# Branch Prediction Sharing in SIMT Architectures: Exploiting Peer Information on Future Events

Steven Braeger
Department of Electrical Engineering and
Computer Science
University of Central Florida
Orlando, Florida 32826
Email: steve@soapforge.com

Nicholas Arnold
Department of Electrical Engineering and
Computer Science
University of Central Florida
Orlando, Florida 32826
Email: narnold@knights.ucf.edu

*Abstract*—**Branch Prediction has long been used as a method of reducing the latency incurred by a control hazard. However, traditionally, SIMT architectures do not implement branch prediction hardware in order to reduce the complexity of each of the GPU cores. Although the SIMT model provides some opportunities for control hazard penalty reduction in the form of context switching, there are several cases where the penalty is incurred regardless. We experimented with the application of a branch predictor to a SIMT architecture in order to reduce those hazards. Furthermore, our model is capable of sharing information about symmetric runs from multiple concurrent contexts. By using a shared Smith_2 branch predictor [1] with a large branch history table, we can demonstrate significantly increased prediction accuracy over a traditional single-context scheme.**

## I. Introduction

There is a great deal of interest in the evolution of Single-Instruction Multiple-Thread (SIMT) architectures in recent years. Due to their tremendous performance and parallelism, as well as an increased generality of these architectures for scientific applications, these types of architectures have moved beyond their traditional application in graphics programming to many other application domains, and are becoming more and more useful. In addition, another factor driving their development is their rapid architectural evolution, which is facilitated by the fact that they do not have a standard instruction set to implement.

The SIMT model, which we describe in more detail in III, is not without its flaws. In order to achieve greater parallelism and FLOPS, SIMT processors often remove a lot of the superscalar architectural features such as caching, predication, branch prediction,out-of-order execution, and sometimes even pipelining. These features are replaced with multiple concurrent contexts of execution, known as 'threads'.

However, we believe that benefit can be gained from re-integrating one of those superscalar features: specifically, branch prediction, to gain performance by reducing the latency from control hazards that occur when the incorrect instruction is fetched during a jump.

In addition, we make the observation that branch prediction on a SIMT architecture can actually perform better then branch prediction on a MIMD architecure like a modern multicore CPU. The intuition for this observation, described in IV-B, derives from the fact that the SIMT model guarantees that each context of execution is not working alone, but in teams, called context groups or 'warps', and each of these context groups is also working on a single core. Whereas traditional branch prediction is limited to predicting based on knowledge gained from the past, SIMT branch prediction schemes are able to gain information about the present and near-future, by sharing information about execution with its peer contexts and context-groups.

We demonstrate an implementation of this scheme using a Smith_2 predictor [1] described in **??**. Our implementation runs in simulation [2], and demonstrates the impact of sharing no information (traditional MIMD model), sharing information per context group, and sharing information per core.

Our experiment demonstrates that utilizing peer information in the SIMT model provides a significantly improved prediction accuracy rate over the MIMD model for a subset of the ISPASS09 [2] benchmarking set.

## II. Related Work

### *A. Branch Prediction*

Yeh and Patt [3] discussed implementing an adaptive branch predictor that would use information being collected at run-time, as opposed to other schemes which would require a pre-run of the program for training. Similar to our approach, their setup requires a Branch History Table and a Branch History Pattern Table, which is updated based on the outcomes of the branches during run-time.

Eden and Mudge [4] developed YAGS, or Yet Another Global Scheme, which is a combination of several strong points of previously implemented schemes. Using information from implementations such as GShare, the Agree Predictor, the Bi-Mode Predictor, the Skewed Branch Predictor, and the Filter Mechanism, the authors use some schemes to implement a predictor to store the biases of branches and other schemes to implement a predictor to store the direction of those branches. The two varieties of predictor catch not only the regular behavior of branches, but also the special cases instead of just allowing them to be mispredictions.

Pan, So, and Rahmeh [5], also attempted improved branch prediction, this time by referencing a subhistory of branch in addition to the branch history table itself. They claim that by adding only a shift register to the architecture, they were able to achieve an additional 11% prediction accuracy.

### B. Branch Prediction on GPU

He and Zhang [6] also attempt branch prediction on the GPU. Their focus, however, was to to test out parallel branch-prediction on GPUs for future use in general purpose many-core CPUs. Their goals were to get independent branch predictors operating on different cores to cooperate with each other in an effort to:

1) increase prediction accuracy,
2) dynamically combine predictors to make them more powerful, or
3) switch them off if they are behaving the same in order to save power.

