A CUDA-based Batcher-Banyan Network for Bitonic Sort

Kourosh Vali, Mahyar Samani, and Mason del Rosario

***Abstract*—We describe the Batcher-Banyan (BB) network, a sorting network which performs bitonic mergesort by alternating between comparisons and routings. We discuss how to partition the BB network such that memory transactions are coalesced, and we specify how shared memory can encompass large sections of the sorting network. We present a CUDA-based implementation of the BB network, and compare its performance to readily available CUDA-based sorting algorithms including bitonic sort and radix sort.**

***Index Terms*—Bitonic sort, sorting network, parallel computing, CUDA**

I. Batcher-Banyan Sorting Network

The Batcher-Banyan (BB) network [1] sorts an array of $N = 2^n$ elements by alternating between 1) creating bitonic sequences of length 2^m (with Batcher networks) and 2) merging bitonic sequences to generate sequences of length 2^{m+1} (with Banyan networks).

The BB network is comprised of two types of work: **comparisons** between adjacent elements and **routings** between different addresses.

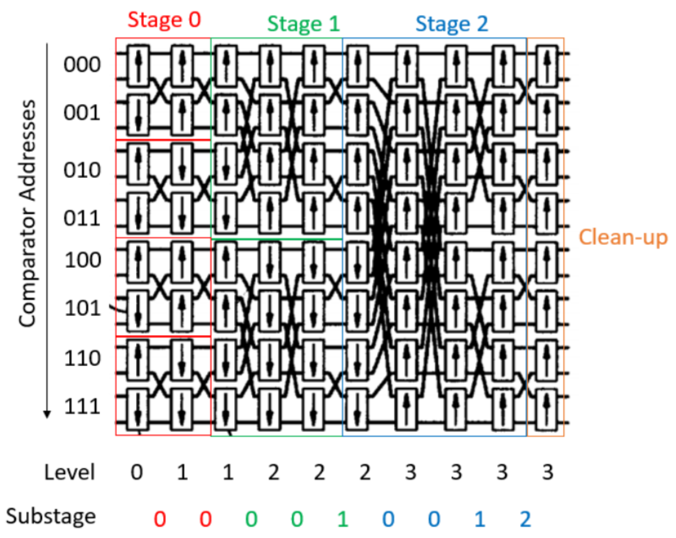


Fig. 1: Annotated BB network [2] showing different stages of work to perform.

This project report was completed in accordance with the final project requirements for the Winter 2020 offering of EEC289Q: Modern Parallel Computing taught by Prof. John Owens and Kerry Seitz at the University of California, Davis.

K. Vali, M. Samani, and M. del Rosario are with the Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA 95616 USA (e-mail: [kvali,msamani,mdelrosa]@ucdavis.edu).

A. Comparisons

Comparators (i.e., the boxes in Fig. 1) operate on adjacent elements in the array. Each comparator takes two elements at its input and moves the larger element to the arrow's tip and the smaller element to its tail. Looking at a single column of the network (Fig. 1), the i-th comparator looks at the elements at $2i$ and $2i+1$. The `level`-th bit of `comparator address` dictates the direction of the comparison. The discussion of how `level` is determined is in Section I-C.

B. Routing Toplogies

Routings are done by the wires connecting comparators (Fig. 1). The BB network uses two routing topologies: `shuffle` and `butterfly`. Each topology routes 2^n inputs to 2^n outputs (with $n \geq 2$). Figure **??** shows the structure of the routing topologies for $n = 4$ (i.e., 16 inputs/outputs).

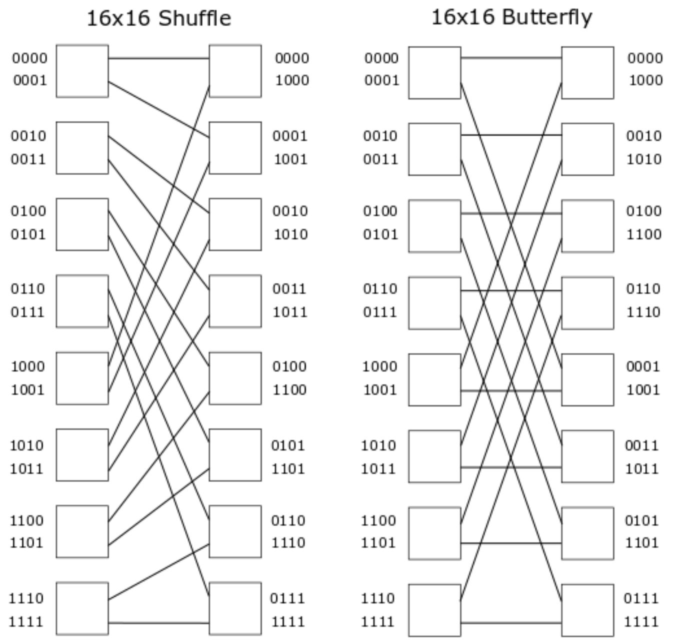


Fig. 2: Two routing topologies used in BB network, `shuffle` and `butterfly`, routing 16 elements.

