# *Revisiting ILP Designs for Throughput-Oriented GPGPU Architecture*


| *Ping Xiang* | *Yi Yang* | *Mike Mantor* | *Norm Rubin* | *Huiyang Zhou* |
|---|---|---|---|---|
| *Dept. of ECE* | *Dept. of Integrated Systems* | *Graphics Group* | *Nvidia Research* | *Dept. of ECE* |
| NCSU | NEC Labs | AMD | Nvidia | NCSU |
| Raleigh, NC | Princeton, NJ | Orlando, FL | Santa Clara, CA | Raleigh, NC |
| pxiang@ncsu.edu | yyang@nec-labs.com | Michael.Mantor@amd.com | nrubin@nvidia.com | hzhou@ncsu.edu |



***Abstract*—Many-core architectures such as graphics processing units (GPUs) rely on thread-level parallelism (TLP) to overcome pipeline hazards. Consequently, each core in a many-core processor employs a relatively simple in-order pipeline with limited capability to exploit instruction-level parallelism (ILP). In this paper, we study the ILP impact on the throughput-oriented many-core architecture, including data bypassing, scoreboarding and branch prediction. We show that these ILP techniques significantly reduce the performance dependency on TLP. This is especially useful for applications, whose resource usage limits the hardware to run a high number of threads concurrently. Furthermore, ILP techniques reduce the demand on on-chip resource to support high TLP. Given the workload-dependent impact from ILP, we propose heterogeneous GPGPU architecture, consisting of both the cores designed for high TLP and those customized with ILP techniques. Our results show that our heterogeneous GPU architecture achieves high throughput as well as high energy- and area-efficiency compared to homogenous designs.**



***Keywords-GPGPU; Heterogeneous; ILP; Energy***


## I.INTRODUCTION

The design philosophy of many-core architectures such as graphics processing units (GPUs) is to exploit thread-level parallelism (TLP) to achieve high throughput. Compared to central processing unit (CPU) designs, GPU-like many-core architectures spend the on-die area mainly for computational/instruction execution logic rather than caches or complex instruction processing, such as register renaming and out-of-order execution, to extract instruction-level parallelism (ILP). Each core in a GPU, referred to as a shader core (SC), is a relatively simple in-order multi-threaded processor, which primarily relies on TLP to overcome pipeline hazards. In this paper, we propose to architect SCs for GPU-like many-core processors to achieve both high performance and high energy- and area-efficiency.

We first revisit the ILP techniques [5][9] for in-order processors, including data bypassing/forwarding, scoreboarding and branch prediction, to gain insight on how they interact with throughput-oriented many-core architectures. Similar to CPUs, data bypass in a GPU accelerates execution of producer and consumer instruction pairs from the same threads; a scoreboard checks data independency to support the 'stall-on-use' policy, i.e., the pipeline is not stalled by a long latency instruction such as a cache miss, instead it is stalled by the consumer instruction of the loaded value; and branch prediction aims to reduce the impact of control hazards. Since GPU-like many-core architectures execute instructions in the single-instruction multiple-thread (SIMT) mode, there are new challenges and opportunities to implement these ILP techniques. In this paper, we present the many-thread-aware ILP designs and analyze their impacts on performance, area, and power/energy consumption.

Modern GPUs have implemented some ILP techniques like bypass or scoreboard. But the implementation details have not been disclosed and our study provides insight into such designs. More importantly, our proposed scoreboard design supports *precise interrupts*. A key motivation for precise interrupts is that with GPUs being widely used for general purpose computation, they need to support virtual memory. In addition, in the server environment, GPUs will need to provide the context switch capability in order to service multiple tasks, which also requires precise interrupts. In a recent work [16], support for precise interrupts in GPUs is also proposed using idempotent code regions generated by the compiler.

Our experiments show that the ILP techniques in GPGPU are effective for two types of applications: (1) applications with high resource requirement in registers or shared memory, which limits the number of threads that can run concurrently, thereby limiting the capability for TLP to hide pipeline hazards, especially those due to long-latency cache misses; and (2) applications with uneven workloads, in which few threads (or thread blocks) with the largest workloads will dominate the overall performance. It is worth pointing out that these applications benefit significantly from the many-core architecture, achieving tens or even hundreds instructions per cycle. Therefore, these applications are more suitable to GPUs than CPUs. The problem, however, is that they cannot fully utilize the massive computational power available in GPU-like many-core architectures. ILP techniques, in this case, effectively utilize the otherwise idle hardware resources. On the other hand, for applications that TLP alone can achieve high hardware utilization, the ILP techniques are not effective. To efficiently handle such workload-dependent behavior, we propose heterogeneous GPU-like many-core architectures, which contain two types of cores, one customized for ILP friendly applications and the other for TLP friendly ones. This heterogeneous architecture is particularly useful for concurrent kernel execution, which is supported by current GPUs such as NVIDIA Fermi/Kepler architecture for general purpose computation [19]. When concurrent kernels have different characteristics, heterogeneity improves both throughput and energy efficiency. For single kernels or homogeneous concurrent kernels, we show that our heterogeneous architecture also

achieves similar or higher performance compared to homogeneous designs.

In summary, this paper makes the following key contributions: (1) we present a detailed study to reveal the effectiveness of ILP on throughput-oriented many-core GPU-like architecture; (2) we show that ILP techniques present an interesting alternative to high degrees of TLP to achieve high performance. Considering the significant resource requirements to support high TLP, certain ILP techniques can be area and energy efficient even in throughput-oriented many-core architectures; (3) we propose heterogeneous GPU-like many-core architectures and a policy to steer applications to the appropriate type of SCs; and (4) we present a detailed analysis on performance as well as area- and energy-efficiency to make the case for heterogeneous GPU architecture for high performance GPU computing.

The remainder of the paper is organized as follows. Section 2 gives an overview of GPU-like many-core architecture and the SIMT execution model. Section 3 presents the experimental methodology. The ILP techniques are discussed in detail in Section 4. In Section 5, we make the case for heterogeneous GPU-like many-core architecture. Section 6 discusses the related work. Section 7 concludes the paper.

## II.BACKGROUND

Recently, GPU-like many-core architectures have become a promising platform to achieve high performance computing in an energy-efficient way. The cores, referred to as shader cores (SCs), in a many-core GPU processor are organized in a hierarchical manner. A GPU contains multiple streaming multiprocessors (SMs) or compute units (CUs) and each SM/CU includes several SCs, which are also called streaming processors (SPs) or thread processors (TPs). Each SM has a register file (RF), shared memory, and an L1 (global) data cache, which are used by all the SCs in the SM.

Modern GPUs execute programs, commonly referred to as GPU kernels, in the single-instruction multiple-thread (SIMT) mode. When a GPU kernel is invoked, the threads are grouped into many thread blocks (TBs) according to the kernel invocation parameters. The TB identifier (id) and the thread id of a thread help to determine the data to be operated upon. One or more TBs can be dispatched to one SM, dependent on the register file usage and shared memory usage of a TB. Threads in a TB are organized in multiple warps (also called wavefronts). Each warp has one program counter (PC) and all the threads in the same warp execute instructions in the single-instruction multiple-data (SIMD) mode. Multiple warps from the same or different TBs can run concurrently in an SM. If there are enough concurrent warps, pipeline hazards can be effectively overcome: if one instruction, which may potentially cause a pipeline hazard, e.g., a long latency memory access or a branch instruction, gets in the pipeline, the pipeline simply executes instructions from other warps until the hazard is resolved. A warp scheduler is responsible of selecting instructions to issue from different warps and different scheduling policies have been studied, including round-robin (RR), fairness, two-level, etc. [7][8][10][17].