However, the authors use the GPU to test CPU-style computations, not exploiting the parallel neighbor efforts of the SIMT model. Their efforts apply to CPU-style runtime paradigms but tested on a GPU, compared to our approach that truely exploits the SIMT nature of GPU-style computations.

## III. SIMT vs MIMD

In general, CPU architecture is different than GPU architecture in a few ways. CPU architecture includes logic for the fetch and decode phases, an ALU for execution, the execution context (registers, etc.), control logic, branch predictor, memory pre-fetcher, and the data cache. With this setup, a general CPU is designed to perform Single Instruction, Single Data (SISD) operations well, but performs poorly on Single Instruction, Multiple Data (SIMD), Multiple Instruction, Single Data (MISD), and Multiple Instruction, Multiple Data (MIMD). GPU architecture removes all of the logic designed to make single threaded processes run more efficiently. There is no control logic, no branch predictor, no memory pre-fetcher, and no data cache [7]. With this architecture, a general GPU is designed to perform SIMD operations well, but performs poorly on SISD, MISD, and MIMD.

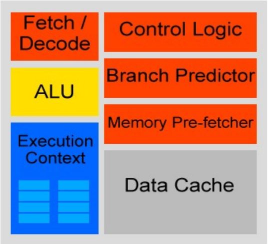

Fig. 1: A general-purpose CPU core

In contrast, a SIMT architecture often has none of these featuresets, but has many more concurrent ALU threads or 'contexts' on the die.

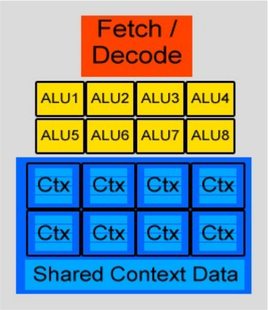

Fig. 2: a single simple GPU core

Recently, CPU designers have begun building chips with multiple cores. This gives the CPU the ability to run SISD and MISD operations well, however there are still performance issues with SIMD and MIMD. GPU design has been taking advantage of multiple cores per chip for years, and since the archtecture takes up less room than a traditional CPU core, a GPU is capable of fitting many cores on the chip [8]. This gives the GPU the ability to run SIMD and MIMD operations well. However, due to the multiple contexts per core, and how the data elements are assigned to these cores, the GPU is not well designed to handle SISD or MISD operations well.

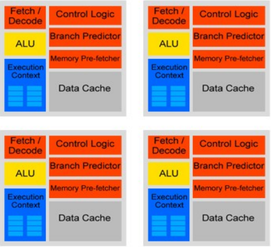

Fig. 3: A multicore CPU design

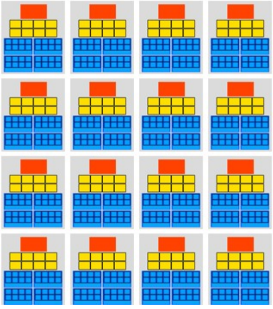

Fig. 4: A multicore GPU design

GPU cores have the ability to modify their context storage areas to vary the number of context elements it holds by modifying the size of the context elements [7], which affects the number of context groups for the core to alternate between during computation. These alternating context groups is how the GPU tries to hide data stalls. When a group encounters a branch, since there is no forwarding logic, the group must wait for the data. While this is happening, the core switches to the next group and performs the same instructions that it just executed on the previous group. By alternating these groups in this manner, called interleaving, the GPU can hide the vast majority of data stalls or branch calculation waits.

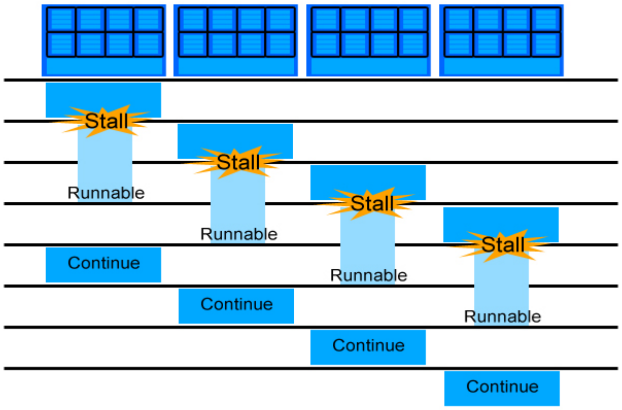

Fig. 5: GPU Context switching

However, what if the stall time exceeds the amount of time it would take this particular GPU core to alternate amongst all if its context groups? In this case, there is nothing that can be done to avoid the stall, and the core must wait. But what if this could be avoided? Our proposal will endeavor to solve this problem.