How are the inputs addressed mapped to the output addresses for each of these routing topologies? Consider the binary representation for the addresses. Denote the number of elements to be routed as M and the $i-$th input (output) address as $\mathbf{in}_i(\mathbf{out}_i)$. The address manipulation for each routing is as follows:

- `shuffle`: Circularly shift $\mathbf{in}_i$ one bit to the right, store in $\mathbf{out}_i$.
- `butterfly`:
  - If $\mathbf{in}_i < M/2$ and $\texttt{mod}(\mathbf{in}_i, 2) == 1$, then switch the first and last bit of $\mathbf{in}_i$ and store in $\mathbf{out}_i$.
  - If $\mathbf{in}_i \geq M/2$ and $\texttt{mod}(\mathbf{in}_i, 2) == 0$, then switch the first and last bit of $\mathbf{in}_i$ and store in $\mathbf{out}_i$.
  - Else, $\mathbf{out}_i = \mathbf{in}_i$

C. *Stage-Based Algorithm*

Algorithm 1 shows pseudocode for the **stage-based algorithm** for the BB network. Figure 1 highlights the stages of the algorithm which the BB network follows [2]. `level` determines how to set a given column of comparisons, and `substage` determines the height of a given `shuffle` or `butterfly` network.

Algorithm 1: Stage-based algorithm for Batcher-Banyan sorting network.

```
Result: Sorted array of length $N = 2^n$
Initialize stage $= 0$, level $= 0$, substage $= 0$;
Input x[N];
while While stage $< n$ do
    while substage $\leq$ stage do
        size = pow(2, 2+stage−substage);
        if stage $< n - 1$ then
            if substage==0 then
                comparators(x,size,level);
                shuffle(x,size);
                level++;
            end
            comparators(x,size,level);
            butterfly(x,size);
            substage++;
        else
            comparators(size,level);
        end
        substage $= 0$;
        stage++;
    end
end
```

II. CUDA Implementation

The BB network lends itself well to parallelization. Each column of comparators in Fig. 1 only acts on two adjacent locations in memory at any given time, meaning they can be performed in parallel. Furthermore, each routing topology is a bijection between the its input and its output, so any given thread will write its output to an address without conflicting with another thread.

The repository includes two implementations: a **global memory** version and a **shared memory** version.

A. *Global Memory BB Network*

When using global memory for inter-block communication, three `__global__` kernels are implemented: `compareAndSwap`, `shuffleN`, and `butterflyN`. These kernels are called from the CPU sequentially as per Algorithm 1.

B. *Shared Memory BB Network*

Rather than calling `__global__` kernels from the CPU, the shared memory implementation defines the work kernels as `__device__` functions, and a `__global__` kernel populates shared memory with portions of the target array then calls the `__device__` functions.

Given particularly large arrays, shared memory might be insufficient to store portions of the array. Consider Fig. 3, which illustrates a case where our shared memory is only capable of storing eight elements.

We dispatch two thread blocks (red and blue) on two halves of the network. Once we reach the purple region, we see that the routing crosses over the red-blue boundary, meaning the addresses which the routing outputs are outside of each block's shared memory. At this point, the kernel needs to write elements out to global memory and re-read those values back into shared memory before continuing execution.

