Efficient Batched Predecessor Search in Shared Memory on GPUs

Ben Karsin
karsin@hawaii.edu

Henri Casanova
henric@hawaii.edu

Nodari Sitchinava
nodari@hawaii.edu

Information & Computer Sciences Dept., University of Hawai'i at Mānoa
Honolulu, Hawai'i, USA

***Abstract*—Many-core Graphics Processing Units (GPUs) are being used for general-purpose computing. However, due to architectural features, for many problems it is challenging to design parallel algorithms that exploit the full compute power of GPUs. Among these features is the memory design. Although the issue of coalesced global memory access has been documented and studied extensively, another important architectural feature is the organization of shared memory into banks. The study of how bank conflicts impact algorithm performance has only recently begun to receive attention.**

In this work we study the predecessor search algorithm and the effects of bank conflicts on its execution time. Via complexity analysis we show that bank conflicts cause significant loss in parallelism for a naive algorithm. We then propose two improved algorithms: one that eliminates bank conflicts altogether but that uses a work-inefficient linear search, and one that is work-optimal but that experiences a limited number of bank conflicts. We develop GPU implementations of these algorithms and present experimental results obtained on real-world hardware. These results validate our theoretical analysis of the naive algorithm and allow us to assess the performance of our algorithms in practice. Although both our improved algorithms outperform the naive algorithm, our main experimental finding is that our conflict-limited algorithm provides a larger performance gain.

I. Introduction

Graphics Processing Units (GPUs) are increasingly used for general-purpose computing [1]–[4]. Modern GPUs host thousands of compute cores, enabling them to achieve enormous performance gains for easily parallelizable problems when compared to CPUs. The *parallel random access memory (PRAM)* model [5] is a classical model for parallel algorithm design. It consists of processors and a single memory shared by all processors. The execution is performed in lockstep and each processor can access any address in memory in unit time concurrently with other processors. Due to the model's simplicity, a large number of PRAM algorithms have been designed [5], [6]. PRAM algorithms are good starting points for GPU implementations, but it is difficult for them to leverage the full computational power of GPUs due to specifics of the GPU architecture. In particular, features of the hierarchical memory design (e.g., the use of shared memory, coalesced global memory access, shared memory bank conflicts) violate the unit-time access assumption of the PRAM model, making the PRAM execution time prediction inaccurate on GPUs in practice. Consequently, these architectural features must be taken into account for designing efficient parallel GPU algorithms.

Searching is a fundamental operation that is at the core of a number of algorithms. In this paper we study the effects of bank conflicts on the *batched predecessor search* (BPS) problem:

Definition 1.1: Given a list $\mathcal{K}$ of K keys $\mathcal{K}[0], \mathcal{K}[1], \cdots, \mathcal{K}[K-1]$, sorted in non-decreasing order, and a set $\mathcal{Q}$ of Q queries, the BPS problem asks to find for each $q \in \mathcal{Q}$ the largest i, such that $\mathcal{K}[i] \leq q$.

BPS can be solved optimally in the standard RAM model in $O(Q \log K)$ time by running the classical binary search algorithm on $\mathcal{K}$ for each query $q \in \mathcal{Q}$. In the p-processor CREW PRAM model[1], a straightforward and optimal parallelization consists in performing each binary search in parallel, achieving $O(\frac{Q}{p} \log K)$ time.[2] Performing a single predecessor search using binary search is also optimal on the GPU, but for a batch of queries the memory design on the GPU raises challenges.

The issue of coalesced access to global memory has been documented and studied extensively [2], [7], [8]. In this work we focus instead on the effects of shared memory bank conflicts. When the input is stored in the GPU's shared memory, we show that bank conflicts significantly decrease the parallelism of the straightforward parallel binary search algorithm. We construct a family of worst-case input queries for which it incurs a large number of bank conflicts and, therefore, always achieves sub-optimal performance. We propose modifications to the binary search algorithm to achieve optimal performance even in the presence of bank conflicts. More specifically, our contributions are:

- Via complexity analysis we show that, in the worst case, the straightforward parallel binary search algorithm

This work was partially supported by the National Science Foundation under Grant No. 1533823.

[1]The CREW PRAM model allows concurrent reads from, but not concurrent writes to the same address in memory by multiple processors in a single time unit.

[2]To avoid cumbersome notation, throughout the paper the fractions within runtimes are rounded up, i.e., $\frac{x}{y} = \left\lceil \frac{x}{y} \right\rceil$.

cannot exploit the full parallelism of a GPU due to bank conflicts.
- Since the straightforward algorithm experiences poor performance due to shared memory bank conflicts on the GPU, we propose a conflict-free algorithm.
- A drawback of the conflict-free algorithm is that it relies on a work-inefficient linear search. We thus propose another algorithm that incurs a bounded number of conflicts but that is work-optimal.
- We present experimental results that validate our theoretical analysis, show strong correlation between experimental execution time and the number of bank conflicts, and that our conflict-limited algorithm achieves good performance in practice.

The rest of the paper is organized as follows. Section II provides an overview of the GPU architecture. Section III reviews related work. Section IV provides a detailed analysis of the straightforward parallel binary search algorithm and introduces our two improved algorithms. Section V describes implementation details and discusses experimental results. Finally, Section VI concludes with a summary of our findings and future research directions.

II. GPU Architecture

In this section we provide an overview of modern GPU architectures, highlighting features that are germane to this work. See standard references [3], [4] for more details.

Modern GPUs comprise hundreds or even thousands of identical physical processors, or cores. These cores are organized hierarchically, by combining a group of cores into *streaming multiprocessors* (SMs). Each SM contains small, but fast *shared memory*, which is available to the threads running on that SM. A larger, but slower global memory is shared among all SMs (see Figure 1). In addition, each thread has access to a limited number of private registers.

A. Thread organization

To hide the latency of memory accesses, each GPU supports execution of more threads than physical cores. A typical GPU can support thousands of threads, each identified by a unique ID. To manage such large numbers, groups of w threads, called *warps*, are executed in lockstep in *single instruction, multiple data (SIMD)* fashion. Parallel SIMD execution suffers from "branch divergence" if the threads in a warp follow different execution paths (e.g., due to conditionals), in which case thread execution is serialized.