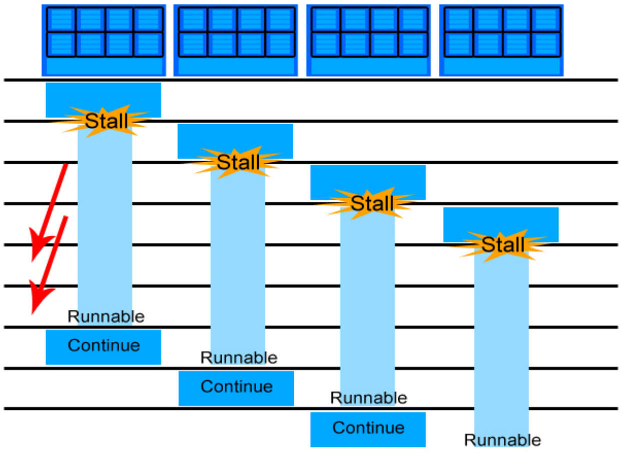

Fig. 6: GPU context switch failure.

The collection of all context belonging to one GPU core is called a ’warp’, and the number of contexts in a warp is its ’warp factor’ [8]. All contexts within a warp will be subjected to the same instruction stream. As mentioned before, these contexts will be processed context group by context group. Due to the nature of how data is assigned to these context elements, there is a great deal of spatial locality amongst the contexts in a warp. Our proposal believes that there is a way to use the information discovered by the first context group being processed by an instruction stream to assist later context groups in their knowledge of stalls.

On a CPU, each core only has one execution context and one branch prediction table. This shows that each context handles its own branches and receives no external assistance from other threads. No external assistance can be offered becuase there is no way to guarantee what the other threads are working on and if it could be useful to the current thread. In contrast, a GPU core has multiple contexts per core that are all performing the same operations. Is it possible to use this information to assist other contexts in their branch stalls?

## IV. Contribution

### A. SIMT Branch Predictor Functional Unit

Our branch predictor is based on the 2-bit Smith predictor [1]. This predictor works with a branch history table, indexed by the PC, where each entry in the table stores a 2-bit branch history register. As seen in the figure below, the branch history register is a saturating counter that represents a state machine. The highest order bit of the register determines the prediction to be made, and after a branch is taken or not taken, the result of the branch changes the state according to the state diagram shown.

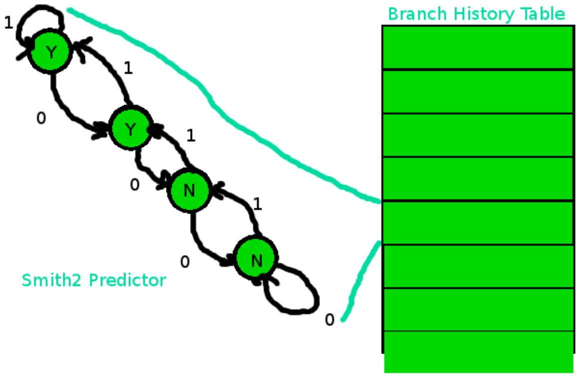

Fig. 7: Smith 2-bit predictor.

Typically, the branch history table is indexed by the low-order bits of the PC. This is done so that each branch in the program can have a unique register associated with it. However, on MIMD architectures the length of the program is not limited in any particular way, so the size of the branch history table is necessarily less then the total number of instructions in the program. This implies that there may be collisions in the program, if the table is too small to include the whole program, where more than one branch maps to the same spot the branch history table. In contrast, on SIMT architectures there is a hard limit on the maximum instruction length that may be executed by a core, and so we may guarantee no collisions in our branch history table by taking advantage of that limit. In our implementation, the size of the branch history table will be exactly the size of the maximum instruction stream length, which is 16,384 instructions. We chose this size so that there would be absolutely no predictor collisions, and each instruction location will have its own dedicated predictor. The total memory used up by this design for our predictor design is 2 bits per entry at 16k entries, for a total size of 4k.