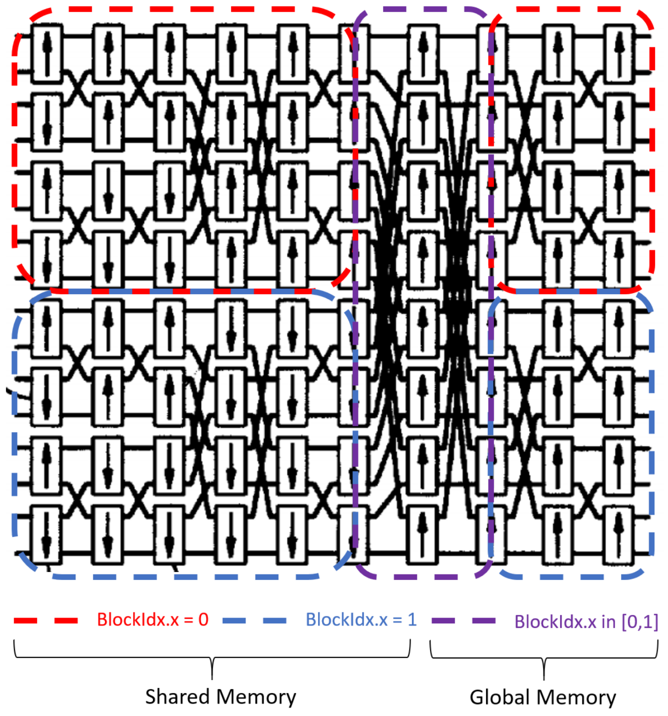
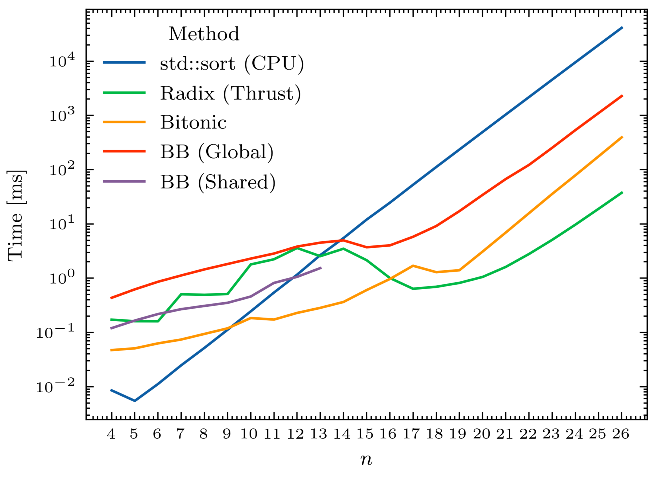
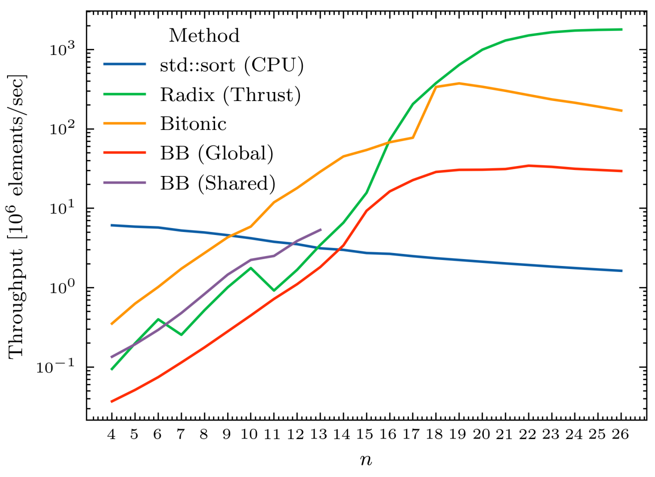


Fig. 3: Annotated BB network [2] showing hypothetical case where shared memory available on an individual SM is smaller than a portion of the array.

We dispatch thread blocks on "stage blocks" of the sorting network. Virtually, we imagine that a stage block is equivalent to at most one thread block. If the number of threads in a block is larger than the stage block's height, m stage blocks will be



(a) Time



(b) throughput

Fig. 4: Time to sort (left) and memory throughput (right) of sorting algorithms for increasing number of floating point elements (2^n elements for $n \in [4, 5, \ldots, 26]$). Batcher-Banyan using global memory (**BB Global**) and shared memory (**BB Shared**) are the proposed sorting networks. Two GPU-based algorithms (**bitonic sort** and **radix sort**) and one CPU-based algorithm (std::sort) are included as benchmarks. Values are averaged over ten trials for each data point.

emulated by one thread block where $m = \frac{blockdim.x}{\text{stage block height}}$. On the other hand, if a thread block is smaller in size compared to a stage block, then each thread block will stride inside a specific stage block before the whole grid is moved over the stage blocks to cover all of the blocks in a stage. In both cases in order to keep track of the shared memory, each comparator has global and local addresses. The global address is used to index the global memory and the local address is used to index the shared memory. This also guarantees that thread blocks are properly divided into stage blocks dependent on each stage's block height.

III. Results

To assess the performance of the BB network, we benchmarked against two GPU-based algorithms, bitonic sort [3] and radix sort [4], as well as a CPU-based algorithm (std::sort). We generate arrays of randomly generated floats of length 2^n with $n \in [10, 11, \ldots, 25, 26]$, and we report the time to sort (Fig. 4(a)) and the sorting throughput (Fig. 4(b)).

The algorithms are benchmarked on a single "NVidia GTX Titan Xp" which is based on Pascal architecture, with a total of 30 Streaming Multiprocessors, and 49kB shared memory per block and 12 GB global memory with a bandwidth of 547 GB/s. Each data point is calculated for 10 times and averaged to reduce unpredictability. The input dataset are randomized floating point numbers which could be positive or negative. Also, another test on the "Intel Xeon CPU E5-2603" with the built in std::sort algorithm was benchmarked. The Sorting algorithm on the CPU outperforms all GPU implementations in short-sized arrays this is because of very good caching abilities of the CPU on smaller datasizes.