Threads are organized into warps deterministically based on thread ID (a warp consists of w threads with contiguous integer IDs, starting with ID 0). The programmer has no control over which warps will be scheduled on which SMs. The only control available to the programmer is via organization of groups of warps into *thread blocks* or *cooperative thread arrays (CTAs)* [8], [9], with all threads of a thread block guaranteed to be scheduled on the same SM. This

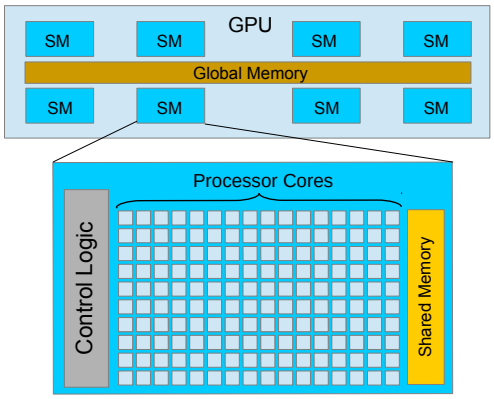

Figure 1: Most modern GPUs consist of Streaming Multiprocessors (SMs), each of which has its own control logic, shared memory, and many processor cores.

organization allows all threads of a thread block to share the data stored in the shared memory of a SM.

Thanks to fast context switching among warps, better memory latency hiding is achieved by increasing the total number of threads, p, up to the point when the memory bandwidth is fully saturated. Thus, when designing algorithms for GPUs, we want them to scale with the number of threads up to the largest possible value of p. However, the maximum *effective* value for p (the one that causes the memory bandwidth to saturate) is dependent on the hardware and can be determined by the programmer experimentally for the specific GPU.

B. Memory Hierarchy

Each level of GPU memory has different latency, throughput, access scope, and optimal access pattern. Although various memory and caching options are possible, in this work we focus on the pervasive global memory and shared memory levels (we do not consider registers explicitly as they are private to threads).

Global memory is the only way to communicate between threads of different thread blocks, but it is at least an order of magnitude slower than the other types of memory [3], [7]. Consequently, to approach peak performance its use must be limited. Research focused on modeling the GPU memory hierarchy [2], [8]–[11] has demonstrated that, to obtain close to peak theoretical throughput, global memory access must be *coalesced*. Memory access is coalesced when, during a SIMD operation, the threads of a warp access consecutive elements in global memory. In this case, all threads obtain an element in a single access concurrently, whereas non-coalesced access requires that each thread perform a separate access in a serial manner. See [7], [10] for an in-depth discussion of this topic.

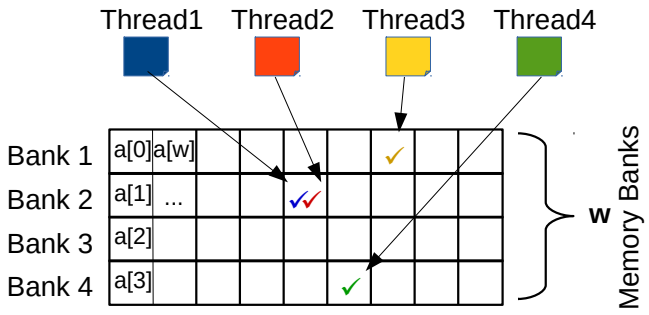

Figure 2: 4 threads accessing shared memory simultaneously without bank conflicts. Threads 1 and 2 can access the same memory cell simultaneously thanks to data multicast.

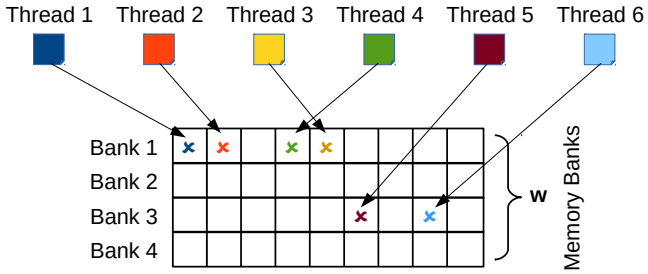

Figure 3: Bank conflicts occur when multiple threads access different memory cells in the same bank. This example corresponds to a 4-way bank conflicts since 4 distinct cells are being accessed in a single bank (bank 1).

Shared memory is a smaller, faster memory that, as seen in Figure 1, is private to each SM. We denote the size of shared memory on each SM by M. Like for global memory, access to shared memory must follow a certain pattern to obtain peak throughput. The shared memory of each SM is implemented as a collection of *memory banks*. We assume w memory banks – the same as the number of threads in a warp. This is typical on modern GPUs because more threads than memory banks would lead to bank conflicts (see below), and too few threads would waste memory bandwidth. Each memory bank thus contains $\frac{M}{w}$ *cells* and shared memory can be viewed as a $w \times (M/w)$ matrix with memory banks forming its rows. Linear data arrays are laid out in column-major order in this matrix.

During a SIMD instruction, as long as threads within a warp access either the *same address* (in which case the data is multicasted to the threads) or *distinct banks*, the data is delivered to the threads concurrently (see Figure 2). However, if during a SIMD instruction threads of a warp attempt to access different addresses within the same bank, a *bank conflict* occurs, and the memory accesses are serialized. Thus, if during a SIMD instruction, x is the maximum number of threads accessing distinct addresses within the same bank, we say that there is an x*-way bank conflict*, and the instruction takes x times longer to execute (see Figure 3). Since bank conflicts serialize memory accesses, they must be avoided if one hopes to approach peak performance.

III. Related Work

Several authors have designed efficient GPU implementations for many classical problems, including parallel scan and prefix sums [9], [12]–[14], sorting [8], [15], [16], graph algorithms [17]–[19]. Since there many results on searching, in this section we only focus on recent work relevant to the problem of searching on GPUs [20]–[24].

For *dynamic datasets* indexing is the most common optimization technique. Kaczmarski [20], [21] studies the construction of the B$^+$-tree data structure to index data in GPU global memory. By focusing on the B$^+$-tree, the author takes advantage of coalesced global memory accesses. Similarly, Shekhar [22] proposes GPU-efficient global memory data structures designed to improve performance of the IP lookup operation. Kim et al. [23] create a hybrid CPU/GPU data structure to achieve high peak query throughput. Soman et al. [24] address the limited memory on the GPU by looking at compressing search tree data structures.

For *sorted static datasets* that are small enough to fit in GPU shared memory, binary search remains one of the fastest practical ways of searching for an element due to small constant factors in the time complexity of the algorithm. As a result, several GPU sorting implementations successfully utilize binary search as a subroutine [15], [16], [25]. In the process of developing a GPU Sample Sort, Leischner et al. [15] eliminate branch divergence during SIMD execution of a binary search within a warp by developing an implementation of the search that uses predicates to eliminate conditional branches. But their implementation does experience loss of parallelism due to shared memory bank conflicts.