## III.EXPERIMENTAL METHODOLOGY

In this work, we made extensive changes to GPGPUsimV3.0.1 [12] to model the baseline GPU, as shown in Table 1, and our ILP designs. In our experiments, we vary the parameters including the register file size, core frequency, and memory bandwidth, to examine the impacts. We use the shared memory of 16kB per SM. Although recent GPUs feature higher shared memory capacity, they integrate many more SPs (or ALUs) and accommodate more TBs in each SM, therefore increasing the pressure on shared memory. We modified McPAT [13] using similar approaches to GPUWattch [12] to model the area and power.

TABLE 1 THE BASELINE GPU CONFIGURATION

| | |
|---|---|
| Shader core frequency | 325/650/1300Mhz |
| Number of SMs | 30 |
| Warp size | 32 |
| SIMD width (per SM) | 8 |
| Max. Num. of TBs Per SM | 8 TBs/1024 Threads |
| Register File size | 8k/16k/32k registers |
| Shared Memory Size Per SM | 16KB |
| Warp scheduling policy | Round robin |
| L1 Cache Per SM | 8-way set assoc. 64B cache line (48KB) |
| L2 cache | 8-way set assoc. 64B, 256 KB per memory channel |
| Number of Memory Channels | 16 |
| GDDR Memory | 8 banks, 800Mhz, total bandwidth: 200GB/S, TCL = 10, TRP = 10, TRCD = 12 |

TABLE 2 . WORKLOADS USED IN EXPERIMENTS

| Benchmark | Inputs | Grid Dim. | TB Dim. | Concurrent TBs (warps) on an SM | Total threads | Total Inst | Inst. Per core cycle |
|---|---|---|---|---|---|---|---|
| N-Queen solver (NQU) | (8,1) | (256, 1) | (96, 1) | 1 TB or 3 warps | 24576 | 2M | 22.97 |
| StoreGPU (STO) | (196625,1) | (384, 1) | (128, 1) | 1 TB or 4 warps | 49152 | 91M | 271.4 |
| Ray Tracing (RAY) | (512, 512) | (32, 32) | (16, 16) | 1 TBs or 8 warps | 262144 | 257M | 516.5 |
| MatrixMultiplication(MM) | (512, 512) | (32, 32) | (16, 16) | 4 TBs or 32 warps | 65536 | 120M | 894.2 |
| prefix-sum(Scan) | (256,1) | (256, 1) | (128, 1) | 4 TBs or 16 warps | 32768 | 10M | 579.98 |
| Convolution(Con) | (512, 512) | (32, 32) | (16, 16) | 4 TBs or 32 warps | 262144 | 520M | 872.8 |
| Fast fourier transform (FFT) | (16384,1) | (128, 1) | (64, 1) | 8 TBs or 16 warps | 8192 | 37M | 374.5 |
| BlackSchol (BS) | (480, 128) | (480, 1) | (128, 1) | 6 TBs or 24 warps | 61440 | 394M | 848.82 |
| Dxtc (DT) | (125, 125) | (1024, 1) | (128, 1) | 4 TBs or 16 warps | 131072 | 568M | 751.72 |
| ScalarProd (SCP) | (2048, 256) | (2048, 1) | (128, 1) | 6 TBs or 24 warps | 262144 | 32M | 681.96 |
| Matrix vector multiplication (MV) | (128K, 32) | (512, 1) | (256, 1) | 4 TBs or 32 warps | 131072 | 32M | 879.37 |

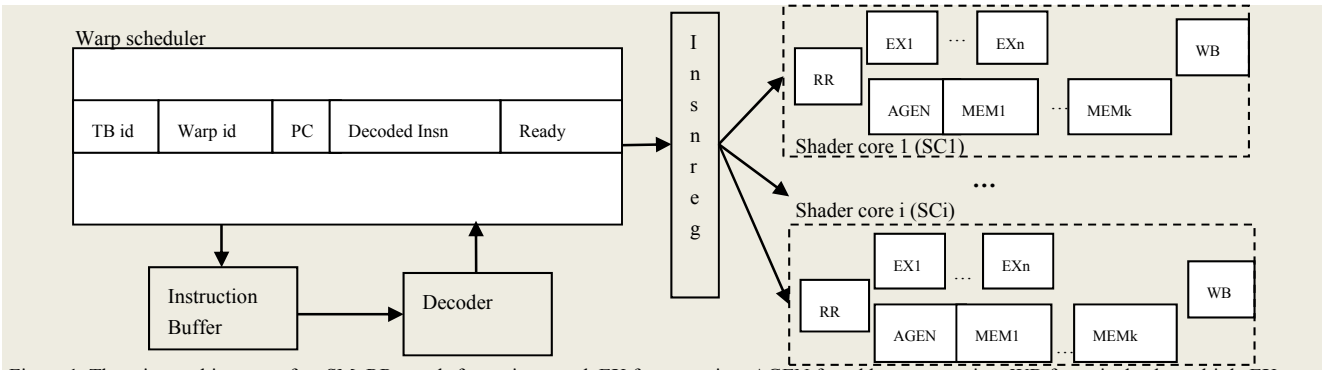


Figure 1. The microarchitecture of an SM. RR stands for register read, EX for execution, AGEN for address generation, WB for write back, multiple EX or MEM stages account for multi-cycle latency.

In our experiments, we selected 11 benchmarks from NVIDIA CUDA SDK[18] as well as the ones released along with the GPGPUsim simulator. The data inputs, the thread grid configuration, the TB configuration, the maximal number of TBs that can run in an SM, the overall number of threads, the overall number of instructions, and the baseline GPU performance measured in instructions per cycle (IPC) for each workload are shown in Table 2. As a reference, the peak IPC is 960 (= SIMD width per SM (32) x 30 SMs) in our GPU model. Among the workloads, NQU and STO have a limited number of concurrent TBs that can run on an SM. NQU and STO have high shared memory usage per TB, 15.4kB and 16kB, respectively. Therefore, each SM can only accommodate one TB at one time. In addition, for NQU, its GPU kernel contains the following if-statement '*if(idx<total_conditions){…}*', where '*idx=blockDim.x *blockid + threadid*'. As a result, TB0 (TB with blockid 0) of NQU has the highest number of instructions to execute compared to other TBs and therefore dominates the performance. Among the remaining benchmarks, RAY, Scan, and DT have a moderate number of warps (less than or equal to 16 warps) that run concurrently on an SM. For FFT, each SM is capable of running 16 warps (or 8 TBs with the TB size of 64) concurrently. Here, it is worth to point out that even the low TLP workloads, NQU and STO, have hundreds of concurrent threads running on each SM. Therefore, they fit better with GPUs and much higher performance is achieved than running on multi-threaded CPUs.

