# Implementation of CAPRI: Prediction of Compaction Adequacy for Handling Control Divergence in GPGPU Architectures

Lakshay Grover, Monam Agarwal, Prashant Solanki, Sabir Ahmed
Electrical and Computer Engineering
North Carolina State University
North Carolina, United States of America
lgrover, magrwa2, psolanki, sahmed12 @ncsu.edu

***Abstract*—Running general purpose workloads on wide SIMD Graphical Processing Units require execution of codes with irregular control flow to be executed on the SIMD pipeline. Typically this is handled by the hardware by serializing the control flow and having a masked execution of the SIMD threads based on the outcome of the branch instruction. However, masked execution leads to wasted execution slots which degrades the performance. Thread Block Compaction (TBC) seeks to alleviate this problem by dynamic compaction of all the warps across a thread block. However, dynamic compaction of warps might not always be beneficial depending on the work load and implementing TBC may just add an overhead without any added advantage. The CAPRI (Compaction Adequacy Predictor) technique presented in this paper proposes a technique that computes the efficiency of dynamic compaction for a particular divergent instruction and thus removes the unnecessary synchronizations required for TBC in conditions where TBC is not effective.**
**In our implementation we statically analyze the active mask of all branch instructions for a given application and compare the SIMD utilization for the original Post-Dominator based execution with SIMD utilization which could have been achieved if a CAPRI based scheduler is used for the divergent instructions.**

***Keywords—SIMD Architecture; control divergence; GPGPU; thread block compaction***

## I. INTRODUCTION

The basis of SIMD architecture is high throughput and performance. The SIMD model exhibits Data level Parallelism and hence shows excellent performance for highly Data parallel applications. However, the occurrence of conditional code introduces serialization that becomes a bottleneck in the performance. Divergence directly impacts parallelism and hence requires careful consideration of techniques that aid in mitigating the performance degradation because of the divergence. GPUs have a hardware stack based structure to keep track of the divergent branches and their execution in sequential order (taken and not-taken). In recent research, several techniques have been introduced to reduce this serialization. Dynamic Warp Formation is one such realistic hardware implementation that dynamically regroups threads on occurrence of divergent branches into new warps aiming to reduce serialization [5]. Another effective technique introduced is the Thread Block Compaction [2] that works at the Thread block granularity and dynamically compacts warps within a thread block into new warps that result in increased SIMD resource utilization in case of detected divergence. Although an efficient technique, TBC has an inherent requirement to put a synchronization barrier as soon as divergence is detected i.e. stalling all the warps of the considered thread block. This becomes an overhead if the compaction of warps doesn't result in performance improvement. The objective of the current project implementation is to have a selective mechanism to stall/bypass the warps based on prediction and evaluation of the compaction effectiveness of the warps in a thread block built on top of TBC. This technique called Compaction Adequacy Predictor (CAPRI) inspired by [1], introduces a prediction hardware and logic to determine the compaction effectiveness. For a particular thread block, warps are selectively stalled and reshuffled to form new warps to maximize the SIMD resource utilization.

In this project, a single level predictor based on the 1-bit latest configuration has been implemented to predict the compaction effectiveness of branches. Logic is implemented to evaluate the compaction adequacy. A trace-based analysis has been done to detect divergence in the application and the data corresponding to the thread block is analyzed and sent for further processing. This has been done to give the effect of the TBC.

## II. BACKGROUND & RELATED WORK

### A. *Divergent Control Flow and PDOM Based Scheduling*

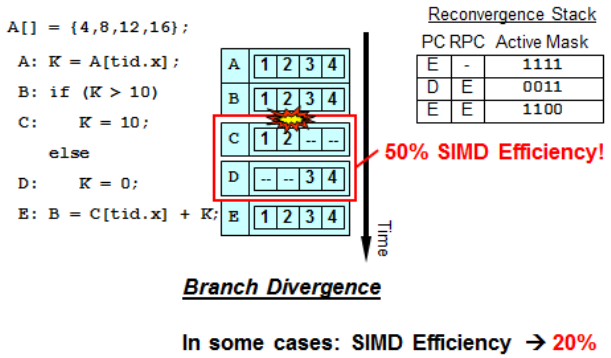


Fig. 1. Branch Divergence control flow using PDOM, courtesy [2]

The underlying SIMT architecture of a GPU allows each SIMD thread to compute its own control flow outcome for a branch instruction. A condition where different SIMD threads within a single scheduling unit (warp as per NVIDIA terminology) follow different execution paths it leads to divergence in the control flow. Typically such a condition is handled in hardware by generating a warp wide active mask based on the outcome of the branch instruction for each thread and then serializing the execution of both the branch targets. Each thread is executed for both the paths but results computed by threads with non-active masks are discarded. A Post-Dominator based warp wide stack is used to keep a track of the active mask and scheduling of divergent instructions.