Researchers have recently started to pay attention to shared memory bank conflicts and on how they lead to loss of parallelism [10], [11], [26], [27]. These works incorporate the number of bank conflicts in execution time analyses or propose algorithms that are bank conflict free. In this work we continue the study of the effects of bank conflicts by analyzing and improving the binary search algorithm. Our implementations also avoid conditional branches, thus eliminating any loss of parallelism due to branch divergence in the same way as in Leischner et al. [15].

IV. Batched Binary Search on GPU

There are many small variations on how to implement binary search, each leading to a different worst-case input for the algorithm. We define here a simple version for concreteness of exposition, noting that our analysis can be performed for any of the variations. Algorithm 1 shows pseudo-code for the standard binary search algorithm for a single query q. The *min* and *max* functions are used to prevent out-of-

Algorithm 1: Pseudo-code for binary search.

```
BinarySearch(𝒦, q):
index = ⌊K/2⌋
δ = ⌈K/4⌉
for ⌈log K⌉ times do
    if q ≥ 𝒦[index] then
        index = min(index + δ, K − 1)
    else
        index = max(index - δ, 0)
    δ = ⌈δ/2⌉
end
if q < 𝒦[index] then
    index = index - 1
return index
```

bounds memory accesses, but in our implementations we instead pad the input to eliminate this extra computation. The details of memory padding are explained in Section IV-B.

Let *Parallel Binary Search (PBS)* be the straightforward parallel solution to the BPS problem by running Algorithm 1 for each query $q \in \mathcal{Q}$ concurrently.

In the p-processor CREW PRAM model, the PBS algorithm takes optimal $\Theta(Q \log K)$ work and $\Theta\frac{Q}{p} \log K)$ time. However, if data $\mathcal{K}$ is stored in shared memory of the GPU, the serialization of memory accesses due to bank conflicts violates the PRAM assumption that access to each memory location takes unit time. Thus, the above bound is not representative for the worst-case runtimes on GPUs (see Section IV-A).

We assume that $\mathcal{K}$ fits in shared memory. If it does not, then one must load portions of it into shared-memory and batch the searches. But an efficient binary search in shared memory, the focus of this work, is still needed for each batch. For simplicity of exposition we also assume that w is a power of 2. Although this is the case for most modern GPUs, our results extend to other values of w via straightforward, but tedious analysis.

A. Analysis of the PBS algorithm on GPUs

Let $T_{PBS}(Q, K, p, w)$ denote its execution time for Q queries and K keys, on a GPU with w shared memory banks and p threads organized into warps running w threads at a time. Recall that p is no more than the the number of threads required to saturate the shared memory bandwidth (see Section II-A).

Theorem 4.1: For every sorted sequence $\mathcal{K}$, $|\mathcal{K}| = K \geq 2w^2$, there exists a set of queries $\mathcal{Q}$, $|\mathcal{Q}| = Q \geq p$, for which

$$T_{PBS}(Q, K, p, w) = \Omega\left(w \cdot \frac{Q}{p} \log \frac{K}{w^2}\right).$$

This theorem states that, in the worst case, the PBS algorithm cannot fully exploit the parallelism of the GPU (compare to the PRAM complexity $\Theta(\frac{Q}{p} \log K)$). The proof of this theorem relies on the following lemma:

Lemma 4.2: For every sorted sequence $\mathcal{K}$, $|\mathcal{K}| = K \geq 2w^2$, there exists a query set $\mathcal{Q}$, $|\mathcal{Q}| = Q = w$, for which the time required to run the PBS algorithm using w threads of a warp is at least

$$T(K, w) = \Omega\left(w \cdot \log \frac{K}{w^2}\right).$$

Proof: For simplicity, we construct the query set $\mathcal{Q}$ so that each query $q \in \mathcal{Q}$ is an element of $\mathcal{K}$.

Recall from Section II-B that with w memory banks, a bank conflict occurs when separate threads within a warp access shared memory cells $\mathcal{K}[i]$ and $\mathcal{K}[j]$, such that $i \neq j$ and $i \equiv j \pmod w$. Since the PBS algorithm is deterministic, for a given sorted sequence $\mathcal{K}$, the access pattern of each thread is a function of the query $q \in \mathcal{Q}$ that the thread is processing. More specifically, in the k-th iteration of the binary search loop ($k = 1, 2, \ldots, \log K$), a thread may access one of 2^{k-1} possible memory addresses. Let $r = 2 \log w + 1$. Then for the w queries, in the r-th iteration, there are w^2 memory addresses that each of w threads may be accessing. Therefore, by the pigeonhole principle, among these w^2 addresses, there exists a subset of w *distinct* memory addresses that reside in the same memory bank. We can choose $\mathcal{Q}$ so that these w distinct memory addresses are accessed by the w threads in the r-th step, thus resulting in a w-way bank conflict.

Consider an arbitrary pair of threads t_i and t_j within the same warp, running on the above input $\mathcal{Q}$. Let them access memory addresses i and j, respectively, in the r-th step. Since i and j are distinct, $|i - j| \geq K/2^r$. This implies that (1) we can set each query to one of $K/2^r$ possible entries of $\mathcal{K}$ and still cause w-way bank conflicts in iteration r; and (2) any choice of these values will result in each thread accessing distinct addresses in each of the final $\log(K) - r$ iteration (including iteration r). Of these $K/2^r$ choices, we choose each query so that in the rest of the algorithm all threads take the same branch of the `if` statement.

Let the addresses accessed by t_i and t_j in any iteration $k \geq r$ be $i + \delta_k$ and $j + \delta_k$, respectively. Since every pair of threads accesses the same bank in round r, $i \equiv j \pmod w$, and it follows that $i + \delta_k \equiv j + \delta_k \pmod w$.