### B. SIMT Functional Unit Sharing

We propose to implement 3 GPU Branch Prediction schemes with the following features. In order to communicate information from neighboring execution contexts, we share a single branch predictor between the contexts in their context group, and also between all the context groups. Because all contexts concurrently modify the predictor, information from one context about the branches that occured may propogate sideways in time to peer threads.

Our simulated hardware designs are as follows.

- One predictor is shared amongst the contexts within a GPU core
- One predictor is shared only amongst its context group (1 per group)
- One predictor is shared only amongst its context (1 per context), as in a multicore CPU, for experimental control

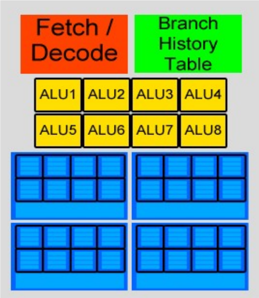

Fig. 8: One shared BHT for the entire core.

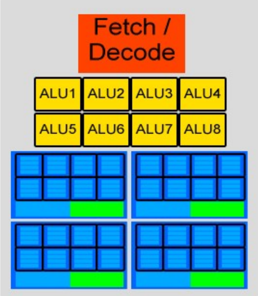

Fig. 9: One shared BHT for each context group.

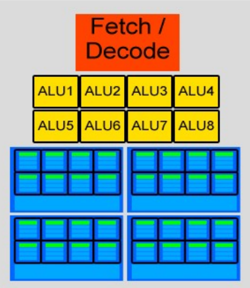

Fig. 10: One BHT for each individual context. This implementation most closely resembles how a general CPU would handle branch prediction.

The intuition for our approach is as follows. The SIMT programming paradigm involves the preparation of a single "kernel" which is a short snippet of executable code to be compiled and run in parallel on each of the threads,warps, and possibly cores. Each instance of this kernel running on a thread of execution is distinguished from the others by a unique identifying thread id. Below is an example matrix-vector multiply kernel with a single branch point.

GemV.cl

```
__kernel gemv(double * y, double * A, double * x)
{
    int rowout = cl_thread_id(0);  // index
    y[rowout] = 0.0;
    for(int i = 0; i < A.cols; i++)
        y[rowout] += A[rowout,i] * x[i];
}
```

If this kernel was run with a traditional MIMD approach (e.g., one independent branch predictor FU per context of execution), then it is almost impossible for that branch predictor to correctly predict branch during the final iteration of the inner for loop, as all previous executions of the loop have been taken, it is rational for the branch predictor to predict not taken.

However, if this kernel was run with one of our SIMT branch predictors, then it IS possible for many of the threads of execution to correctly predict the final iteration of the for loop as "not-taken", because when only a few of its peers mispredict, they update the shared predictor with what happens. Due to the instruction parallel nature of the SIMT paradigm, if a peer mispredicts then it is likely that you will as well, allowing you to pre-emptively fix your prediction.

We will compare prediction accuracies across these three implementations. Our goal is that after the initial training of the predictors, the GPU cores will process their contexts groups as shown in the figure below:

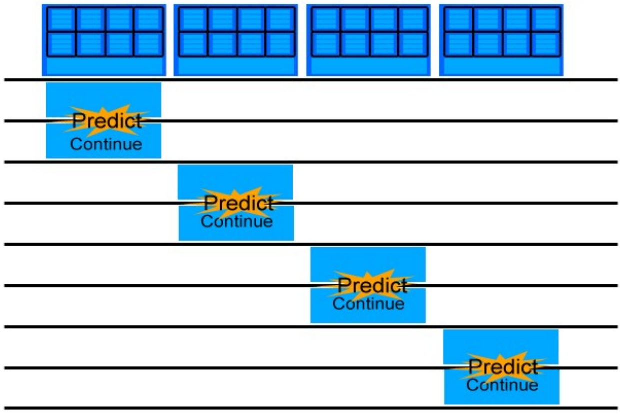

Fig. 11: Context interleaving is not necessary, as the groups are properly predicting where to branch and continue operation.