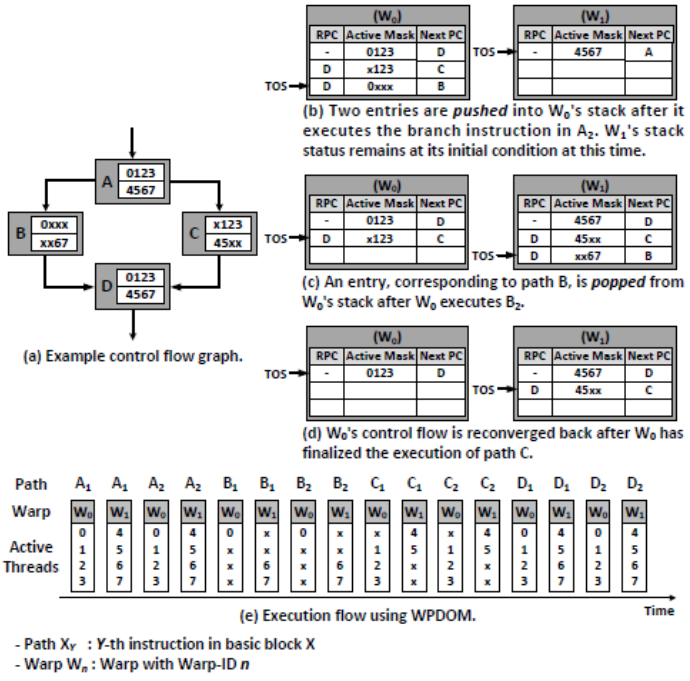


Fig. 2. WPDOM handling branch divergence example, courtesy [1]

For each branch instruction the active mask for the branch instruction and the re-convergent PC (i.e. the PC where the active mask is the same as that at the branch instruction) are pushed into the stack. Now if divergence is detected two more entries are pushed into the stack one each for the two branch paths. Now instructions are fetched from branch path whose first path is given by the next PC entry given by the TOS entry. The entry is popped from the stack when the PC of the fetched instruction is equal to the re-convergent PC. The process is continued till TOS points to the branch instruction after which the threads re-converge and execution is continued with the original active mask.

This is an effective method to handle instruction scheduling for divergent instructions, however with each nested branch instruction the number of active threads for each path of the branch instruction reduces, reducing the performance.

### *B. Thread Block Compaction*

[2] Thread block compaction is one effective method to improve the performance (SIMD utilization) in case of divergent instructions, by dynamically compacting all the active threads for a particular branch instruction into minimum possible number of warps.

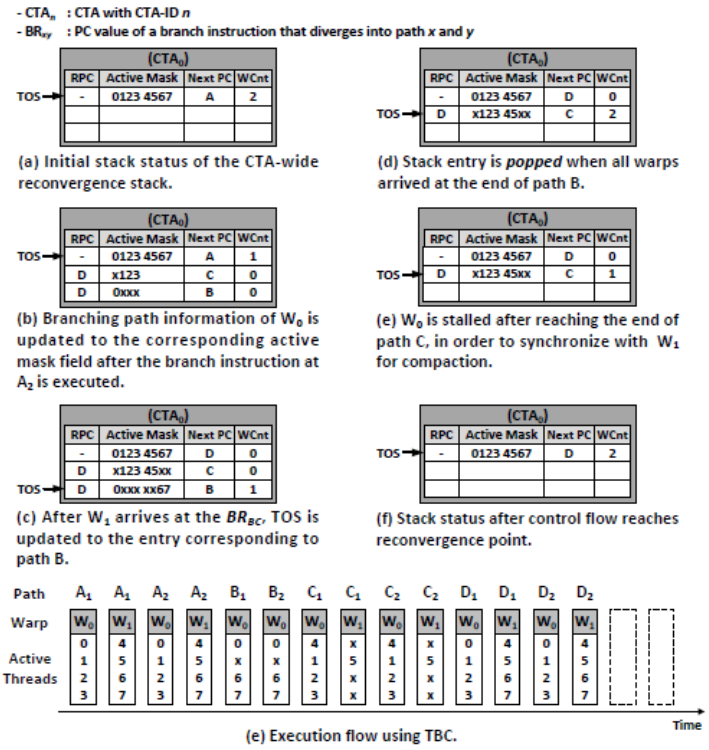


Fig. 3. TBC handling branch divergence example, courtesy [1]