Thus, for $\log K - r + 1$ rounds, our input $\mathcal{Q}$ causes w-way bank conflicts, resulting in running time of at least $\Omega(w \cdot (\log K - r + 1)) = \Omega\left(w \cdot \log \frac{K}{w^2}\right)$. ∎

Proof (of Theorem 4.1): A typical (deterministic) GPU implementation breaks down Q queries into Q/w groups of w queries each, and each group is processed in SIMD fashion by one of the p/w warps. Note, that only p/w out of Q/w groups can be processed simultaneously. By Lemma 4.2, each such group takes $\Omega\left(w \cdot \log \frac{K}{w^2}\right)$ time to

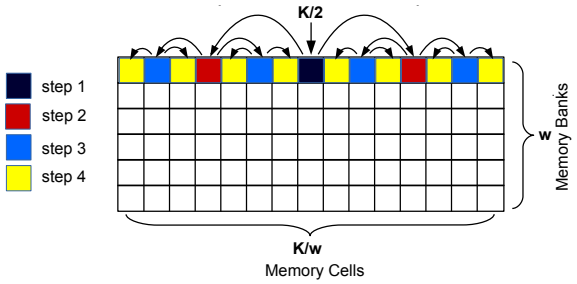

Figure 4: Worst-case example for the first $\log K - \log w$ iterations of the PBS algorithm, when K is a multiple of w^2 (Corollary 4.1).

process. Thus, the total execution time is

$$T_{PBS}(Q,K,p,w) = \Omega\left(\frac{Q/w}{p/w}\cdot\left(w\cdot\log\frac{K}{w^2}\right)\right) = \Omega\left(w\cdot\frac{Q}{p}\log\frac{K}{w^2}\right)$$

■

It is well known that inputs that are multiples of w tend to be very bad in terms of bank conflicts [14], [15], [28]. The following corollary shows a slightly stronger lower bound when K is a multiple of w^2:

Corollary 4.1: For every sorted sequence $\mathcal{K}$, such that $|\mathcal{K}| = K \geq w^2$ and $K \equiv 0 \pmod{w^2}$, there exists a query set $\mathcal{Q}$, $|\mathcal{Q}| = Q \geq p$, for which

$$T_{PBS}(Q,K,p,w) = \Omega\left(w\cdot\frac{Q}{p}\log\frac{K}{w}\right).$$

Proof: If $K \equiv 0 \pmod{w^2}$, then all possible memory addresses that each thread might access in the first $r' = \log w + 1$ iterations reside in memory bank 0 (see Figure 4). Thus, we can find queries that cause w-way bank conflicts starting from the iteration $r' = \log w + 1$ instead of iteration $r = 2\log w + 1$ proven in Lemma 4.2. The rest of the proof is similar to the proofs of Lemma 4.2 and Theorem 4.1. ■

When K is a multiple of w^2, an example set of Q queries that causes worst-case performance is when the i-th thread of a warp ($0 \leq i < w$) searches for query $q = \mathcal{K}[i\cdot\frac{K}{w} + C]$, for any $0 \leq C < \frac{K}{w}$ fixed for the threads within that warp.

B. Conflict-Free Search

In this section we present a modified PBS algorithm, which we call *Parallel Binary Search - Conflict Free (PBS-CF)*, that is free of shared memory bank conflicts. To do so, we divide the PBS algorithm into two stages, and for each stage we design a solution to eliminate all bank conflicts. *Stage 1* consists of the first $\log K - \log w$ iterations, during which $\delta \geq w$ (see Algorithm 1). *Stage 2* consists of the remaining iterations. The access pattern and reason for bank conflicts differ between the two stages, so we must develop a different conflict-free solution for each pattern.

As seen in the proof of Lemma 4.2, after $2\log w + 1$ iterations, in the worst case the PBS algorithm incurs w-way bank conflicts. Since any thread can access any memory bank, after enough iterations bank conflicts are unavoidable. To remedy this situation we must ensure that, at each iteration, threads within a warp access different memory banks. For illustration purposes, let us consider the special case where K is a multiple of w^2, resulting in the access pattern illustrated in Figure 4. In this case, for the first $\log K - \log w$ iterations, δ is always a multiple of w, causing the same memory bank to be accessed at every iteration. If we initially assign each thread a separate bank in this case, we do not need to change the δ calculation and no bank conflicts will occur. This is accomplished by giving each thread an initial offset equal to its $threadID$ within its warp. The resulting access pattern is illustrated in Figure 5.

To extend the above approach to the general case for any K, we alter the δ calculation. This can be achieved by enforcing that δ be a multiple of w at each iteration, so that each thread accesses only its assigned memory bank and no conflict can occur. Therefore, we round the δ calculation down to the nearest multiple of w at each iteration. Algorithm 2 shows pseudocode for stage 1 of the PBS-CF algorithm. As mentioned regarding Algorithm 1, we pad the key list $\mathcal{K}$ to eliminate the need for checking bounds. Specifically, we add w elements at the beginning and end of $\mathcal{K}$, with padding values of $-\infty$ and ∞, respectively.

Algorithm 2: Pseudo-code for PBS-CF, stage 1.

```
PBS-CF-1(K, q):
offset = threadID mod w
index = ⌊K/2⌋ + offset
δ = ⌈K/4⌉
δ -= δ mod w
while δ ≥ w do
    if K[index] ≥ q then
        index = index + δ
    else
        index = index - δ
    δ = ⌈δ/2⌉
    δ -= δ mod w
end
return index
```

Note that the offset can be pre-computed before the computation begins. Furthermore, for the special case where K is a multiple of w^2 we can eliminate the extra rounding computation. As illustrated in Figure 5, this solution forces every thread within a warp to exclusively utilize its own memory bank, eliminating all bank conflicts from stage 1.

Once $\delta < w$, the symmetry that enables stage 1 to be completely conflict-free is lost. At this point, the memory cells accessed by a thread can change memory banks, leading to the possibility of bank conflicts. Therefore, we

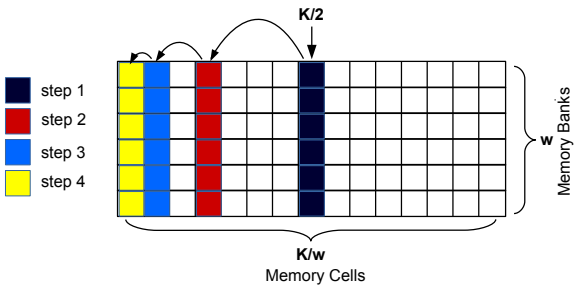

Figure 5: Illustration of the shared memory access pattern for stage 1 of the PBS-CF algorithm ($\delta \geq w$).

must utilize a different search method to ensure there are no bank conflicts for stage 2. We note that w is quite small (typically $w = 32$), so a linear search seems like a reasonable choice. Conveniently, since the threads end stage 1 accessing a different memory bank each, we need to continue only from that position. In lockstep, each thread accesses the next key ($K[i+1]$), until all w banks have been visited and the query is found. This process requires w shared memory accesses, but is completely bank conflict free. Algorithm 3 shows pseudo-code for stage 2 of the PBS-CF algorithm.

Algorithm 3: Pseudo-code for PBS-CF, stage 2.

```
PBS-CF-2(𝒦, q):
if q < 𝒦[index] then
    index -= w
for i = 0 to w − 1 do
    if 𝒦[index+i] ≤ q then
        result = index + i
end
return result
```

Note that stage 2 of the PBS-CF algorithm begins by checking if the target is larger or smaller than the starting point, $\mathcal{K}$[index]. Depending on this check, we either search the w elements smaller or larger than $\mathcal{K}$[index] to find the index of q. Because we pad $\mathcal{K}$, this operation cannot result in out-of-bounds memory accesses.

Since both stages are guaranteed to be bank conflict free, the overall execution time of the PBS-CF algorithm on w queries using a single warp will be $\Theta(\log \frac{K}{w} + w)$, resulting in overall execution time on Q queries $\Theta(\frac{Q}{p} \log \frac{K}{w} + \frac{Qw}{p})$.

C. Conflict-Limited Search

Because the PBS-CF algorithm is completely conflict-free, it always requires the same number of memory accesses per query. However, during stage 2 it is not work optimal, requiring $O(w)$ memory accesses (and operations) to search w elements. The PBS algorithm, on the other hand, is work optimal, requiring only $\Theta(\log w)$ iterations to search the last w elements. However, in the worst case, it incurs w-way bank conflicts at each of these iterations, leading to $O(w \log w)$ total memory accesses, a factor $\Theta(\log w)$ more than PBS-CF. In this section we implement a hybrid algorithm *PBS-CL* (Parallel Binary Search - Conflict Limited) with the goal to minimize the number of bank conflicts while remaining work optimal.

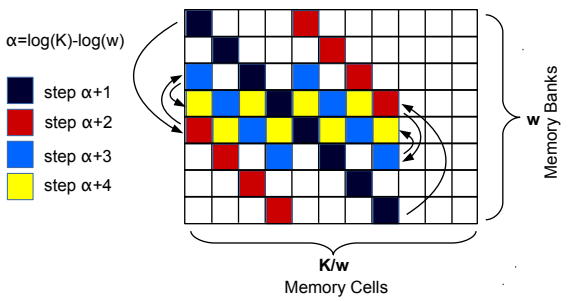

Figure 6: Illustration of the shared memory access pattern for stage 2 of the PBS-CL algorithm ($\delta < w$), for a worst-case example.

PBS-CL uses the same conflict-free stage 1 (the first $\log K - \log w$ iterations) as PBS-CF. However, it uses a hybrid approach for stage 2 that introduces few bank conflicts while keeping the number of operations the minimal $\Theta(\log w)$. Recall that, for the PBS-CF algorithm, each thread accesses a different memory bank during stage 1 (specifically, each thread accesses bank i where $i \equiv threadId \pmod w$). Upon starting stage 2, the offset between each thread is preserved, causing them to each access a different bank. If we then perform a binary search, with all threads utilizing the same δ value, we can preserve these offsets and reduce the number of bank conflicts while remaining work optimal, requiring only $\Theta(\log w)$ iterations. As with PBS-CF, we pad $\mathcal{K}$ to prevent out-of-bounds memory accesses.

Figure 6 illustrates which cells may be accessed at each iteration of stage 2 of the PBS-CL algorithm, in the *worst case*. For the first iteration of stage 2, $\delta = \frac{w}{2}$, and, since w is a power of 2 and

$$(threadId + \frac{w}{2}) \equiv (threadId - \frac{w}{2}) \pmod w,$$

there will be no conflicts (and thus only 1 parallel access). For the second iteration, however, there is potential for at most a 2-way conflict. For the third iteration, the worst case would have one 4-way conflict, and so on. Thus, as illustrated in Figure 6, the worst-case number of memory accesses for stage 2 is $\sum_{i=0}^{\log w - 1} 2^i = w - 1$.

Since stage 1 is bank conflict free, the overall execution time of the PBS-CL algorithm on w queries using a single warp is $O(\log \frac{K}{w} + w)$, resulting in overall execution time $O(\frac{Q}{p} \log \frac{K}{w} + \frac{Qw}{p})$ for Q queries, which is the same as for PBS-CF. However, PBS-CF requires this much time for

every input, while PBS-CL will result in no bank conflicts on some inputs, taking only $O(\log w)$ steps in stage 2, and resulting in overall execution time as low as $O(\frac{Q}{p} \log K)$ on some inputs. Therefore, we claim that PBS-CL should outperform both PBS and PBS-CF in practice.

V. Experimental Results

A. Environment and Methodology

While our algorithms are generalizable to most modern GPU hardware, we present results obtained on our specific hardware and software environment, an nVidia GTX770 graphics card with the following specifications [29]:

- Number of SMs: 8
- Number of cores per SM: 192
- Number of threads per warp: 32
- Clock speed: 1046 MHz
- Global memory size: 4GiB
- Global memory peak throughput: 224.1 GiB/sec
- Shared memory size per SM: 48KiB
- Number of shared memory banks: 32
- Shared memory cell size: 64 bits

We implement the algorithms in C++ with the nVidia CUDA (Compute Unified Device Architecture) API version 7 [3], using the Visual Studio 2013 IDE on Windows 8. We employ full compiler optimization (-Ox) for all experiments that measure execution time.

In all experiments we first generate the list $\mathcal{K}$ in shared memory, sorted in non-decreasing order. We then generate Q queries randomly chosen from $\mathcal{K}$, stored in global memory. Queries are divided evenly among all threads in a striped manner to allow for coalesced global memory access. Each thread reads a query, performs the search, and writes the resulting predecessor result back to global memory, in a coalesced manner. Each experiment is repeated 100 times and we compute average execution time. Since the measured standard deviation is at most 1 millisecond we omit error bars from figures.

When running experiments on the GPU, we must select the total number of thread blocks and the number of threads per thread block. We found that it is sufficient to pick a number of thread blocks large enough to provide work to all SMs (and so that there are enough threads to utilize all the cores). All our experiments are with 256 thread blocks. The number of threads per block defines the size of thread groups that share the same partition of shared memory. Since our algorithms utilize increasingly large amounts of shared memory as we increase K, each SM may only be able to run very few thread blocks at a time. As a result, we find that using the maximum number of threads per block (i.e., 1,024) leads to the best performance.

B. Bank Conflicts

We wish to validate the worst-case estimate of the number of memory accesses for the PBS algorithm in Section IV.

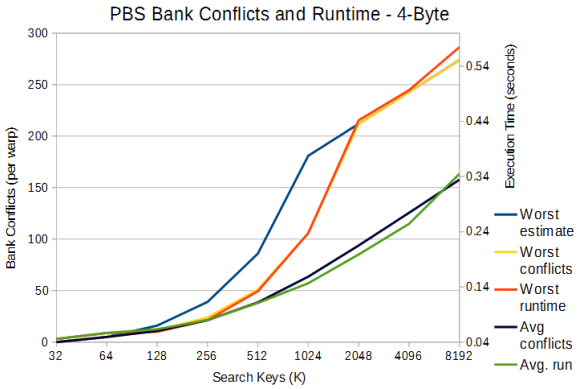

Figure 7: Estimated and measured bank conflicts (left vertical axis) and average execution time (right vertical axis) for 4-Byte keys and 500 million queries vs. K, with both worst-case and randomly generated queries for the PBS algorithm.

We cannot generate a worst-case $\mathcal{Q}$ for any arbitrary K since the distribution of memory accesses among memory banks depends on K and w. However, if K is a multiple of w^2 a worst-case set of queries $\mathcal{Q}$ is obtained by having each thread i within a warp query $\mathcal{K}[i \cdot \frac{K}{w} + C]$, for any $0 \leq C < \frac{K}{w}$ (see Section IV). For this query set, the number of bank conflicts incurred by PBS increases exponentially for $\log w$ iterations (Figure 4), after which every iteration results in a w-way bank conflict. We can thus estimate the total number of memory accesses as:

$$\begin{aligned} A_{PBS} &= \sum_{i=0}^{\log K - 1} \max\{2^i, w\} \\ &= 1 + 2 + 4 + \cdots + 2^{\log w - 1} + w(\log K - \log w) \\ &= w(\log K - \log w + 1) - 1 \end{aligned}$$

Furthermore, this is the worst-case input for any $K \geq w$ that is a power of 2. Extending this to smaller K values allows for scenarios where there are fewer than w cells per bank. However, if $K < w^2$, the maximum number of bank conflicts per iteration is limited to $\frac{K}{w}$, since there are fewer than w total memory locations per bank.

We use the nVidia Nsight performance analysis tool (Visual Studio Edition 4.6) [30] to measure bank conflicts empirically. The above estimate, however, is for the total *memory accesses* per warp, which includes the first access at each cycle. Bank conflicts, on the other hand, are measured by Nsight as the total *additional* memory accesses. Therefore, subtracting $\log K$ from our estimate gives us the estimated number of bank conflicts. We measure the average number of bank conflicts for both worst-case and randomly generated query sets.

Figure 7 plots the number of bank conflicts (left vertical axis) and the average execution time (right vertical axis) vs. the number of search keys (K) for 4-Byte keys (the `float`

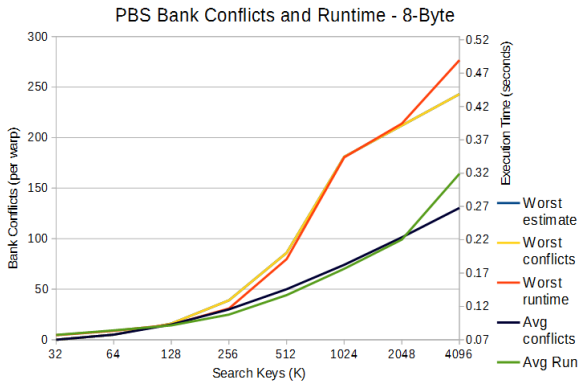

Figure 8: Estimated and measured bank conflicts (left vertical axis) and average execution time (right vertical axis) for 8-Byte keys and 500 million queries vs. K, with both worst-case and randomly generated queries for the PBS algorithm.

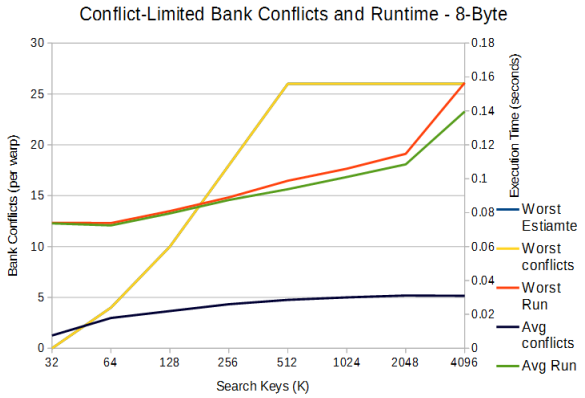

Figure 9: Estimated and measured number of bank conflicts (left vertical axis) and average execution time (right vertical axis) for 8-Byte keys and 500 million queries vs. K, with both worst-case and randomly generated queries for the PBS-CL algorithm. The worst-case estimate is not visible because it exactly matches the measured number of conflicts.

datatype), for both worst-case and randomly generated sets of $Q = 500$M queries. The trend of the average execution time closely follows that of the number of conflicts as K varies. However, our conflict estimate in Figure 7 does not match the measured worst-case number of conflicts. To determine the cause of this mismatch we examine the measured number of conflicts for each iteration of the search. For $K = w^2 = 1,024$, since there are enough cells per bank, the number of conflicts should exponentially increase until each iteration has a w-way conflict (which would be measured by Nsight as $w-1 = 31$). However, the measured result shows that we do not exceed 16 conflicts per iteration. It is not until $K = 2,048$ that one reaches the expected 31 conflicts per iteration. It turns out that on our GPU hardware each memory bank can be set to either 4- or 8-Byte mode, and in 4-Byte mode two values are stored in each cell. This, combined with the multicast feature, effectively doubles the number of memory banks when the stride between queries is only 32. When $K = 2,048$, the stride becomes 64 and we reach our expected maximum number of conflicts. To verify this explanation, Figure 8 shows results similar to Figure 7 but for 8-Byte (the `double` datatype) values. Expectedly, our estimate exactly matches the measured number of conflicts.

For both 4- and 8-Byte values, we see that both the random and worst-case execution times have trends that follow the number of bank conflicts. This indicates that the number of bank conflicts is a performance driver for the PBS algorithm, thus justifying the need to reduce this number by developing algorithms such as our PBS-CF and PBS-CL algorithms. Note, however, that the execution time sharply increases for the largest K value in Figure 8. This is attributed to inefficiencies in parallelism. When $K = 4,096$, with 8-Byte values, each thread block requires more than half of the total shared memory on each SM. Therefore, only one thread block can be resident at a time, preventing latency hiding and increasing synchronization overhead [1], [3]. Future GPUs will, undoubtedly, have larger shared memory sizes, alleviating this problem and allowing larger lists $\mathcal{K}$ to be searched with this method.

The results in Figures 7 and 8 are for the PBS algorithm, and an important conclusion from these results is that the number of bank conflicts is strongly correlated to the algorithm's execution time. While this cannot be the case for the conflict-free PBS-CF algorithm, one may wonder whether a similar observation holds for the PBS-CL algorithm. Figure 9 is similar to Figure 8 but shows results for the PBS-CL algorithm. As for the PBS algorithm, our estimate exactly matches the measured number of bank conflicts. Unlike for the PBS algorithm, however, the performance of the PBS-CL algorithm is not driven by bank conflicts. Furthermore, the average and worst-case performance differ by at most 5.5%, indicating that the PBS-CL algorithm is more resilient to the distribution of query values.

C. Algorithm Scalability

In this section we compare the execution time of our three algorithms (PBS, PBS-CF, and PBS-CL). Figures 10 and 11 show average execution vs. K for 4-Byte and 8-byte values ($Q = 500M$). For low values of K, i.e., with few search keys, PBS does not suffer from many bank conflicts (with 32 elements each bank has only 1 cell), so it performs as well as PBS-CL and better than PBS-CF. However, as K increases, its performance degrades quickly.

The results also show that while PBS-CF suffers from overhead in both experiments, it performs significantly worse with the 8-Byte values. We see an average overhead of 0.061 seconds and 0.153 seconds when compared with

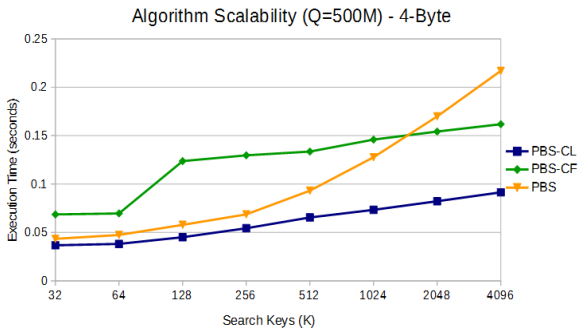

Figure 10: Average execution time vs. K for the PBS, PBS-CF, and PBS-CL algorithms, with 4-Byte keys and queries.

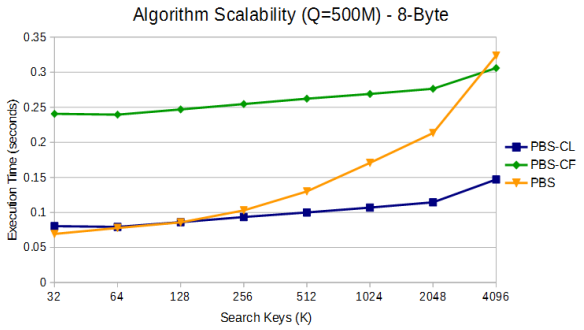

Figure 11: Average execution time vs. K for the PBS, PBS-CF, and PBS-CL algorithms, with 8-Byte keys and queries.

PBS-CL with 4-Byte and 8-Byte values, respectively. We postulate that this is due to the linear search component during stage 2 of the algorithm. With an 8-Byte value, the same number of elements can be accessed per cycle, but computation takes longer, making the algorithm more compute-bound. By contrast, the PBS-CL algorithm performs only $\log w$ computations during stage 2, allowing it to perform better for more computationally intensive data types. Overall, the PBS-CL achieves the lowest average execution times across the board.

D. Impact of Global Memory Accesses

The high relative latency of global memory access has motivated work on reducing accesses and maximizing throughput [1], [31]. Since our query set, $\mathcal{Q}$, resides in global memory (and we write each query result back), we investigate the impact of global memory accesses on the execution time of our algorithms. As discussed in Section IV, the PBS algorithm should run in $\Omega(w \cdot \frac{Q}{p} \log \frac{K}{w^2})$ time (in the worst case), and our improved algorithms in $O(\frac{Q}{p} \log \frac{K}{w} + \frac{Qw}{p})$ time. Therefore, we presume that these algorithms have high enough computational complexity to prevent them from being global-memory-bound. To verify this presumption, we measure the performance of a global memory 'read/write' baseline that simply reads, increments, and writes each query. Table I compares this baseline and our three algorithms using average execution time as well as two global memory metrics obtained from the nSight performance analysis tool: bandwidth and memory stalls. Bandwidth is the average utilized global memory bandwidth and memory stalls are the total % of cycles that are stalled due to 'memory dependencies'. Note that memory stalls include cycles stalled due to both global and shared memory, so the number of memory installs is an upper bound on cycles that are global-memory-bound.

Algorithm	Execution Time	Bandwidth	memory stalls
PBS	0.323 sec	23.76 GB/sec	1.79%
PBS-CF	0.306 sec	25.14 GB/sec	2.33%
PBS-CL	0.148 sec	48.94 GB/sec	5.48%
Read/write	0.051 sec	160.2 GB/sec	50.42%

Table I: Execution time and global memory statistics for each algorithm using 500M queries, 4096 8-byte shared memory keys. Bandwidth and memory stalls are obtained from the nSight performance analysis tool [30].

The results shown in Table I indicate that none of our three search algorithms is global-memory-bound, while the read/write baseline clearly is. The peak bandwidth for our hardware is 224.1 GB/sec, though in practice one can typically obtain around 70% of this peak [32]. Thus, our read/write baseline, obtaining 160.2 GB/sec (71.5% of the peak), is bound by global memory bandwidth. Our other algorithms, however, utilize much less global memory bandwidth. Furthermore, during the baseline experiment, over half of all cycles were stalled due to memory dependencies, while this caused less than 5.5% of cycles to stall when running our algorithms. We conclude that, while the execution time of the baseline read/write is relatively close to our algorithms (29% of PBS-CL), our algorithms are not global-memory-bound and utilize the compute resources of the GPU efficiently.

VI. Conclusion

In this paper we have analyzed the performance of algorithms that solve the predecessor search problem on modern GPU hardware. We find that performing a naive parallel binary search algorithm (PBS) over search keys in shared memory leads to performance degradation due to bank conflicts. In the worst case, the performance degradation asymptotically leads to $\Omega(w \cdot \frac{Q}{p} \log \frac{K}{w^2})$ shared memory accesses, drastically reducing inter-warp parallelism. As shared memory sizes and, consequently, the number of memory banks grow over time, the impact of bank conflicts on runtimes will become even more pronounced.

To remedy this issue, we introduced two improved search algorithms, the conflict-free PBS-CF and the conflict-limited PBS-CL. The PBS-CF algorithm performs the search without any shared memory bank conflicts, though it is not work

optimal and requires $O(w - \log w)$ *additional* operations per query. PBS-CL performs the optimal $\Theta(\log K)$ operations per query and incurs $w - 1$ total bank conflicts in the worst case. Even on current hardware with relatively small $w = 32$, our algorithms already provide a clear advantage over the PBS algorithm. We have verified this claim via a series of experiments with implementations of the algorithms on GPU hardware. While both of our improved algorithms scale better than the PBS algorithm, we have found that PBS-CL provides the largest performance gain for the majority of problem instances. It achieves its best relative performance when searching large lists stored in shared memory, with a maximum speedup of 2.98 over the PBS algorithm. PBS-CL scales well as K increases, requiring only $O(\frac{Q}{p} \log \frac{K}{w} + \frac{Qw}{p})$ shared memory accesses.

A clear future direction would be to apply our PBS-CL algorithm to even larger datasets, by performing queries on keys loaded in batches from global memory. Another direction would be to enhance existing applications that utilize binary search and quantify the impact on overall performance. Searching is such a fundamental and ubiquitous operation that we would expect our work to be beneficial to many such applications. Finally, a broader avenue for future research would be to apply the insights gained from this work to more complex search techniques and data structures, e.g. B-Trees.

REFERENCES

[1] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. of PPoPP*. ACM, 2008, pp. 73–82.

[2] T. A. T. Han, "hiCUDA: High-level GPGPU programming," in *Proc. of TPDS*, vol. 22, 2010, pp. 78–90.

[3] NVIDIA, "CUDA programming guide 7.0," 2015. [Online]. Available: http://docs.nvidia.com/cuda

[4] D. B. Kirk, *Programming Massively Parallel Processors*. Elsevier Science, 2012.

[5] J. JaJa, *Introduction to Parallel Algorithms*. Reading, MA: Addison-Wesley, 1992.

[6] J. H. Reif, *Synthesis of Parallel Algorithms*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[7] N. Fauzia, L. N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proc. of CGO*, 2015, pp. 12–22.

[8] D. G. Merrill and A. S. Grimshaw, "Revisiting Sorting for GPGPU Stream Architectures," in *Proc. of PACT*. ACM, 2010, pp. 545–546.

[9] D. Merrill and A. Grimshaw, "Parallel Scan for Stream Architectures," Department of Computer Science, University of Virginia, Tech. Rep. CS2009-14, 2009.

[10] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of IPDPSW*, May 2012, pp. 794–803.

[11] ——, "The hierarchical memory machine model for GPUs," in *Proc. of IPDPSW*, May 2013, pp. 591–600.

[12] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for GPUs," NVIDIA Technical Report NVR-2008-003, 12 2008.

[13] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," *Graphics Hardware*, 2007.

[14] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manfedelli, "Fast scan algorithms on graphics processors," in *ICS*, 2008.

[15] N. Leischner, V. Osipov, and P. Sanders, "GPU sample sort," in *Proc. of IPDPS*, April 2010, pp. 1–10.

[16] F. Dehne and H. Zaboli, "Deterministic sample sort for GPUs," vol. abs/1002.4464, 2010. [Online]. Available: http://arxiv.org/abs/1002.4464

[17] Z. Wei and J. JaJa, "Optimization of linked list prefix computations on multithreaded GPUs using CUDA," in *Proc. of IPDPS*, 2010.

[18] A. Davidson, S. Baxter, M. Garland, and J. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *Proc. of IPDPS*, 2010, pp. 78–90.

[19] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," *SIGPLAN Not.*, vol. 47, no. 8, pp. 117–128, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2370036.2145832

[20] K. Kaczmarski, "Experimental B$^+$-tree for GPU," in *Proc. of ADBIS*, vol. 2, Rome, Italy, 2011, pp. 232–241.

[21] ——, "B-tree optimized for GPGPU," in *Proc. of OTM 2012*, Rome, Italy, 2012, pp. 843–854.

[22] A. Shekhar, "Parallel binary search trees for rapid IP lookup using graphic processors," in *Proc. of IMKE*, 2013, pp. 176–179.

[23] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldeway, V. Lee, S. Brandt, and P. Dubey, "FAST: fast architecture sensitive tree search on modern CPUs and GPUs," in *Proc. of SIGMOD*, Indianapolis, Indiana, USA, 2010.

[24] J. Soman, K. Kothapalli, and P. J. Narayanan, "Discrete range searching primitive for the GPU and its applications," *J. Exp. Algorithmics*, vol. 17, pp. 4.5:4.1–4.5:4.17, Oct. 2012.

[25] O. Green, R. McColl, and D. A. Bader, "GPU merge path: a GPU merging algorithm," in *Proc. of ICS*, 2012, pp. 331–340.

[26] B. Catanzaro, A. Keller, and M. Garland, "A decomposition for in-place matrix transposition," in *Proc. of PPoPP*, 2014.

[27] N. Sitchinava and V. Weichert, "Provably efficient GPU algorithms," *CoRR*, vol. abs/1306.5076, 2013. [Online]. Available: http://arxiv.org/abs/1306.5076

[28] S. Baxter, "Modern GPU," 2013. [Online]. Available: http://nvlabs.github.io/moderngpu/

[29] NVIDIA, "GeForce GTX770 specifications," 2014. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-770/specifications

[30] ——, "Nsight," 2015. [Online]. Available: http://www.nvidia.com/object/nsight.html

[31] H. Wong, "Demystifying GPU microarchitecture through microbenchmarking," in *Proc. of ISPASS*, 2010, pp. 235–246.

[32] P. Enfedaque, F. Auli-Llinas, and J. Moure, "Implementation of the DWT in a GPU through a register-based strategy," *IEEE Trans. PDS*, vol. PP, no. 99, 2014.