## V. Experimental Methodology

In order to properly model our proposal, we must have an appropriate simulator modeling SIMD/SIMT architecture. The SimpleScalar simulator was not sufficient, as it simulates CPU behaviors (including all of the behaviors that are removed from GPUs). SimpleScalar is not sufficient, so we needed to find a GPU-specific simulator. We found GPGPU-Sim, which is a rewrite based on the SimpleScalar simulator. This simulator

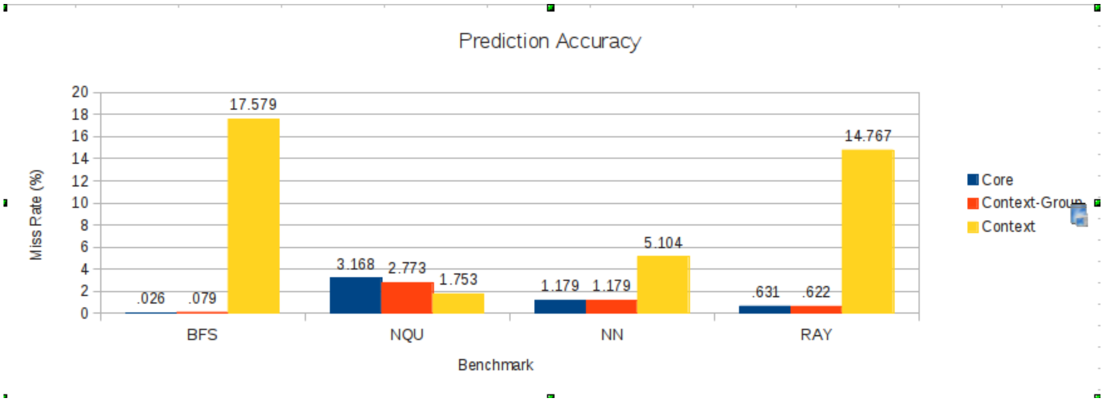

Fig. 12: Simulation Results

simulates an NVIDIA Quadro 5800 graphics card, with the following statistics:

- 16k register file
- 4 GB RAM
- 32 GPU cores
- 1024 contexts

This simulator creates libcuda.so and libOpenCL.so files on the fly to intercept GPU commands from CPU program during execution. This method of execution requires NVIDIA hardware to build benchmarks because the driver does the compilations.

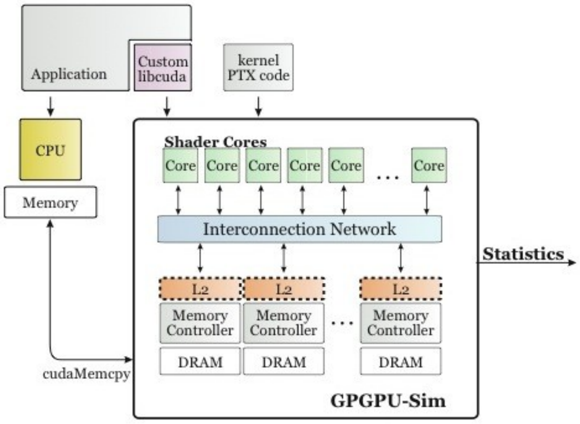

Fig. 13: GPGPU-Sim architecture.

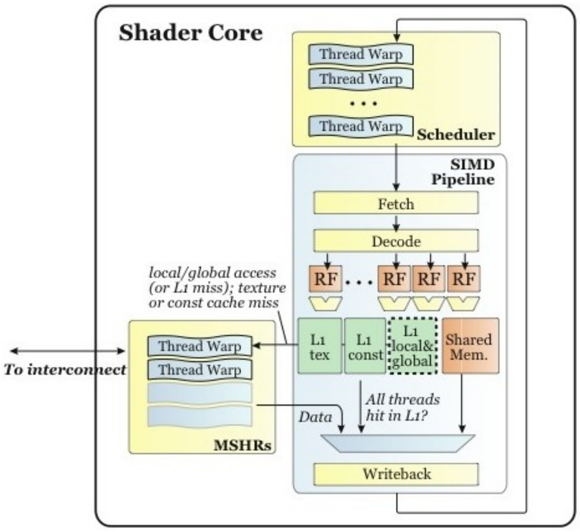

Fig. 14: GPGPU-Sim Shader core.

Since we are simulating GPU hardware, it is important to run our simulator on GPU-specific benchmarks as gcc, gzip, and other common benchmarks are designed to benchmark CPU hardware changes. We settled on using a subset of ISPASS09 CUDA benchmarks [2]. We used a subset of these benchmarks, due to problems with the CPU component or problems with the revision of the CUDA architecture supported by our simulator. (double precision vs. single precision computations). The AES benchmark was unsuitable because none of the instructions were branches.