Instead of having a single re-convergence stack for each warp a new thread block wide re-convergence stack is introduced. Whenever a branch instruction is encountered for one of the warps a new entry is pushed into the stack which consists of the active mask of the branch instruction re-convergent PC and number of warps yet to arrive at the branch instruction (WCnt). Two more entries are pushed into the table for each of the branch paths. The warp is now stalled to wait for all the pending warps to arrive at the branch instruction. As each new warp arrives its active mask is pushed into the table along with the previous entries of the previous warps and the value of WCnt is decremented by one. When WCnt becomes zero, active mask for all the threads in the thread block are available, the entry at TOS is sent to the WCU, which returns the new compacted warps, which can now be scheduled. The rest of the functioning of the thread block stack is similar to the PDOM stack.

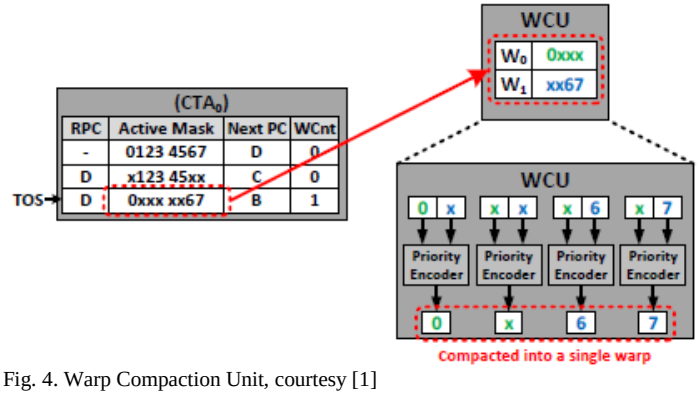


Fig. 4. Warp Compaction Unit, courtesy [1]

Compaction is only effective when number of warps formed after compaction is less than the number of warps before compaction. This is possible only when the active threads for different instructions do not lie in the same SIMD lane. In cases where they lie in the same lane, the number of warps after compaction remains the same. Hence unnecessary stall is introduced. Another drawback is that this unnecessary shuffling of warps may lead to an un-coalesced memory access thereby further reducing performance.

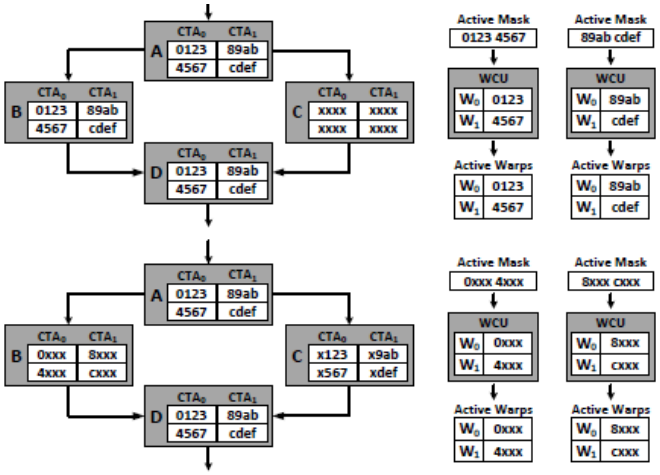


Fig. 5. Examples showing flow graphs of compaction-ineffective branches, courtesy [1]

### C. *Compaction Adequacy Predictor*

As described above dynamic thread block compaction is not always effective. CAPRI is basically a single-bit predictor based on whose output the scheduler decides when it would be suitable to stall a warp to be compacted with the rest of the warps in the thread block. It consists of three major components, the **CAPT i.e. the Compaction Adequacy Prediction Table** which keeps a track of the Compaction Adequacy for each branch instruction, the **Decision Logic** which decides whether to stall a warp for compaction and the **WCU i.e. the Warp Compaction Unit** which computes compaction adequacy for a particular branch instruction.

