmc.com/english/dedicatedFoundry/technology/65nm.htm.

[12] Synopsys, "Power Optimization in Design Compiler," Technical Report, 2013.

[13] Y. Zhang, L. Peng, B. Li, J.-K. Peir, and J. Chen, "Architecture Comparisons Between Nvidia and ATI GPUs: Computation Parallelism and Data Communications," in *Proceedings of 2011 IEEE International Symposium on Workload Characterization*, 2011, pp. 205–215.

[14] TSMC, "40nm Technology," Technical Report. http://www.tsmc.com/english/dedicatedFoundry/technology/40nm.htm.

[15] Nvidia, "GeForce GTX 580 Specifications," http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications, 2014.