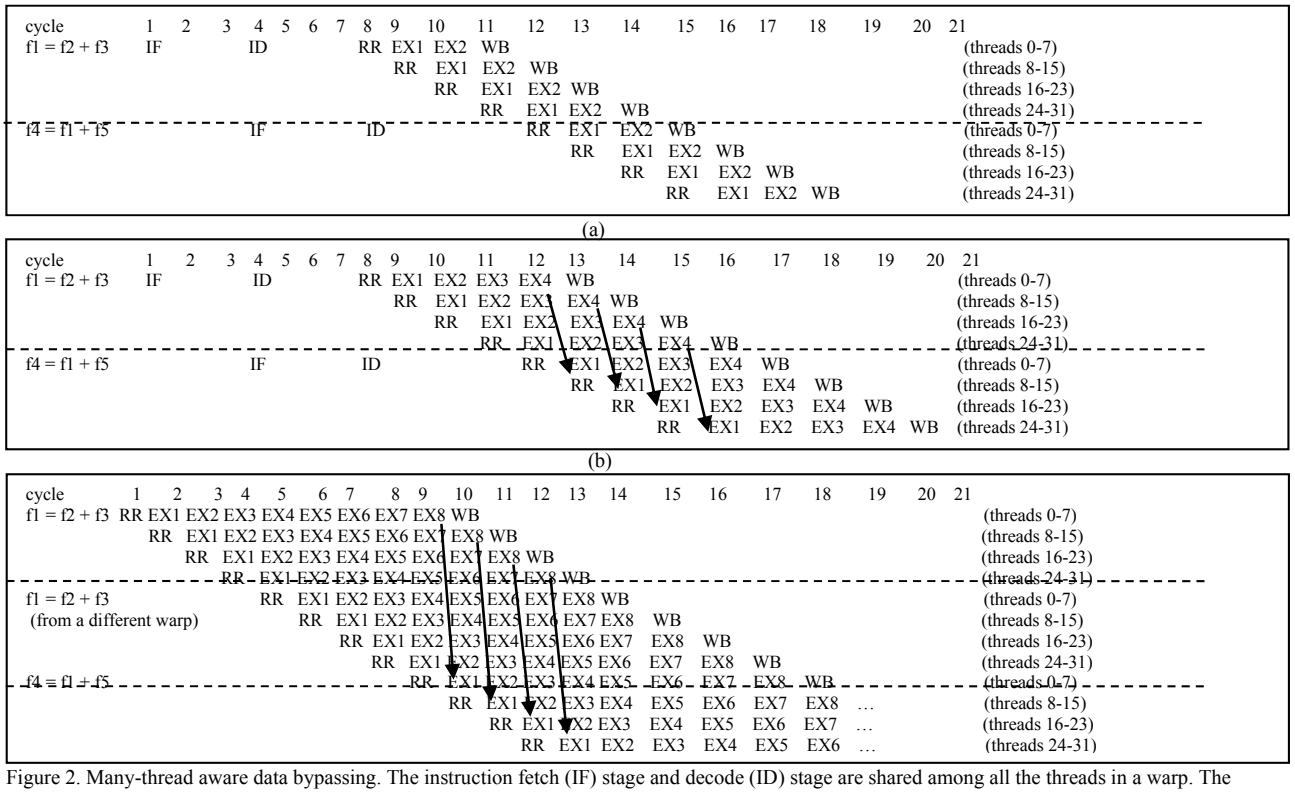


Figure 2. Many-thread aware data bypassing. The instruction fetch (IF) stage and decode (ID) stage are shared among all the threads in a warp. The register read (RR) stage, execution (EX) stage, and write back (WB) stage process 8 threads at a time as there are 8 SCs. (a) No need for data bypassing when the number of EX stages is less than the ratio of warp size/number of SCs (32/8 = 4); (b) Data bypassing from EX4->EX1 if the ALU latency is 4 cycles; (c) Data bypassing from EX8->EX1 if the ALU latency is 8 cycles.

## IV. INSTRUCTION-LEVEL PARALLELISM FOR SHADER CORES

In this section, we present our detailed many-thread aware ILP designs in throughput-oriented GPUs. Our discussion is based on the GPU model with the configuration shown in Table 1. The microarchitecture of an SM in a GPU is presented in Figure 1. In an SM, a warp scheduler selects and issues instructions from 'ready' warps to the multiple SCs. A ready warp means that the next instruction from this warp has all its dependencies resolved. Each entry in the warp scheduler contains the information of a warp. Using the program counter (PC) field, it reads the instruction from the instruction buffer. During decode, the TB id, the warp id and the thread id are used to generate the mapping from logical register numbers to physical register numbers for SCs to read from and write to the physical register file. An issued instruction will be executed by multiple SCs for all the threads in the warp. Depending on the type of the instruction, it either goes through the ALU pipeline, the memory pipeline, or the special functional units. The ready field of a warp scheduler entry is cleared when an instruction is issued from this warp. When an instruction reaches the write back stage, it uses its warp id to locate the corresponding warp scheduler entry and set the 'ready' bit.

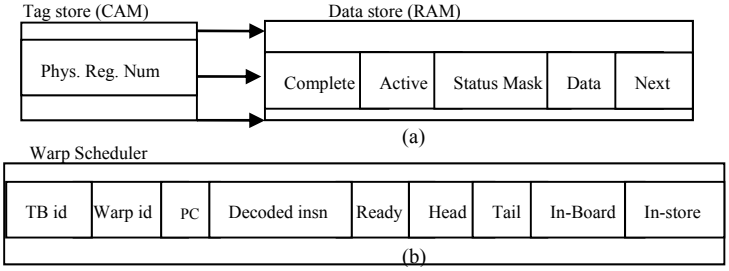


Figure 3. Architecture of (a) a many-thread aware scoreboard and (b) the associated warp scheduler.

### *4.1. Data Bypassing for Dependent Instructions*

In pipeline designs, data bypassing is an effective way to reduce the penalty of read-after-write data hazards. In an SM of a GPU, once an instruction is issued from a warp, the same instruction will be executed by the SCs for all the threads in that warp. Given the warp size of 32 and the number of SCs as 8 in an SM, the SCs will be fully utilized if one instruction can be issued every four cycles. As a result, it presents a different tradeoff for data bypassing compared to single-threaded pipelines. Considering a pair of immediate producer (f1 = f2 + f3) and consumer (f4 = f1 + f5) instructions, these two instructions can be issued back-to-back without data bypassing if latency of the producer instruction is fewer than 4 cycles, as shown in Figure 2a. The reason is that the producer results have been written to the register file (e.g., register f1 is updated in cycle 12 for threads 0-7) before the consumer instruction from the same threads (register read at cycle 13 for threads 0-7) is executed. If the producer execution latency is 4 cycles, however, the dependent instruction can only be issued when there is a data bypass path from the EX4 stage to the EX1 stage, as shown in Figure 2b. If the producer execution latency is 8 cycles, the pipeline requires at least two warps to fully utilize the pipeline with a data bypass path from EX8 to EX1, as shown in Figure 2c. In our experiments, we assume that the ALU instructions have 8-cycle latency as it is consistent with the AMD GPUs [2] which require two wavefronts (warps) to make the pipeline busy when there is such a producer-consumer pair in the code.

In summary, in our design data bypass is supported with a single bypass path (EX4->EX1 or EX8->EX1) for ALU instructions. Also, such data bypass is limited within each SP since each thread will be executed in a fixed SP (e.g., thread 0 by SP0) and there is no communication among different threads in a warp. Instructions accessing shared memory or global memory have variable latencies due to bank conflicts or cache misses. Therefore, we choose not to support data bypassing for these instructions. Instead, they wait until the data are fetched and being written to the register file before setting the corresponding warp to be ready to issue the dependent instructions. With the same warp size and an increased SIMD width, e.g., 16 or 32, the demand for TLP to hide the read-after-write data hazard becomes higher as each instruction will take fewer (2 or 1) cycles to issue.

### *4.2. Scoreboards for Independent Instructions*

When a long latency instruction, such as a cache-missing load, is issued from one warp, the stall-on-use policy, i.e., issuing subsequent independent instructions from the same warp until the consumer of the loaded value, provides opportunities to hide the latency beyond leveraging the independent warps within the same SM. To support such a stall-on-use policy, the following issues need to be addressed carefully. First, as discussed in Section 1, we argue that precise interrupts are required for next generation GPUs and we propose to support in-order instruction commit or write back. An alternative way for precise interrupts is to allow out-of-order commit and resort to checkpoints and replay to reconstruct the precise states. However, it is challenging to checkpoint periodically the aggregated architectural states of a large number of threads. Second, we need an efficient way to check data dependencies to enable independent instructions to be executed. Third, as a warp contains multiple (32) threads, control divergence (i.e., not all the threads in a warp have the same control flow) complicates the handling of data dependence. Next, we propose a cost-effective design that addresses all these challenges. The microarchitecture of our many-thread aware scoreboard is shown in Figure 3a.

As shown in Figure 3a, our proposed scoreboard is a fully-associative cache with a small number of entries (e.g., 8, see Section 4.4.1), maintaining the information of issued-but-not-yet–committed (i.e., outstanding) instructions. A tag entry in our scoreboard has a 'physical register number' field, corresponding to the destination register of an outstanding instruction. Since the registers in the GPU register file are organized as vector registers to support SIMD execution of a warp, for a register file with 16k registers and the warp size of 32, there are 512 (=16k/32) physical vector registers in the register file. Therefore, the width of each tag is 9 bits. Since the logic-to-physical register mapping ensures that the same logic register numbers from different warps will be mapped to different physical registers, there is no need to keep the

warp id information. A data entry contains a 1-bit 'completion' flag, a 32-bit 'active mask', a 32-bit 'status mask',128-byte 'data values' (corresponding to 32 registers or 1 vector register), and a K-bit 'next' field, where K is log2(num. of scoreboard entries). We also append four additional fields to each entry in the warp scheduler, including a 'head pointer' (K bits), a 'tail pointer' (K bits), a 1-bit 'in-board' flag, and a 1-bit 'in-store' flag, as shown in Figure 3b.

To support precise interrupts, outstanding instructions' execution results (or loaded values) are kept in the data value field rather than updating the register file out-of-order. If each warp has its own scoreboard, we can manage the scoreboard as a circular buffer, similar to the reorder buffer in CPU designs, to achieve in-order commit (i.e., update the register file in order). However, this approach adds too much overhead as an SM can support up to 32 concurrent warps. Therefore, we propose to share the scoreboard among all the warps in an SM. Due to such sharing, we introduce the pointers to maintain the order of outstanding instructions in each warp. The head pointer and the tail pointer in each warp scheduler entry point to the oldest and the youngest instruction, respectively, inside the scoreboard from this warp. To maintain these pointers, when an instruction from a warp is issued, if there is an unused entry in the scoreboard and there are no older instructions from the same warp in the scoreboard (i.e., when the 'in-board' bit from the warp scheduler entry is not set), the head pointer and tail pointer are set to this scoreboard entry. When another instruction is issued from the same warp, the scoreboard entry indexed by the warp's tail pointer will update its next field to point to the newly issued instruction. Then, the warp's tail pointer in the warp scheduler will be updated as well. Each cycle, each warp will use its head pointer to examine whether its oldest instruction (i.e., pointed to by its head pointer) completes. Since every instruction is executed for all the threads in a warp, the status mask is used to maintain which threads have updated their results in the data value field. If all threads complete (assuming no control divergence), the oldest instruction commits the results into the register file and then frees the scoreboard entry. In the meanwhile, the corresponding entry in the warp scheduler will update its head pointer according to the next field of the instruction to be committed. This way, in-order commit is enforced for precise interrupts.

The tag store (not the data store) in the scoreboard is used for checking data dependence. After an instruction is fetched and decoded from a warp, it searches the tag store to see whether any of its source registers hits in the tag store. If not, it means that all the source registers are available from the register file. If there is one or more hit, then the complete status bits of the corresponding entries are used to determine whether the operands can be read from the scoreboard. In either case (i.e., no tag match or matches with the complete bit being set), the ready bit in the warp scheduler entry is set so that the warp can be selected to issue this instruction. Otherwise, the warp is marked not ready as the source operand(s) of the instruction is not available yet. This way, the warp is not stalled until it encounters the consumer instruction of a long latency instruction. For example, for the following instructions in a GPU kernel, 'A: load r1, -,-; B: Add r2, -, -; C: Load r3, r2, -; D Add -, r1,-', instruction A results in a long latency cache miss. With the scoreboard, the same warp can issue the subsequent independent instruction B. After B is completed, its results are available in the scoreboard but not in the register file. Therefore, when instruction C is issued from the same warp, it searches the tag store of the scoreboard for its source operands. If there is a match (r2), the data from the corresponding data value field will be used instead of the data from the register file. Instruction D, due to its source operand match (r1) in the tag store, stalls the warp.

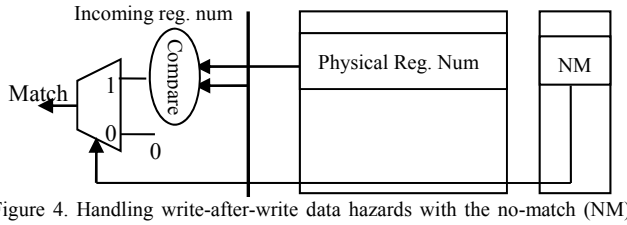

Figure 4. Handling write-after-write data hazards with the no-match (NM) flag.

Since the instructions from the same warp are issued in order, write-after-read data hazards do not present a problem. To handle write-after-write data hazards, we introduce a 'no match' (NM) bit for each tag entry in the scoreboard, as shown in Figure 4. When an instruction is issued, if it finds that there is a match in the tag store of the scoreboard with its destination register, the NM bit of the older definition is set to zero so that subsequent use will not generate a match to it. For example, for the instructions 'A: load r1,-,-; B: Add r1, -, -; C: Add -, r1, -;', when instruction B issues, it will reset the NM bit of instruction A in the scoreboard. Then, when instruction C is issued, it is guaranteed that there is only one match to r1, which is defined by instruction B. The NM bit is set when the scoreboard entry is allocated to a new instruction.

The active mask field is introduced to handle control divergence. By default, all the bits are set for this mask. When a branch results in control divergence, only the active threads along the control path will have their corresponding mask bit set, leveraging the existing branch execution logic. The completion flag is the result of inverted XOR between the active mask and the status mask, meaning that only if all active threads have updated their data values, the instruction is completed and ready to be committed. Furthermore, for an instruction having a write-after-write data hazard with an outstanding instruction in the scoreboard, it is only allowed to be issued if the active threads for this instruction are the same or the superset compared to those of the previous define instruction. Otherwise, the warp scheduler entry will be marked as 'not ready' and the warp is stalled until the previous define instruction is committed.

To handle memory dependence, we propose the following approach. When a store instruction is decoded, the warp waits for all of its outstanding instructions to be completed to ensure that the store is the oldest instruction. Then, it can issue the store instruction. This way, the precise interrupts are supported. Once a store instruction is issued, the 'in-store' flag in the warp scheduler entry is set, which is used to prevent any subsequent load instructions to be issued

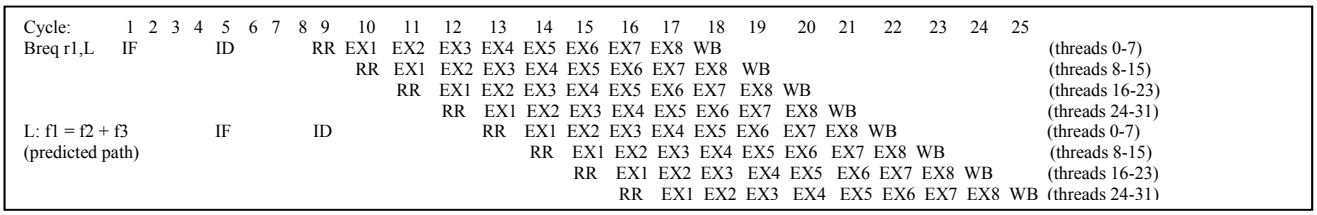


Figure 5. Branch prediction/resolution in SIMT architectures. The branch outcomes, if available at EX4 or EX8, can guard the execution or write back operations of the instruction from the predicted path.

from the same warp. The independent ALU instructions, however, can still be issued from this warp. When the store instruction commits, the 'in-store' flag in the corresponding warp scheduler entry is reset.

To reduce the structural hazard impact due to the limited size of the scoreboard, we use the 'in-board' flag for each warp to track whether there is any outstanding instruction from the warp residing in the scoreboard. If this flag is zero, meaning no outstanding instructions from the warp, and there is no free entry in the scoreboard, the warp can still issue one instruction to the SCs as it is the only instruction from this warp that will be in the pipeline. Therefore, it would not violate any dependency. In this case, after the instruction is issued, the warp scheduler entry's ready bit will be reset, preventing any subsequent instruction from this warp from being issued. The ready bit is set when the issued instruction reaches the write back stage. If the in-board flag is set, the ready bit of the warp scheduler entry is controlled by the tag store matches as discussed earlier.

In addition, the warp scheduler handles memory synchronization as well as thread execution synchronization (i.e., barrier) by marking the corresponding warps as 'not ready' until the barrier has been reached by all the warps in a thread block.

Note that our proposed scoreboard is much simpler than the reservation stations used in CPU designs. It only maintains the execution results of the issued instructions and does not have the capability to issue an instruction to the pipeline (i.e., no wake up and select logic).

### *4.3. Branch Prediction for Control Hazards*

After a warp issues a branch instruction, if there are no other ready warps, it may impose a control hazard. In GPUs, control hazards have different characteristics compared to single-threaded CPUs. First, the branch target can be computed promptly in GPUs. Due to the SIMD execution mode of a warp (8 SCs executing an instruction for 32 threads), fetching one instruction in every four cycles is sufficient to keep the pipeline busy. During the 4 cycles of IF stage, the branch target can be computed. Even with 16 SCs in an SM, there are 2 cycles to generate the target. Therefore, there is no need for a branch target buffer (BTB) as commonly used in CPU designs. It also means that unconditional branches should impose no control hazards. Second, for conditional branches, branch divergence among the threads in a warp makes them friendly to branch prediction. As shown in Figure 5, the branch instruction results in a divergence, i.e., some threads follow the taken path and some follow the not-taken path. Assume a branch takes 4 cycles to resolve (i.e., the branch outcome or the predicate bitmask is known after EX4). If a taken prediction is made, right before the instruction at the predicted target L is executed, the bitmask from the branch has already been computed for the corresponding threads. For example, at cycle 14 before the EX1 of the instruction L for threads 0-7, the bit mask of these threads has been computed and can be used to void the computation for the threads that should follow the not-taken path. Even if the branch is resolved at EX8 stage, it can still be used to predicate the write back operation of instruction L. Therefore, divergent branches can be predicted either taken or not-taken and either prediction will not cause any misprediction penalty. In other words, divergent branches are immune to mispredictions. For non-divergent branches, a misprediction incurs a recovery by nullifying the instructions from the predicted path, similar to CPU designs. The only difference is that rather than nullifying all the instructions in the pipeline after the branch, we only need to nullify those from the same warp. To make predictions for conditional branches, we can use hardware based predictors such as gshare predictors [14]. Such a gshare predictor can be shared by all the thread blocks in the same SM. For divergent branches, since either taken or not taken is a correct prediction, the predictor is updated as if it makes a correct prediction. A more cost-effective (nearly free) way is to use the simple heuristics 'backward taken and forward not-taken'. Since the target can be computed during the IF stage, it can be used to compare to the branch PC to make a prediction without any prediction tables. We refer to this approach as 'static' prediction in this paper. Note that, we will not make predictions of a branch, which is dependent upon a long latency operation, since either the stall-on-miss (i.e., without scoreboards) or the stall-on-use policy stalls the warp and prevents the branch from entering the pipeline.

### *4.4. Experimental Results*

#### *4.4.1. Performance impacts of reducing data hazards*

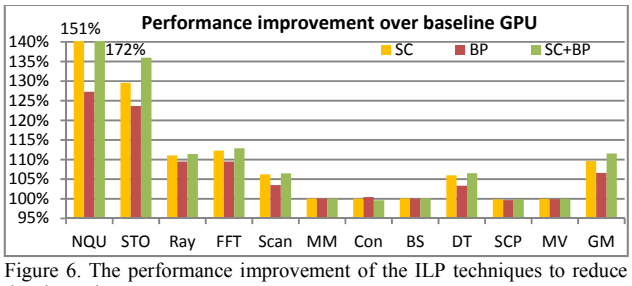


Figure 6. The performance improvement of the ILP techniques to reduce data hazards

In our first experiment, we evaluate the performance gains from reducing data hazards. Since the bypassing and the scoreboard techniques target different types of data hazards, we can combine the two. The performance results,

normalized to the baseline GPU, of the bypassing (labeled 'BP'), the scoreboard (labeled 'SC'), and the combined bypassing and scoreboard (labeled 'BP+SC') are shown in Figure 6. The scoreboard has 8 entries, translating into the following hardware overhead. The tag store of the scoreboard is an 8-entry CAM with each entry containing 9 (Physical reg. number) + 1 (NM) = 10 bits. The data store of the scoreboard has a total of 8 x (1+32+32+128x8+3) = 8736 bits (=1092 B). Compared to the large register file size (16k registers=16k x 4B=65536 B) in each SM, such overhead is quite limited.

From Figure 6, we can see that the data bypassing and scoreboard techniques show variable performance gains on different workloads. Among them, two benchmarks, NQU, and STO, benefit significantly from reducing data hazards. The main reason is that these two benchmarks have relatively low TLP due to their resource usage and the ILP techniques effectively utilize the otherwise stall cycles. For benchmarks with moderate degrees of TLP, including RAY, FFT, Scan, and DT, reducing data hazards also shows good performance gains, ranging from 6.5% (Scan) to 12.9% (FFT). For benchmarks with high degrees of TLP (MM, Con, BS, SCP, and MV), reducing data hazards shows almost no performance gains as there are enough threads/warps to overlap the latency introduced by data hazards. Overall, on average using the geometric mean (GM), the data bypassing scheme introduces 6.6% performance improvements and the scoreboard has 9.7% performance gains. When the two techniques are combined, there is an average of 11.5% performance enhancement.

We also vary the scoreboard size to evaluate the impact of the structural hazards. And we found out that an 8-entry scoreboard is the most cost-effective choice, which is used in the remaining experiments in the paper.

In the next experiment, we change the SC frequency (from 325MHz to 1300MHz) and the memory bandwidth (from 100GB/s to 400GB/s). The experimental results show similar performance gains (ranging between 12.0% and 14.6% on average) when both data bypassing and scoreboard techniques are used.