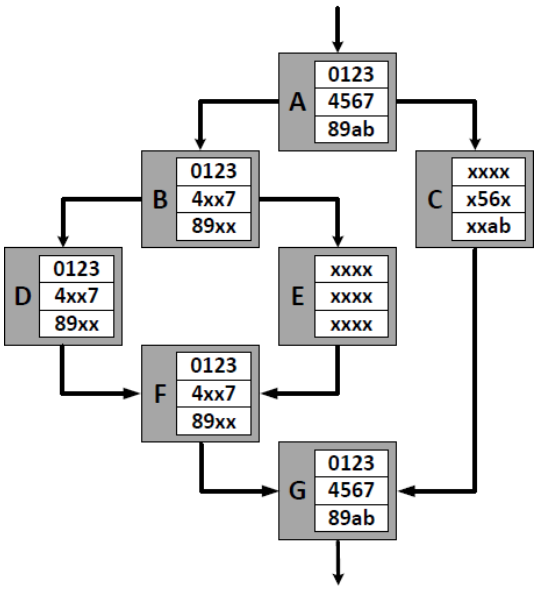


Fig. 6. Examples showing a control divergence flow for CAPRI, courtesy [1]

For the above CFG when Warp 0 reaches the branch $BR_{BC}$ in the basic block A, a new entry is pushed into the active mask stack which consists of the next PC and the number of warps which still need to arrive at the branch instruction within the thread block. Entries corresponding to the not taken and the taken paths are pushed into the stack. Since no divergence occurs in warp 0 its UMask (mask value referred to when generating compacted warps) is set to zero and the warp is marked as ready for scheduling. When warp 3 arrives at the branch instruction, it's taken and not taken masks are pushed into the stack and since divergence takes place the branch instruction is looked up in the CAPT. Since no entry exists in the table corresponding to the branch instruction, a new entry is made in the CAPT and the instruction is marked as adequate for compaction by using a conservative policy.

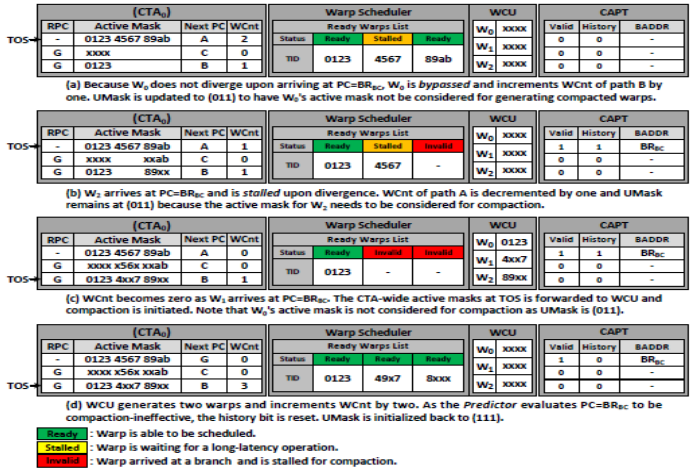


Fig. 6. Different components of CAPRI, courtesy [1]

The UMask of warp 3 is set as 1 and since WCnt is still not zero the warp is stalled to wait for the pending warps. Now warp 2 arrives at the branch instruction and diverges. Its active mask entries are updated into the stack and since the history bit in the CAPT corresponding to the branch instruction is 1, the corresponding UMask is set to 1and the warp is stalled for compaction. Since the WCnt now become zero, the active mask at TOS is sent to the WCU, which generates new compacted warps using threads from the warps for which UMask is set as 1. The WCU returns the new compacted warps which can now be scheduled. Since the warps after compaction remains the same as that before compaction the branch instruction in not compaction adequate and the history bit is set to 0. So control flow if the same branch instruction is encountered again now since the history bit is set a 0, none of the warps will be stalled for compaction.