Given the size of shared memory (49kB), we are restricted to comparing and routing 12k floats (occupying 48kB) per SM. For portions of the network that span more than 12k floats, we utilize global memory. The shared memory implementation faced a problem while trying arrays larger than 2^{13} elements and that's why in the graph above, we are depicting only a section of the possible results for the shared memory implementation. Looking at the trail, we surpassed the radix sort algorithm and obviously the global memory implementation Also, we expect this trend to continue until arrays with 2^{15} elements where we need to rely more on the global memory and performance will be similar to the global memory implementation.

A. Repository

The code for the BB network and the benchmark sorting algorithms is availble in the following Github repository.

IV. Discussion

Complexity and Scalability Analysis. The number of stages in the BB network is

$$n_{stages} = \left(\sum_{i=1}^{\log N} i \right) \propto O\left[(\log N)^2 \right]. \qquad (1)$$

this means that if we double the number of elements in the dataset from N to 2N, the depth of the BB network will be incremented by $\log N + 1$.

Why use the BB network/sorting networks? For large arrays ($n \geq 16$), radix sort is consistently faster than the BB network and bitonic sort. In what situations would BB network be preferable?

- **Small datasets** Assuming the size of the arrays to be sorted is small, the BB network has similar performance to radix sort. This is confirmed from the Results that we have obtained. As we can see, Bitonic Sort outperforms

Radix Sort in less than 2^{16} elements. This is due to the fact that in the Bitonic Sort algorithm there are no more accesses to the global memroy as the data is already loaded in the shared memory of the deice. Whereas, in the Radix Sort algorithm there are new arrays allocated on the global memory for categorizing.

- **More Predictable**: The Bitonic Sort and Batcher-Banyan Sort algorithms are more predictable than Radix Sort algorithm in terms of timing. Despite the data in Fig. III being averaged over 10 trials, the time and throughput of Radix Sort do not change predictably as n increases.
- **Dataset Agnostic**: The sorting networks, Bitonic and Batcher-Banyan included, are useful in the sense that the routing in the network is predefined and won't alter with the change in the values. This predictability stems from the fixed number of stages with a fixed routing pattern of the sorting networks [5]. This in contrast to the Radix sort algorithm where the sorting radices will not remain the same with different data.

Bottlenecks and Improvements The biggest challenges in our implementation is how we could map the comparison operations to threads inside the blocks and how to utilize shared memory within a block of threads. If we take a look at the global Batcher-Banyan network implementation, we are bottle-necked by the global memory bus speed. For the global memory model, the routings and assigning jobs seem to be working well, but we are not using memory efficiently. Especially after 2^{16} elements when we utilize the whole threads and blocks, we observe that the throughput is almost flat. This is the same for Bitonic Sort, but it plateaus at a higher throughput.

Another challenge that we faced in implementing our algorithm in the shared memory was the handling the data in the shared memory block. If the data isn't copied back to global memory after each stage, some conflicts might happen will processing the data in the next levels. In the future work, we can implement the Batcher-Banyan algorithm on the shared memory for all the elements with switching to global memory when necessary.

The problem that we are trying to address is important in the sense that the Batcher-Banyan network is a multistage interconnect network comprised of smaller switching elements and the fixed network topology helps in sorting elements in a mid-sized array, and to the best of our knowledge the Batcher-Banyan Network hasn't been implemented on GPU. One improvement to this implementation will be to breakdown the larger dataset to batches of mid-sized arrays and pass them to the Batcher-Banyan netowrk for sorting. Finally a merging algorithm on CPU can join the batches together to yield the final sorted dataset.

V. Conclusion

In this project, we demonstrated a Batcher-Banyan network that sorts an array of $N = 2^n$ elements. We discussed how to divide thread blocks between different regions of the network such that they could operate on shared memory, and we noted the limitations in using shared memory based on the 'height' of the routing topologies. We compared other methods on GPU and one on CPU with two of our implementations, on using global memory and one using shared memory. We learned that it's important to have a defined methodology of distributing work between threads and blocks evenly as possible, and handling the memory such that we face no conflicts. Also, we found out that the compute power is abundant and it's very hard to utilize it, as we are usually bound to memory speeds. Since using shared memory is comparable to caching, if the GPU designers were to remove the L2 cache in the device and add more shared memory to each SM, the programmer will have more flexibility in using the shared memory.

References

[1] M. J. Narasimha, "The batcher-banyan self-routing network: universality and simplification," *IEEE Transactions on Communications*, vol. 36, no. 10, pp. 1175–1178, Oct 1988.

[2] M. Yang and G. Ma, "Enhanced partially self-routing algorithm for controller benes networks," United States Patent (#5,940,389), August 1999.

[3] "Cuda toolkit documentation - sortingnetworks - cuda sorting networks," https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-sorting-networks, November 2019.

[4] N. Bell and J. Hoberock, *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2011.

[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01012.x