We also vary the physical register file size from 8k to 32k registers to analyze the impact. With a small register file in an SM (e.g., 8k registers), the number of concurrent TBs in the SM is reduced compared to larger register files. The performance gains from the bypassing and scoreboard techniques on GPUs with different register file sizes are shown in Figure 7.

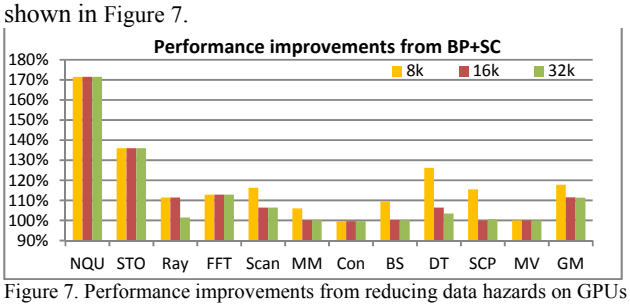


Figure 7. Performance improvements from reducing data hazards on GPUs with different register file sizes.

From Figure 7, we can see that the benchmarks Scan, MM, BS, DT and SCP benefit more from bypassing and scoreboards when the register file is small (8k) and limits the number of concurrent TBs. Further increasing the register file size from 16k to 32k cannot provide additional TLP for Scan, MM and SCP as each SM already runs the maximum number of TBs or threads. For NQU and STO, the resource bottleneck is shared memory. Therefore, we observe similar results to those with the register file size of 16k registers. For the benchmark, Ray, we do not have the result for 8k registers because such a register file is not enough to launch 1 TB, and when we increase the register file size from 16k to 32k, the number of concurrent TBs increases from 1 to 2, therefore the bypassing and scoreboard techniques are less effective for the 32k case compared to the 16k case.

### *4.4.2 Performance impacts of reducing control hazards*

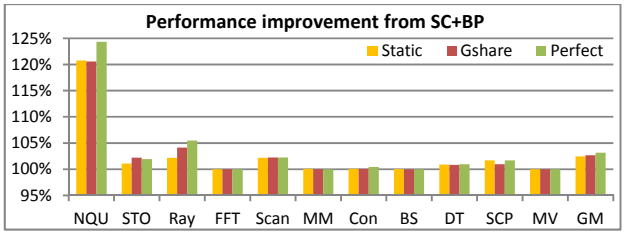


Figure 8. The performance improvements from branch prediction on the baseline GPU equipped with the bypassing and scoreboard techniques.

In this experiment, we first examine the performance impact of branch prediction. We changed the GPGPUsim to reduce the branch latency to 1 cycle to reduce the control hazards in the baseline GPU. In other words, the instruction at the right target of the branch is ready to be issued one cycle after the branch enters the EX stage. In Figure 8, we report the performance improvements over the GPU with the bypass and scoreboard techniques. The predictions are made using the static predictor as well as the gshare predictor discussed in Section 4.3. For reference, we also report the performance results with perfect branch predictions (i.e., no mispredictions).

From Figure 8, we can see that the benchmark NQU benefits significantly from either static or gshare predictors, 20.8% for our static predictor and 20.6% for the gshare predictor over the baseline GPU with bypassing and scoreboard. The remaining benchmarks, however, have smaller impacts, including 4.1% for Ray and 2.2% for Scan. The reason is that in the TB0 of NQU (the performance dominant TB), 13.7% of all its instructions are branches. Among the branches, 42.3% of them do not present a control hazard for the baseline GPU (i.e., without bypassing or scoreboard) due to TLP. The remaining 57.7% result in pipeline stalls. As discussed in Section 4.3, either correct predictions or divergent branches can effectively utilize the otherwise stall cycles, thereby showing high performance gains for NQU. Other benchmarks have less number of branches and there are more warps running concurrently on an SM to overlap the pipeline hazards, thereby showing relatively small performance impacts.

In the next experiment, we examine the predictability of non-divergent conditional branches in the GPU kernels. We report the branch prediction accuracies of our simple static approach and a 1k-bit gshare predictor (shared by all warps in an SM) in Figure 9 (FFT has no conditional branches and is excluded from the figure). From the figure, we can see that in GPU kernels branches can be predicted reasonably well

using our simple static approach (95.8% accuracy on average). The gshare predictor is also effective (96.4%). The reason is that all warps within the same SM share the same predictor and therefore they can accurately update the branch predictor for each other.

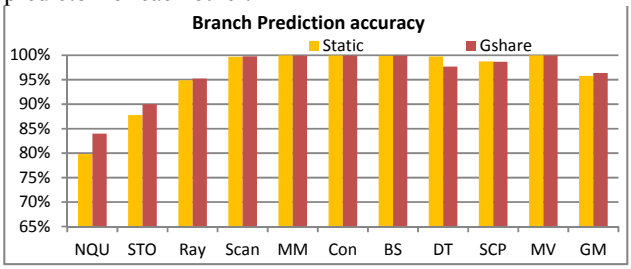


Figure 9. The prediction accuracy for conditional non-divergent branches of different prediction schemes.

Overall, from the results, we can see that although on average the performance gains of branch prediction is moderate (2.5% for our static branch predictor and 2.7% for the gshare branch predictor), it can be quite effective for control intensive benchmarks with low TLP (e.g., NQU). In addition, as discussed in Section 4.3, branch prediction is almost free in terms of hardware cost using our simple static prediction approach. Therefore, we argue that it is a technique to adopt for next generation GPUs.

### *4.4.3. Interaction between ILP and TLP*

Either ILP or TLP can be leveraged to overcome pipeline hazards. The overhead to support the ILP techniques includes additional bypass paths and scoreboards, as discussed in Sections 4.1-4.3. The cost of supporting TLP, on the other hand, is the aggregated resource requirement from a high number of threads. Among such resources, register files maintain the register state of the threads. To study the interaction between ILP and TLP, in the next experiment, we vary the register file size in each SM from 4k to16k registers and evaluate the performance of the GPU model with and without the ILP techniques. The performance results, normalized to our baseline GPU model with a register file having 16k registers, are reported in Figure 10. The benchmark RAY requires at least 10k registers to run one thread block. Therefore, there are no performance results for small register files for this benchmark. When calculating the average using GM, the performance of 10k registers is used for small register files (4k, 6k and 8k registers) for this benchmark.

Two important observations can be made from Figure 10. First, on average, the performance gains for the ILP techniques tend to reduce with a higher degree of TLP as a result of large register files. Among individual benchmarks, for NQU, STO, RAY, and FFT, the performance is not affected by the register file size as the factors such as shared memory other than the register file size are limiting their TLP. The benchmark Scan is sensitive to the warp scheduling policy and performs the best when the register file has 6k registers. For the rest benchmarks, large register files support a higher number of concurrent threads and lead to better performance. Higher degrees of TLP also reduce the effectiveness of the ILP techniques. Take SCP as an example, when the register file has 4k registers, adding ILP techniques improves the performance from 44.8% to 65.6% of the baseline GPU. With a register file containing 12k registers, the improvement from the ILP techniques is from 95.3% to 96.9% of the baseline.

Second, the ILP techniques provide an interesting tradeoff for resources required for high TLP. As seen from Figure 10, on average, adding the ILP support to a GPU with the register file size of 8k registers outperforms the baseline GPU with the register file size of 16k registers by 9.1%. In other words, the ILP techniques can save 8k registers in each register file while achieving higher performance, thereby being potentially more area and energy efficient. We leverage this observation in Section 5 to customize our designs.

## V. HETEROGENEOUS GPU ARCHITECTURE

The results discussed in Section 4.4 show that the ILP techniques have different performance impacts on different applications. To efficiently handle such workload-dependent behavior, we propose a heterogeneous GPU architecture and it contains two types of cores: one with the ILP techniques (referred to as ILP cores) and one without them (referred to as TLP cores). A kernel with high TLP fits the TLP cores while a kernel with limited TLP benefits more from the ILP

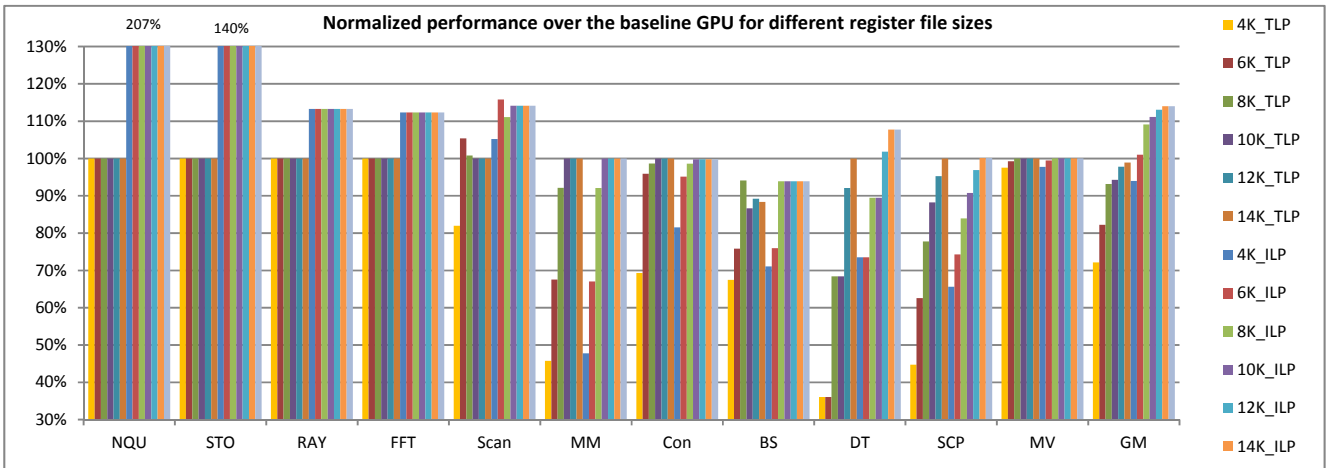


Figure 10. Performance comparisons across different register file sizes. The legend: the number means the register file size and '_ILP/TLP' means with/without ILP techniques.

cores. To classify whether a kernel has high TLP or not, we simply use the number of concurrent warps that can run on an SM. This information depends on the resource usage of a kernel and is readily available at the compile time. If a kernel can have more than 24 concurrent warps (i.e., 24x32=768 threads) running concurrently on an SM, it is classified as high TLP. Next, we compare it with homogeneous TLP cores and homogeneous ILP cores.

First, we modified McPAT [13] using the similar approach to GPUWattch [12] to model the area and power of an SM based on the configuration shown in Table 1. The 40nm technology is used to achieve the target core frequency of 1.3GHz and the operating temperature is 380K. The reported area of an SM of the baseline GPU (or an SM with TLP cores) is 11.823 mm2. The data bypassing support in each SC introduces an area overhead of 0.028 mm2. As there are 8 SCs in an SM, the overall area cost of data bypassing in an SM is 0.224 mm2. The scoreboard support, shared by all SCs in an SM, costs 0.917 mm2. Therefore, an SM with ILP cores has an area of 12.964 mm2, which means a 9.6% area overhead for the ILP techniques. The branch prediction support only requires an adder to compute the target and a comparator between the target and PC. Since such logic has very small area overhead and the current McPAT tool does not model it explicitly, we ignore the area cost of branch prediction support.

Considering the die area that is similar to an NVIDIA GTX480 GPU, we can have the following options: 40 SMs with TLP cores (40x11.823 = 472.92 mm2); 36 SMs with ILP cores (36x12.964 = 466.704 mm2); and 20 SMs with TLP cores plus 18 SMs with ILP cores (20x11.823 + 18*12.964 = 234.46 + 233.352 = 467.812 mm2). We refer to the first one as GPU-T, the second one as GPU-I, and third one as GPU-H. The first two are homogeneous designs and the last one is heterogeneous. There are many possible combinations of ILP and TLP cores for heterogeneous designs. Our choice is based on the option that either type of cores has the same/similar area (234.46 mm2 vs. 233.352 mm2). Compared to our baseline GPU in Table 1, these GPUs have higher number of SMs. Therefore, to maintain the ratio of compute-to-memory bandwidth, we increase the number of memory modules per memory controller from 2 to 3.

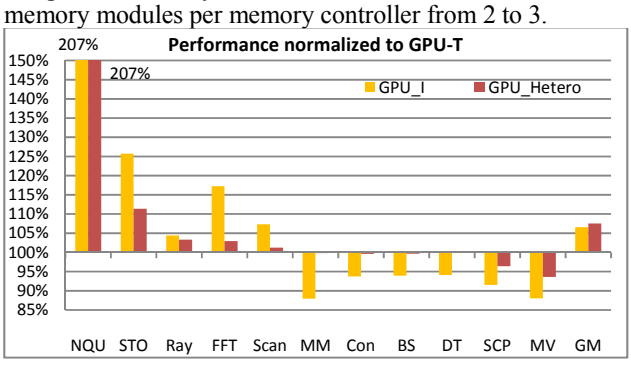


Figure 11. The performance of different GPUs running single kernels.

Next, we run a single kernel on these three different GPU architectures and the performance results, normalized to GPU-T, are shown in Figure 11. Since GPU-T, GPU-I, and GPU-H have similar areas, the performance also represents their area efficiencies.

From Figure 11, it can be seen that when running a single kernel, GPU-I is most effective for applications with limited TLP (NQU, STO, Ray, and Scan) and GPU-T is most effective for applications with high TLP (MM, Conv, BS, DT, SCP and MV). For FFT, although it has reasonably high number of concurrent warps (16) to run on each SM, its overall number of threads (16384) is limited. Therefore, GPU-I achieves higher performance for FFT than GPU-T or GPU-H. Compared to GPU-I and GPU-T, GPU-H presents a nice tradeoff. On average, GPU-H achieves 7.5% higher performance than GPU-T and higher performance compared to GPU-I (6.5%).

Similar trends are observed for energy consumption of three GPU designs. GPU-I consumes lower energy than GPU-T running applications with limited TLP and consumes higher energy than GPU-T running high TLP workloads. GPU-H achieves a good balance between them.

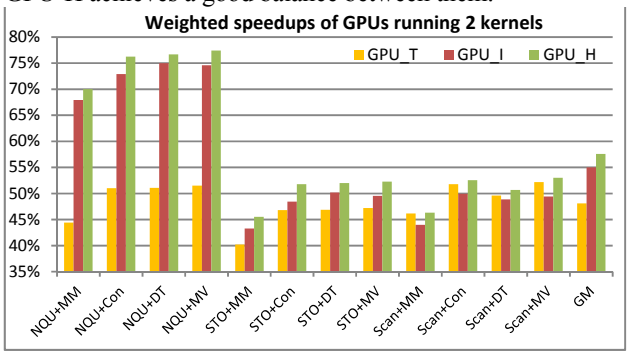


Figure 12. The performance of different GPUs running concurrent kernels (one favors ILP and the other favors TLP).

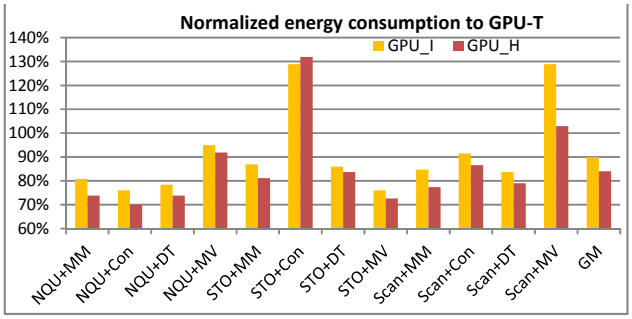


Figure 13. The energy consumption of different GPUs running concurrent kernels (one favors ILP and the other favors TLP).

Then, we run heterogeneous concurrent kernels on these three GPU designs. The heterogeneous kernel mixes are constructed using the following way: one from NQU, STO, or Scan, which favors ILP cores, and the other from MM, Con, DT or MV, which favors the TLP cores. To overcome the issue that different workloads have different number of instructions, we add a loop to each kernel to make it run continuously and then stop execution when the longer kernel finishes at least 1 run. We adopt the weighted speedups [21] from multi-threaded CPU designs to evaluate the performance. Assuming kernel 1 is the one with low TLP and kernel 2 is the one with high TLP. The weighted speedup is computed as follows: $\frac{1}{2} * (\frac{IPC_{GPU}(kernel1)}{IPC_{GPU-I}(kernel1)} + \frac{IPC_{GPU}(kernel2)}{IPC_{GPU-T}(kernel2)}$, where IPC GPU(kernel) is the instructions per cycle of the kernel running on the GPU of interest, $IPC_{GPU-I}(kernel1)$ is the performance of the kernel 1 running on GPU-I, and

$IPC_{GPU-T}(kernel2)$ is the performance of the kernel 2 running on GPU-T. The application steering policy used in GPU-H is mainly based on how many concurrent warps that an SM can run. If this number is less than 16, the application is dispatched to the 18 SMs with ILP cores. Otherwise, it is dispatched to the 20 SMs with TLP cores. If an application has limited overall number of threads (<=32768), it is dispatched to ILP cores. As the resource requirement of a kernel is available statically at compile time, the support for our application steering policy is straightforward.

The performance results of GPU-I, GPU-T, and GPU-H are shown in Figure 12. For concurrent kernels containing NQU and STO, the GPU-I outperforms GPU-T, the reason is that the performance gains from GPU-I for NQU and STO are higher than the performance gains from GPU-T for MM, Con, DT or MV. For concurrent kernels containing Scan, the performance of GPU-T is better because the benefit of GPU-T for MM, Con, DT or MV is larger than the performance gains from GPU-I for Scan. In all these cases, GPU-H outperforms either homogeneous design and on average, GPU-H achieves 4.7% and 19.8% higher performance than GPU-I and GPU-T, respectively. We also collect the energy consumption results of GPU-I, GPU-T, and GPU-H, and report them in Figure 13. As each concurrent kernel contains two applications, the energy consumption is the sum of the energy consumed by executing either application from the beginning to end. From the figure, we can see that on average, GPU-H consumes least energy, 16.0% less than GPU-T and 6.5% less than GPU-I.

## VI. Related Work

Most research work on GPU architectures focuses on reducing the impact of divergent behavior [6][7][15] within warps and improving GPU memory hierarchy[11], rather than architecting SCs. To our knowledge, this is the first work to examine the impact of ILP techniques on SCs for many-core GPUs. Among the ILP techniques, data bypassing support exists in AMD GPUs. Although the detailed architecture has not been disclosed, it is stated in R-700 ISA [2] that special instruction encoding (PS or PV) is used to receive the result of previous instruction. Certain compiler optimizations, such as loop unrolling, assume that GPUs can execute independent outstanding loads from the same warp. However, current GPUs do not support precise interrupts and it is not clear how the read-after-write and write-after-write data hazards are resolved after an older pending long latency instruction. We are not aware of any prior work on branch prediction on GPUs or heterogeneous GPU architectures.

For concurrent kernel execution, [1][20] show that scheduling concurrent kernels on different SMs is more beneficial than letting one kernel occupy all the SMs. Our approach takes one step further by customizing the SMs based on the characteristics of the concurrent kernels.

## VII. Conclusions

Given the high computational throughput and energy efficiency, GPU-like many-core architectures have been increasingly adopted to build supercomputers. In this paper, we focus on the basic building block of the GPU, the shader cores. We present a detailed study to reveal the effectiveness of the ILP on the throughput-oriented many core architectures. We show that for applications, whose resource requirements prevent GPUs from running a sufficient number of concurrent threads, the ILP techniques can significantly improve the performance. Furthermore, we show that the ILP techniques can be used to reduce the TLP requirement and therefore reduce the requirement in critical resources such as the register file. We then propose heterogeneous GPU architectures to include both ILP cores (i.e., cores with the ILP techniques) and TLP cores (i.e., cores without them). Applications are steered to either type of cores depending on how many threads can run concurrently on the hardware. Our results show that the proposed heterogeneous GPUs have higher performance, area- and energy-efficiency than homogenous GPUs based on either ILP or TLP cores.

## Acknowledgment

We thank the anonymous reviewers for their insightful comments to improve our paper. This work is supported by an NSF project 1216569 and a gift fund from AMD Inc.

## References

[1] J. T. Adriaens, et al., The GPGPU Spatial Multitasking, HPCA 2012.
[2] AMD R700-Family Instruction Set Architecture Reference, Guide February 2011.
[3] A. Bakhoda, et al., Analyzing CUDA workloads using a detailed GPU simulator. IPASS 2009.
[4] GTX 680 Kepler Whitepaper.
[5] R. Espasa, M. Valero, and J.E. Smith, Out-of-Order Vector Architectures, MICRO 1997.
[6] W. Fung, et al., Thread block compaction for efficient SIMT control flow, HPCA 2011.
[7] W Fung, et al., Dynamic warp formation and scheduling for efficient GPU control flow, MICRO 2007.
[8] A. Jog, et al, OWL:Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In ASPLOS, 2013
[9] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers.
[10] N. Lakshminarayana et al., Effect of instruction fetch and memory scheduling on GPGPU performance, Workshop on Lang. Comp. and Arch. Support for GPGPU, 2010.
[11] J. Lee, et al., Many-thread aware prefetching mechanisms for GPGPU applications. MICRO 2010.
[12] J. Leng, et al., GPUWattch: Enabling Energy Optimizations in GPGPUs, ISCA, 2013
[13] S. Li at al., McPAT: an integrated power, area and timing modeling framework for multicore and manycore architectures, MICRO 2009.
[14] S. McFarling, Combining branch predictors, Digital Western Research Lab (WRL) Technical Report, 1993.
[15] J. Meng, et al. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In ISCA, 2010.
[16] J. Menon et al., iGPU: Exception support and speculative execution on GPUs, ISCA 2012.
[17] V. Narasiman, et al, Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In MICRO, 2011
[18] NVIDIA GPU Computing SDK 3.1.
[19] NVIDIA CUDA Programming Guide 4.1, 2011.
[20] V.T. Ravi, et al, Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In HPDC, 2011
[21] D. Tullsen et al., Handling long-latency loads in a simultaneous multithreading processor, MICRO 2001.