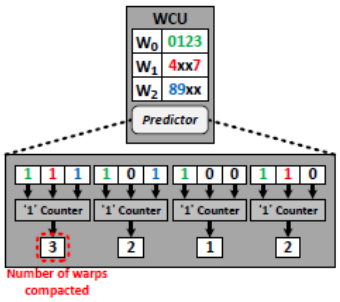


Fig. 8. Finding the number of warps formed, courtesy [1]

## III. METHODOLOGY & IMPLEMENTATION

The CAPRI is modeled in the GPGPU-Sim (version 3.2.2), which is a publically available, detailed cycle-based simulator of a general purpose GPU. The GPGPU simulator has been modified to implement the functionality of the CAPRI i.e. a single level Compaction Adequacy Predictor is implemented based on a single level branch predictor (1-bit latest configuration). The CAPT is indexed with the help of the Branch PC of the branch operation and the adequacy bit entry in the CAPT is updated based on the results obtained by a trace-based static analysis, generated by studying the data corresponding to various thread blocks, warps and their active masks. The prediction logic inside the WCU mechanism has been used at every branch instruction to update the compaction adequacy bit in the CAPT as suggested in [1]. Initially the kernel is run on the simulator and data corresponding to all the branch instructions i.e. the operation code along with the Thread Block IDs of the thread blocks being executed, the Warp IDs, the PC value and the Active mask of the threads in the current warp being executed is obtained. This data is then searched for the occurrence of the Branch operation in the sequence of instructions being executed in all the warps. The minimum PC instruction in any warp in a particular thread block is searched. The active masks of the threads belonging to a single thread block are then analyzed for compaction adequacy through a simple implementation of compaction adequacy logic inside the Warp Compaction Unit. For a particular diverged path, the number of active threads in one lane in a particular thread block is counted. This gives the number of warps that can be obtained after compaction. It is compared with the number of warps active in that particular path without compaction. If this number is less, the CAPT adequacy bit entry is updated to indicate compaction effectiveness at that particular instance of the branch instruction in the code. The nested branch divergence has been handled by maintaining a Thread Block wide stack that gets a new entry whenever a new branch is encountered. It has been implemented to count the number of instructions to be executed in one particular divergent path. It is used for the calculation of SIMD Utilization and other parameters measuring performance of the simulator.

## IV. EXPERIMENTAL RESULTS

Experiments have been carried out by running the modified GPGPU simulator for the BFS (Breadth First Search), TAPFC (Two Pt. Ang. Cor. Fun.) CUDA codes which have been taken from parboil benchmark suite [8]. To study the effect of the modifications in the GPGPU Sim, a simple test-case was created as shown in the snippet [Fig. 9] below. The code runs with 64 threads and a single thread block. The first *for* loop gives rise to compactable warps, because first half threads in the warp take the not taken path and a totally opposite scenario applies to the next warp. The second *for* is a non-compactable one, where threads with odd and even *tid*s take follow a different path.

Two major parameters have been compared i.e. the Average SIMD utilization factor and the number of mispredictions. The SIMD Utilization factor is defined as the ratio of the number of active threads in a warp to the total number of threads in that warp taken for all the instructions being executed in that particular divergent path. The number of mispredictions is calculated as the number of times the last entry of the adequacy bit in the CAPT for a particular branch instruction does not match with the latest prediction made with context to the adequacy bit.

```
for(i=0; i<DATA_LENGTH; i++){
    //first half of first warp
       //and second half of 2nd warp
    if(tid < 16 || tid > 47)
      out[i] = in1[i]+in2[i];
    else
      out[i] = in1[i]+in2[i];
                  }

  for(i=0; i<DATA_LENGTH; i++){
    //even threads not compactable
    if(threadIdx.x%2 == 0 )
      out[i] = in1[i]+in2[i];
    else
      out[i] = in1[i]+in2[i];
    }
```

Fig. 9. Sample test-case to test modified GPGPU-Sim

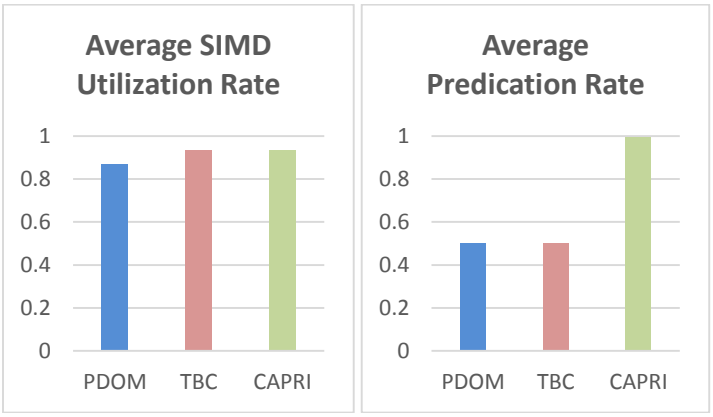


Fig. 10. Comparison of PDOM, TBC, CAPRI over Average SIMD Utilization Rate and Average Prediction Rate for the sample test-case

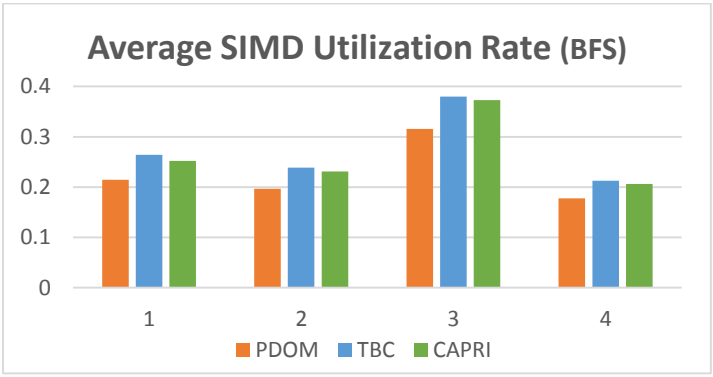


Fig. 11. Comparison of PDOM, TBC, CAPRI over Average SIMD Utilization Rate

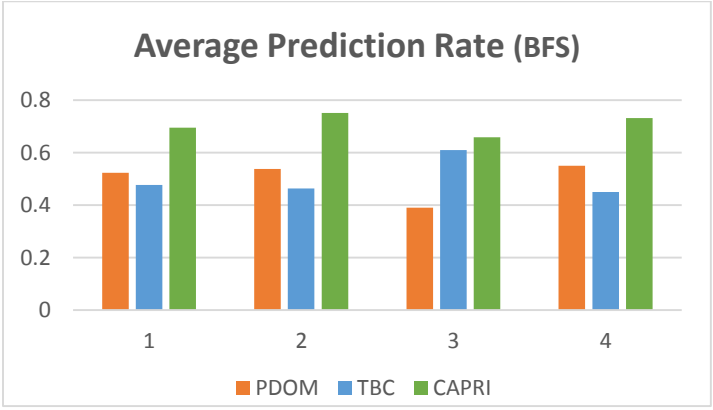


Fig. 12. Comparison of PDOM, TBC, CAPRI over Average Prediction Rate

## V. CONCLUSION

This report presents implementation of a technique introduced in [1]. The idea of compaction adequacy prediction implemented is a purely hardware based mechanism that tries to reduce the unnecessary synchronization overheads accompanied by Thread Block Compaction that is otherwise meant to improve performance in divergent applications. The implementation of the CAPRI gives a boost in performance by adding a decision making capability based on the predictor. Compaction is only performed when the CAPRI's prediction is true, else there is no need to stall the warps. Also, CAPRI is self-learning as it keeps updating the Adequacy bit after execution of each branch. For the evaluation of CAPRI, the CAPT, the decision logic and the predictor logic inside the Warp Compaction Unit have been implemented. In conclusion, CAPRI bring about a performance improvement of approximately 17.79% over baseline PDOM approach for BFS benchmark [Fig. 11]. CAPRI avoids 41.78% of the unnecessary that would have been incurred by TBC, as shown by the Average Prediction Rate [Fig. 12].

## VI. REFERENCES


[1] Minsoo Rhu, and Mattan Erez. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. In Proceedings of the International Symposium on Computer Architecture (ISCA'12). June, 2012

[2] W. W. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In 17th International Symposium on High Performance Computer Architecture (HPCA-17), February 2011

[3] J. Meng, D. Tarjan, and K. Skadron. "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence." In Proceedings of the 37th ACM/IEEE International Symposium on Computer Architecture, June 2010

[4] Minsoo Rhu and Mattan Erez, Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation, ISCA 2013

[5] Wilson W. L. Fung Ivan Sham George Yuan Tor M. Aamodt, Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow In Proc. MICRO, 2007, pp. 407–420.

[6] GPGPU-Sim- http://www.gpgpu-sim.org.

[7] GPGPU-Sim Manual- http://www.gpgpu-sim.org/manual

[8] Parboil Benchmark Suit- http://impact.crhc.illinois.edu/Parboil