We settled on a suite of four benchmarks:

- *NQU* - Solving the NQUEENS problem
- *NN* - Classification of images with a neural net
- *RAY* - Raytracing a scene
- *BFS* - Breadth-First Search of a graph

## VI. Simulation Results

We ran our simulations on all three branch prediction architectures: Shared Branch Predictor per Core, Shared BP per Context Group, Shared BP per Context (MIMD), and got the results shown in Figure 12.

There were significant improvements in prediction accuracy in the per group and per core predictors when compared to the per context predictor. The per context predictors on *BFS* scored nearly an 18% miss rate. However, both the per group and per core predictors scored amazing 0.9% and 0.8% miss rates, respectively. While not as huge of an improvement as with the *BFS*, the per group and per core predictors posted very respectable improvements over the per context predictor while operating on *NN* and *RAY*.

Second, we noticed that the per group and per core predictors performed worse than the per context predictor while operating on the *NQU* benchmark. It is possible that this occurred due to the nature of the *NQU* problem, which does not exhibit the same kind of spatio-temporal locality as would be required to take advantage of either the per group or per core predictor information sharing.

## VII. Conclusion

We have demonstrated that the data-parallel nature of the SIMT model can be successfully used to dramatically increase prediction accuracy by sharing prediction structures accross multiple gpus. We have shown, through experiment, that even a simple branch predictor such as a Smith bimodal predictor benefits greatly from information given to it by peer contexts sharing dynamic branch information. We have shown several possible configurations of this branch predictor and demonstrated the performance gains.

### *A. Future Work*

There are several important areas of future research.

1) Multiple predictors can be studied that may perform better than a simple bimodal predictor for this circumstance of peer sharing.
2) The exact impact of stall improvement latency was not studied by our approach, as our simulator did not have the appropriate timing model to model it. Determining the benefit of adding predictors in terms of real CPI performance is crucial to the importance of this class of techniques.
3) Techniques were not studied to possibly share a predictor over multiple concurrent cores. However, although this is possible, we believe this would not be a fruitful area of research due to the fact that multiple cores do not have locality in an program,space,or time sense.
4) The details of physical implementation deserve further study. A large BHT per core or per context would be a significant amount of die space, and each element in the table would need to be able to handle multiple concurrent accesses from multiple threads in a single clock, which may prove to be difficult practically.

## References

[1] J. E. Smith, “A study of branch prediction strategies,” in *Proceedings of the 8th annual symposium on Computer Architecture*, ser. ISCA ’81. Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, pp. 135–148. [Online]. Available: http://dl.acm.org/citation.cfm?id=800052.801871

[2] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” pp. 163–174, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2009.4919648

[3] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive training branch prediction,” in *Proceedings of the 24th annual international symposium on Microarchitecture*, ser. MICRO 24. New York, NY, USA: ACM, 1991, pp. 51–61. [Online]. Available: http://doi.acm.org/10.1145/123465.123475

[4] A. N. Eden and T. M. ane, “The yags branch prediction scheme,” in *In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998, pp. 69–77. [Online]. Available: http://www.eecs.umich.edu/~tnm/papers/yags.pdf

[5] S.-T. Pan, K. So, and J. T. Rahmeh, “Improving the accuracy of dynamic branch prediction using branch correlation,” *SIGPLAN Not.*, vol. 27, pp. 76–84, September 1992. [Online]. Available: http://doi.acm.org/10.1145/143371.143490

[6] L. He and G. Zhang, “Parallel branch prediction on gpu platform.” in *HPCA (China)*, ser. Lecture Notes in Computer Science, W. Zhang, Z. Chen, C. C. Douglas, and W. Tong, Eds., vol. 5938. Springer, 2009, pp. 153–160. [Online]. Available: http://dblp.uni-trier.de/db/conf/cnhpca/cnhpca2009.html#HeZ09

[7] K. Fatahalian, “Beyond Programmable Shading: How Shader Cores Work,” ser. SIGGRAPH 2009, 2009. [Online]. Available: http://s09.idav.ucdavis.edu/

[8] D. Luebke, “GPU Architecture: Implications Trends,” ser. SIGGRAPH 2008, 2008. [Online]. Available: http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf