Elastic Pipeline: Addressing GPU On-chip Shared Memory Bank Conflicts

Chunyang Gou Georgi N. Gaydadjiev
Computer Engineering Lab
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, The Netherlands
{C.Gou, G.N.Gaydadjiev}@tudelft.nl

ABSTRACT

One of the major problems with the GPU on-chip shared memory is bank conflicts. We observed that the throughput of the GPU processor core is often constrained neither by the shared memory bandwidth, nor by the shared memory latency (as long as it stays constant), but is rather due to the *varied latencies* caused by memory bank conflicts. This results in conflicts at the writeback stage of the in-order pipeline and pipeline stalls, thus degrading system throughput. Based on this observation, we investigate and propose a novel *elastic pipeline* design that minimizes the negative impact of on-chip memory bank conflicts on system throughput, by decoupling bank conflicts from pipeline stalls. Simulation results show that our proposed elastic pipeline together with the co-designed *bank-conflict aware warp scheduling* reduces the pipeline stalls by up to 64.0% (with 42.3% on average) and improves the overall performance by up to 20.7% (on average 13.3%) for our benchmark applications, at trivial hardware overhead.

Categories and Subject Descriptors: C.1.2[Multiple Data Stream Architectures (Multiprocessors)]:SIMD; B.3.2[Design Styles]:Interleaved memories
General Terms: Design, Performance

1. INTRODUCTION

The trend is quite clear that multi/many-core processors are becoming pervasive computing platforms nowadays. GPU is one example that uses massive lightweight cores to achieve high aggregated performance, especially for highly data-parallel workloads. Although GPUs are originally designed for graphics processing, the performance of many well tuned general purpose applications on GPUs have established them among one of the most attractive computing platforms in a more general context – leading to the GPGPU (*General-purpose Processing on GPUs*) domain[2].

In manycore systems such as GPUs, massive multithreading is used to hide long latencies of the core pipeline, interconnect and different memory hierarchy levels. On such heavily multithreaded execution platforms, the overall system performance is significantly affected by the efficiency of both on-chip and off-chip memory resources. As a rule, the factors impacting the on-chip memory efficiency have quite different characteristics compared to the off-chip case. For example, on-chip memories tend to be more sensitive to dynamically changing latencies, while bandwidth limitations are more severe for off-chip memories. In the particular case of GPUs, the on-chip first level memories, including both the software managed shared memory and the hardware cache, are heavily banked, in order to provide high bandwidth for the parallel SIMD lanes. Even with adequate bandwidth provided by the parallel memory banks, however, applications can still suffer drastic pipeline stalls, resulting significant performance loss. This is due to unbalanced accesses to the on-chip memory banks. This increases the overhead in using on-chip shared memories, since the programmer has to take care of the bank conflicts. Furthermore, often the GPGPU shared memory utilization range is constrained due to such overhead.

In this paper, we observed that the throughput of the GPU processor core is often hampered neither by the on-chip memory bandwidth, nor by the on-chip memory latency (as long as it stays constant), but rather by the varied latencies due to memory bank conflicts, which end up with writeback conflicts and pipeline stalls in the in-order pipeline, thus degrading system throughput. To address this problem, we will investigate novel *elastic pipeline* design that minimizes the negative impact of on-chip memory bank conflicts on system throughput.

This paper makes the following specific contributions:

- We analyzed the GPU on-chip shared memory bank conflict problem, and identified how the bank conflicts are translated into pipeline performance degradation;
- We investigate and propose a novel *elastic pipeline* design that minimizes the negative impact of on-chip shared memory conflicts on overall system throughput, by cutting the link between bank conflict and pipeline stall;
- We co-designed *bank-conflict aware warp scheduling* to assist our elastic pipeline hardware;
- We carefully simulated our proposal and have shown pipeline stalls reduction of up to 64.0% leading to overall system performance improvement of up to 20.7%.

The remainder of the paper is organized as follows. In Section 2, we provide the background and motivation for this work. In Section 3, we discuss our proposed elastic pipeline design. The co-designed bank-conflict aware warp scheduling technique is elaborated in Section 4. Simulated performance of our proposed elastic pipeline in GPGPU applications is evaluated in Section 5, followed by some general discussions of

CF'11, May 3–5, 2011, Ischia, Italy.

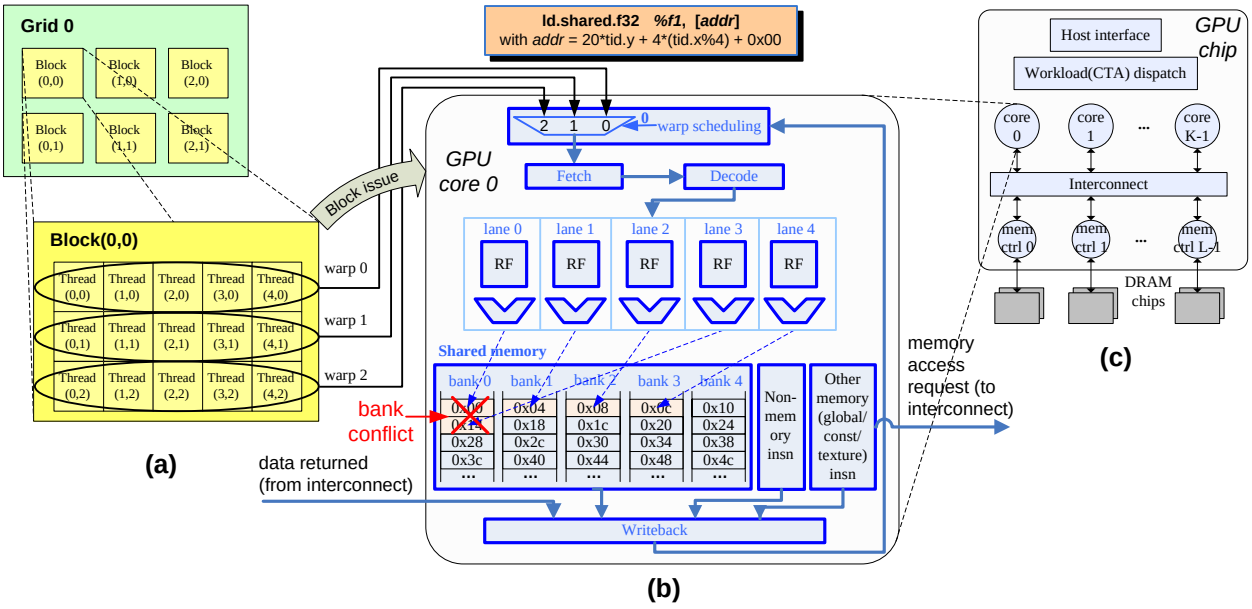

Figure 1: (a)CUDA threads hierarchy; (b)thread execution in GPU core pipeline; (c)GPU chip organization

our simulated GPU core architecture along with the elastic pipeline in Section 6. The major differences between our proposal and related art are described in Section 7. Finally, Section 8 concludes the paper.

2. BACKGROUND AND MOTIVATION

In this section, we will first introduce some GPGPU related background and their shared memory accesses. Then we provide a motivation example.

2.1 Shared Memory Access on GPGPU

GPU utilization has spanned far beyond graphics rendering, covering a wide spectrum of general purpose computing known as GPGPU[2]. The programming models of GPGPU (such as OpenCL[4] and CUDA[19]) are generally referred to as *explicitly-parallel, bulk-synchronous* SPMD (*Single Program Multiple Data*). In such programming models, the programmer extracts the data-parallel section of the sequential application code, identifies the basic working unit (typically an element in the problem domain), and explicitly expresses the same sequence of operations on each working unit in a *kernel*. Multiple kernel instances (called *threads* in CUDA) are running independently on GPU cores. The parallel threads are organized into a two-level hierarchy, in which a kernel (*grid* in CUDA) consists of parallel CTAs (*Cooperating Thread Array*, or *block* in CUDA), with each CTA composed by parallel threads, as shown in Figure 1(a). Explicit, localized synchronizations and on-chip data sharing mechanisms (such as CUDA shared memory) are supported inside each CTA.

During execution, a batch of threads from the same CTA are grouped into a *warp*, which is the smallest unit for the pipeline front-end processing (i.e., warp scheduling, fetching and decoding stages in Figure 1(b)) in GPU cores, as illustrated in Figure 1. For high efficiency, warps are executed on the fine-grain multithreaded GPU core in a SIMD fashion. Figure 1 shows a warp configuration of 5 threads per warp and a SIMD data path consisting of 5 lanes. The warps are scheduled and issued to the pipeline in an *interleaved* manner which is also known as *barrel processing*[22].

GPUs rely mainly on massive hardware multithreading to hide external DRAM latencies. In addition, on-chip memory hierarchies are also deployed in GPUs in order to provide high bandwidth and low latency. Such on-chip memories include, software managed caches (shared memory), or hardware caches, or a combination of both[13]. To provide adequate bandwidth for the GPU parallel SIMD lanes, the shared memory is heavily banked. However, when accesses to the shared memory banks are unbalanced, shared memory bank conflicts occur. For example, with the memory access pattern shown on top of Figure 1(b), data needed by both lanes 0 and 4 reside in the same shared memory bank 0. In this case *hot bank* is formed at bank 0, and the two *conflicting* accesses have to be *serialized*, assuming a single-port shared memory design[1]. As a result, the GPU core throughput may be substantially degraded, as to be exemplified by the following example.

2.2 Motivation Example

A snapshot of the AES encryption kernel source is shown in Figure 2. The code shown there deals with the second encryption stage. First, the stage input data indexes are loaded from shared memory region *stageBlock2* (phase I). Then the stage input data are loaded from shared memory regions *tBox*Block* (phase II), with the indexes from phase I. Afterwards the data is processed (phase III), and finally stored to the shared memory region *stageBlock1* (phase IV). The other stages of the encryption process work similarly. In phase II an irregular access pattern called *indirection* or *gather* is required, causing shared memory bank conflicts during AES execution. As a result, the kernel suffers from a large number of pipeline stalls and non-trivial performance loss. With our proposed elastic pipeline design (Section 3) together with the bank-conflict aware warp scheduling technique (Section 4), the number of

[1]Even with dual-port shared memory banks, such serialization can not be completely avoided when the bank conflict degree is higher than two.

```
__global__ void aesEncrypt128( unsigned * result, unsigned * inData,
int inputSize) {
__shared__ UByte4 tBox0/1/2/3Block[256];
__shared__ UByte4 stageBlock1/2[BSIZE];
unsigned tx = threadIdx.x;
...
unsigned op1 = stageBlock2[posIdx_E[mod4tx*4]   + idx2].ubval[0];
unsigned op2 = stageBlock2[posIdx_E[mod4tx*4+1] + idx2].ubval[1];
unsigned op3 = stageBlock2[posIdx_E[mod4tx*4+2] + idx2].ubval[2];
unsigned op4 = stageBlock2[posIdx_E[mod4tx*4+3] + idx2].ubval[3];
op1 = tBox0Block[op1].uival;
op2 = tBox1Block[op2].uival;
op3 = tBox2Block[op3].uival;
op4 = tBox3Block[op4].uival;
...
stageBlock1[tx].uival = op1^op2^op3^op4^keyElem;
...
}
```

(I) *shared memory loads:* ***no bank conflict***

shared memory loads: ***bank conflicts!*** (II)

(III)

(IV) *shared memory store:* ***no bank conflict***

Figure 2: AES source code

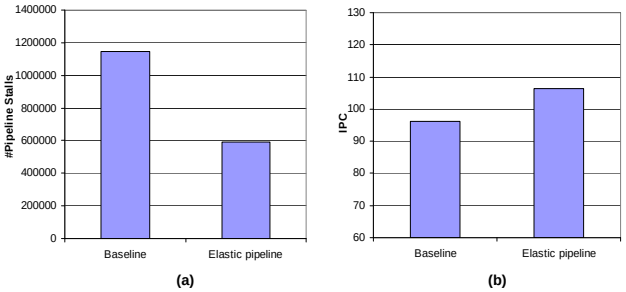

Figure 3: Effect of elastic pipeline in: (a)reducing pipeline stalls and (b)improving performance

pipeline stalls is reduced by 48.2%, which is translated to the overall performance improvement by 10.5%, as Figure 3 shows.

3. ELASTIC PIPELINE

In this section, we will first analyze the mechanism through which shared memory bank conflicts degrade GPU pipeline performance. Then our elastic pipeline design and its implementation are presented, with emphasis on the conflict tolerance, hardware overhead and pipeline timing impact.

3.1 How Are Bank Conflicts Translated into Pipeline Performance Degradation

The baseline in-order, single-issue GPU core pipeline configuration is illustrated on top of Figure 4(a). The warp scheduling stage is not shown, and only one of the parallel SIMD lanes of the execution/memory stages is shown in Figure 4(a) for simple illustration[2]. Meanwhile, although only sub-stages of the memory stage (*MEM0/1*) are explicitly shown in the figure, other stages are also pipelined for increased execution frequency[3]. t_i denotes execution time in cycle i, and W_i denotes warp instruction fetched in cycle i.

For easy discussion, we have the definition of *bank conflict degree* of SIMD shared memory access in the following:

Bank conflict degree: the maximal number of simultaneous accesses to the same bank during the same SIMD shared memory access from the parallel lanes.

Following this definition, the conflict degree of a SIMD shared memory access ranges from 1 to *simd_width*. For example, the SIMD shared memory access in Figure 1(b) has a conflict_degree of 2. In general, it takes $\left\lceil \frac{conflict\_degree}{\#shared\_memory\_ports} \right\rceil$ cycles to read/write all data for a SIMD shared memory access.

As Figure 4(a) shows, W_i is a shared memory access with a conflict degree of 2, and it suffers from shared memory bank conflict in cycle i+3, at *MEM0* stage. The bank conflict holds W_i at *MEM0* for an additional cycle (assuming single port shared memory), until it gets resolved at the end of cycle i+4. In the baseline pipeline configuration with unified memory stages, the bank conflict in cycle i+3 has two consequences: (1) it blocks the upstream pipeline stages in the same cycle, thus inuring a pipeline stall which is finally observed by the pipeline front-end in cycle i+4; (2) it introduces a bubble into the *MEM1* stage in cycle i+4, which finally turns into a write-back bubble in cycle i+5.

Notice the fact that it is unnecessary for execution of W_{i+1} to be blocked by W_i, if W_{i+1} is not a shared memory access. Thus a possible pipeline configuration which is able to eliminate the above mentioned consequence (1) of the baseline can be considered, as Figure 4(b) shows. With the help of the extra *NONMEM* path, W_{i+1} is now no longer blocked by W_i, instead it steps into the *NONMEM* path while W_i is waiting at stage *MEM0* for the shared memory access conflict to be resolved, as Figure 4(b) shows. Unfortunately, this cannot avoid the writeback bubble in cycle i+5. Moreover, the bank conflict of W_i in cycle i+3 causes writeback conflict[4] at the beginning of cycle i+6, which finally incurs a pipeline stall at fetch stage in the same cycle, as shown in Figure 4(b).

Our observation: Through the above analysis, we can see that the throughput of the GPU core is constrained neither by the shared memory bandwidth, nor by the shared memory latency (as long as it stays constant), but rather by the varied execution latencies due to memory bank conflicts. The variation in execution latency incurs writeback bubbles and writeback conflicts, which further cause pipeline stalls in the in-order pipeline, thus hurting system throughput.

3.2 Elastic Pipeline Design

To address the problem discussed above, we propose an elastic pipeline design which is able to eliminate the negative impact of shared memory bank conflicts on system throughput, as shown in Figure 5. Compared with the baseline pipeline with split memory stages in Figure 4(b), the major change is the added buses to forward *matured instructions* from *EXE* and *NONMEM0/1* stages to the writeback stage. This effectively turns the original 2-stage *NONMEM* pipeline into a 2-entry FIFO queue (we will refer to it as "*NONMEM* queue" hereafter). Note, the output from the *EXE* stage can be forwarded directly to writeback only if it is **not** a memory instruction, whereas forwarding from *NONMEM0* to writeback is always allowed. Such non-memory instructions can bypass some or all memory stages, simply because they do not need any processing by the memory pipeline. As Figure 5 shows, by forwarding matured instructions in the *NONMEM* queue to the writeback stage, the writeback conflict is removed, and thus the link between bank conflict and writeback conflict is cut off and the pipeline stall due to bank conflict is eliminated.

[2]Please refer to Figure 1(b) for the pipeline details.

[3]We adopt a 24-stage pipeline (Section 5). Similar pipeline configurations are widely used in contemporary GPU cores[18, 25] and research GPU microarchitectural models[5].

[4]Note the writeback throughput for a single issue pipeline is 1 instruction/cycle at maximum.

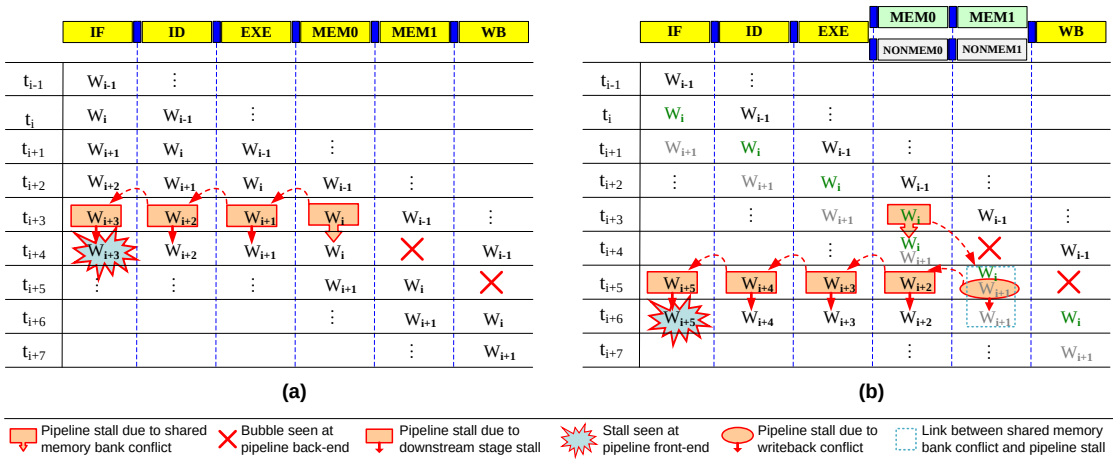

Figure 4: Baseline in-order pipeline: (a)unified memory stages and (b)split memory stages

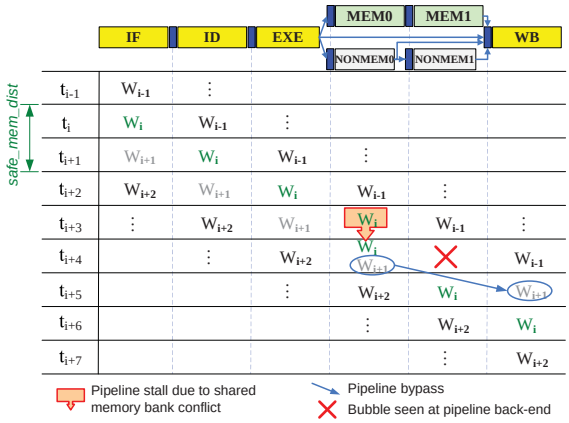

Figure 5: Elastic pipeline

3.2.1 Safe Scheduling Distance and Conflict Tolerance

For easy discussion, we first define warp types as follows:

Memory warp: a warp which is ready for pipeline scheduling and is going to access any type of memory (e.g., shared/global/constant) in its next instruction execution.

Shared memory warp: a ready warp which is going to access on-chip shared memory in its next instruction execution.

Non-memory warp: a ready warp which is not going to access any memory type during its next instruction execution.

In Figure 5 it is assumed that W_{i+1} is a non-memory instruction. Otherwise, W_{i+1} will be blocked at *EXE* stage in cycle i+4, since W_i is pending at *MEM0* in the same cycle, due to its shared memory bank conflicts. Such a problem exists even if W_{i+1} is not a shared memory access[5]. To avoid this problem, we have the constraint of *safe memory warp schedule distance*, defined as:

Safe memory warp schedule distance: the minimal number of cycles between the scheduling of a shared memory warp and a following memory warp, in order to avoid pipeline stall due to shared memory bank conflicts.

It is easy to verify the relationship between $safe\_mem\_dist$ (short for "safe memory warp schedule distance") and the shared memory bank conflict degree, in the following equation:

$$safe\_mem\_dist = \left\lceil \frac{conflict\_degree}{\#shared\_memory\_ports} \right\rceil \quad (1)$$

The safe memory warp schedule distance constraint requires that memory warps should not be scheduled for execution in next $safe\_mem\_dist - 1$ cycles after a bank-conflicting shared memory warp is scheduled. For example, $safe\_mem\_dist$ for W_i in Figure 5 is $\lceil \frac{2}{1} \rceil = 2$, which means that in the next cycle, only non-memory warp can be allowed for scheduling.

It is important to point out that, the elastic pipeline handles bank conflicts of *any degree* without introducing pipeline stalls, as far as the $safe\_mem\_dist$ constraint is met. We will discuss this in more detail in Section 4.

3.2.2 Out-of-order Instruction Commitment

In Figure 5, the elastic pipeline shows the behavior of out-of-order instruction commitment. For GPU cores with *strict barrel processing* (Section 6) (assumed in our evaluation), it is not a problem since the in-flight instructions are from different warps. In the case of *relaxed barrel processing* in which consecutively issued instructions may come from a same warp (but without data dependence), out-of-order instruction commitment within the same execution context may occur. This is penalized by the pipeline being unable to support precise exception. Possible solutions are discussed in Section 6.

3.2.3 Extension for Large Warp Size

Above we have assumed *#warp_size=simd_width*. In real GPU implementations, however, the number of threads in a warp can be a multiple of GPU core pipeline SIMD width[6]. In this case, a warp is divided into smaller *subwarps* with the size of each equaling the number of SIMD lanes. All subwarps from the same warp are executed by the SIMD pipeline consecutively. Therefore, *warp_size/simd_width* free issue slots are needed for a warp to be completely issued into the pipeline. Moreover, each warp will occupy the SIMD pipeline for at least *warp_size/simd_width* cycles during execution. Consider for example $warp\_size/simd\_width = 2$. In this case both W_i and W_{i+1} in Figure 5 will have to execute the same shared

[5]Note, in such case, even if there exists a third path (with fixed number of stages) for that memory access type, write-back bubbles cannot be avoided, due to the same phenomenon illustrated in Figure 4(b).

[6]For example, there are 32 threads per warp in CUDA and 8 SIMD lanes in NVIDIA GPGPUs before Fermi[13].

memory access instruction for the first and second half of the same warp, respectively. Since W_i is blocked at stage *MEM0* in cycle i+4, W_{i+1} is unable to step into *MEM0* from the *EXE* stage at the beginning of the same cycle. This results W_{i+1} being blocked at *EXE* and all upstream pipeline stages being blocked in cycle i+4, thus incurring a pipeline stall.

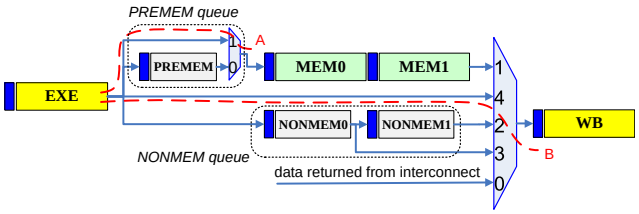

Figure 6: Elastic pipeline logic diagram

To solve this problem, an extension has to be adopted to the elastic pipeline shown at the top of Figure 5. In the extension, we introduce another source of elasticity to the *MEM* path, by placing before the *MEM0* stage a $(warp\_size/simd\_width-1)$-entry FIFO queue ("*PREMEM* queue" in Figure 6). With the help of the *PREMEM* queue, the elastic pipeline can handle all consecutive, back-to-back issued bank-conflicting SIMD shared memory accesses from the same warp, regardless of the conflict degree of each.

The logic diagram of the final elastic pipeline with the extension for large warp size is shown in Figure 6, for the case with 2 memory stages and 1-entry *PREMEM* queue. The numbers inside the multiplexers denote the MUX inputs priority (smaller numbers have higher priorities).

With the elastic pipeline configuration of Figure 6, W_{i+1} in Figure 5 will be buffered in the *PREMEM* queue in cycle i+4, while W_{i+2} will directly step into writeback stage at the beginning of cycle i+5.

To summarize, the elastic pipeline adds two FIFO queues to the baseline pipeline: the *NONMEM* queue with a depth of M and the *PREMEM* queue with a depth of N, where

$$M = \#MEM\_stages \tag{2}$$

$$N = \left\lceil \frac{warp\_size}{simd\_width} \right\rceil - 1 \tag{3}$$

3.2.4 Hardware Overhead and Impact on Pipeline Timing

Table 1: Elastic pipeline HW overhead per GPU core

Type	Logic complexity	Quantity
Pipeline latches	$simd\_width$	M+N
(M+3)-to-1 MUX	M+2	1
(N+1)-to-1 MUX	N	1

The additional hardware overhead as compared with the baseline pipeline is summarized in Table 1. The metric for the logic complexity of pipeline latches is that of a pipeline latch in a single SIMD lane. As we can see in Table 1, the area consumption of the additional pipeline latches is in the order of $(M + N) \cdot simd\_width$. Considering small Ms and Ns in realistic GPU core pipeline designs (e.g. M=4, N=3 in our evaluation), this additional cost is well acceptable. The hardware overhead of the two multiplexers is negligible.

The control paths of the two multiplexers are not shown in Figure 6, since they are simply valid signals from relevant pipeline latches at the beginning of each stage, and are therefore not in the critical path. Compared with the baseline pipeline, all other pipeline stages' timing is untouched, with only one exception of the *EXE* stage, as illustrated in Figure 6[7]. There are two separate paths in which the *EXE* stage is prolonged: path A and B, as marked by the two dash lines in Figure 6. A is the (N+1)-to-1 multiplexer and B is the (M+3)-to-1 multiplexer listed in Table 1. With standard critical path optimizations such as the *priority on late arriving signal* technique[6], both A and B only incur an additional latency of 2-to-1 MUX for the *EXE* stage. Therefore, the increased latency to stage *EXE* is that of a 2-to-1 MUX in total, which will not noticeably affect the target frequency of the pipeline in most cases (assumed in our experimental evaluation).

4. BANK-CONFLICT AWARE WARP SCHEDULING

As discussed in Section 3.2.1, in order to completely avoid the pipeline stall due to shared memory bank conflicts, the constraint of *safe memory warp schedule distance* must be satisfied. Otherwise, two consequences will happen: 1) the *PREMEM* queue will get saturated, which results in pipeline stalls; and 2) the *NONMEM* will get emptied, which results in writeback starvation. In the end the pipeline throughput is degraded. In order to cope with this problem, warp scheduling logic should prevent any memory warp from being scheduled in the time frame of *warp_safe_mem_dist* (Equation (5)) cycles after a bank-conflicting shared memory warp is scheduled for execution. This is called "*bank-conflict aware warp scheduling*", which will be discussed in the following.

4.1 Obtaining Bank Conflict Information

In order to apply bank-conflict aware warp scheduling, we have to first find out which instructions will cause shared memory bank conflicts, and their corresponding conflict degree. This information may be obtained in two ways: 1) static program analysis; 2) dynamic detection. We chose dynamic bank conflict detection instead of compile-time analysis in our implementation for two reasons. First, some shared memory access patterns (and therefore the bank conflict patterns) are only known at runtime. This is the case for regular access patterns (e.g. 1D strided) whose pattern parameters (e.g., the stride) are unknown at compile time, or irregular accesses the bank conflict patterns of which can not be identified statically (such as the AES example). Second, there is no additional hardware cost incurred directly by the dynamic detection itself, as the shared memory bank conflict detection logic is also needed in the baseline pipeline[8].

Note, for warp sizes larger than the number of SIMD lanes, the bank conflict degree of the *entire warp* is the accumulation of all subwarp SIMD accesses, as given by the following equation:

$$warp\_bkconf\_degree = \sum_{i=0}^{\lceil \frac{warp\_size}{simd\_width} \rceil - 1} conflict\_degree_i \tag{4}$$

[7]Note, although not shown in Figure 4(a), there is a MUX at the end of *MEM1* stage in the baseline pipeline, since an arbitration to select writeback data from either inside the GPU core pipeline or from the interconnect is needed.

[8]The shared memory has to identify the conflict degree of each SIMD shared memory access (i.e., $conflict\_degree_i$ in Equation (4)) in order to resolve it.

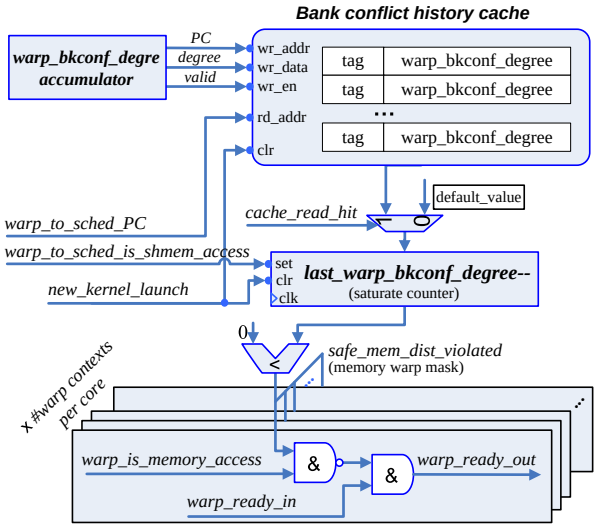

Figure 7: Bank-conflict aware warp ready signal generation

where $conflict\_degree_i$ is the shared memory bank conflict degree of subwarp i, which is measured by the hardware dynamically. $warp\_bkconf\_degree$ is obtained by an accumulator and a valid result is generated at fastest every $\lceil \frac{warp\_size}{simd\_width} \rceil$ cycles (if there is no pipeline stall during that time).

Accordingly, the safe memory warp schedule distance in Equation (1) is extended in the following:

$$warp\_safe\_mem\_dist = \left\lceil \frac{\sum_{i=0}^{\lceil \frac{warp\_size}{simd\_width} \rceil - 1} conflict\_degree_i}{\#shared\_memory\_ports} \right\rceil \quad (5)$$

And the safe memory warp schedule distance constraint now requires that the scheduling interval between bank-conflicting shared memory warp and memory warp should be no less than $warp\_safe\_mem\_dist$ cycles.

It is very important to note that, the bank conflict degree of the latest scheduled shared memory warp can not be obtained *in time* by simply checking the $warp\_bkconf\_degree$ accumulator on the fly. This is due to the fact that it may have not reached memory stages or finished shared memory accesses yet when its bank conflict information is needed by the warp scheduling logic. Therefore, we need to **predict** $warp\_bkconf\_degree$ for a shared memory warp before the real value becomes valid, by only its shared memory instruction PC. In our design, we implement a simple prediction scheme which predicts the bank conflict degree of a shared memory instruction to be the one measured during the *latest execution* of the same instruction.

4.2 Bank Conflict History Cache

In order to maintain the historic conflict degree information, we implement a small private *bank conflict history cache* distributed in each GPU core, as shown in Figure 7. At each time a new kernel is launched, both the bank conflict history cache and the *last_warp_bkconf_degree* counter are cleared. The cache is updated whenever a warp execution of shared memory instruction gets resolved and the $warp\_bkconf\_degree$ accumulator generates a valid value for it[9]. Whenever a shared memory warp is scheduled, the *last_warp_bkconf_degree* counter is set to its latest warp bank conflict degree in history, by checking it in the conflict history cache. If a cache miss occurs, then the *last_warp_bkconf_degree* counter is set to a default value (0 in our design). The memory warp mask is generated by checking if the safe memory warp schedule distance constraint is violated. Note, in our design we assume the warp scheduling stage knows whether or not a ready warp is a shared memory access (the *"warp_to_sched_is_shmem_access"* signal in Figure 7), or a memory access (the *"warp_is_memory_access"* signal). This can be done easily with negligible overhead. For example, we can look up the committing warp's next instruction type in a per-core type bit-vector (initialized at kernel launch time) at pipeline writeback stage (only 2 bits per PC per kernel is enough for this purpose), and setup the 2-bit type register associated with the committing warp (only 2 bits per a hardware warp context).

When we use the bank conflict history cache to predict the conflict degree of scheduled shared memory access, the result is incorrect in two situations: (1) when a cache miss happens (e.g., compulsory misses due to the cold cache after a new kernel is launched) and unfortunately the default output value generated is different from the actual conflict degree; (2) when the conflict degree of the same shared memory instruction varies among consecutive execution. Case (1) is unavoidable for any kernel. Fortunately, its impact on overall performance is usually negligible. Case (2) occurs only in kernels with irregular shared memory access patterns and dynamically changing conflict degree (e.g., AES). It is important to note that, incorrect prediction of the shared memory bank conflict degree in the elastic pipeline will not always result in pipeline stalls. Indeed, the pipeline will be stalled only when the predicted value is *smaller* than the actual conflict degree and there is at least one memory warp scheduled which violates the safe memory warp schedule distance constraint. We will see the impact of incorrect bank conflict degree prediction on pipeline stalls in Section 5.1.

4.3 The proposed Warp Scheduling

With the bank access conflict history for each shared memory instruction maintained in the conflict history cache, bank-conflict aware warp scheduling can apply the same scheduling scheme as the baseline pipeline to schedule the ready warps for execution. The only difference is that if a previously scheduled warp *will be/is still being* blocked at the memory stages due to shared memory bank conflicts, then all memory warps are excluded from the ready warp pool, as Figure 7 shows.

Once it is guaranteed by the warp scheduling logic that there is no memory warp violating the safe memory warp schedule distance constraint, then shared memory bank conflicts incurred by a single warp can be effectively handled by the elastic pipeline design, as discussed in Section 3.2. Otherwise, the elastic pipeline will get saturated and stalls due to bank conflicts will occur. We will see the impact of the bank-aware warp scheduling on overall performance in Section 5.2. It should be noted that, warp scheduling just by itself is unable to reduce pipeline stalls caused by shared memory bank conflicts, without the elastic pipeline infrastructure.

[9]Note, in our design the conflict degree value from the accumulator has been decreased by $\lceil \frac{warp\_size}{simd\_width} \rceil$ before written to the conflict history cache, in order to *align* the value for instruction with no bank conflict to zero.

4.4 Hardware Overhead

Table 2: Hardware overhead of bank conflict prediction and warp mask generation (per GPU core)

Type	Logic complexity	Quantity
Bank conflict history cache	$\#cache\_lines \cdot (log_2(warp\_size) + 14\text{-}log_2(\#cache\_sets))$ bits, dual port (1R+1W)	1
AND/NAND gate	–	2·#warp contexts per core
Accumulator	$log_2(warp\_size)$ bits	1
2-to-1 MUX	–	1
Counter	$log_2(warp\_size)$ bits	1
Comparator	$log_2(warp\_size)$ bits	1

As discussed above, our bank-conflict aware warp scheduling does not incur any additional overhead in scheduling logic – it simply utilizes the same scheduling as the baseline. However, the warp ready signal generation logic needs to be modified to make it be aware of in-flight bank-conflicting shared memory accesses and enforce the constraint on following warps to be scheduled, as shown in Figure 7. Table 2 summarizes the hardware overhead incurred by the bank conflict degree prediction and bank-conflict aware warp ready signal generation. The main contributor in Table 2 is the bank conflict degree history cache. Assuming 14 bits PC (which is able to handle kernels with up to 16K instructions – large enough from our experience), this turns to 14-$log_2(\#cache\_sets)$ bits cache tag size. Remember, each $conflict\_degree_i$ in Equation (4) takes $log_2(simd\_width)$ bits (Section 3.1), therefore the cache content *warp_bkconf_degree* occupies $log_2(warp\_size)$ bits. The total size of the bank conflict history cache is summarized in Table 2. In our design, we implemented a 2-way set associative conflict history cache with 256 sets, which is capable of removing all capacity and conflict misses for all kernels in our evaluation. In this case, the conflict history cache consumes only 704 bytes (with *warp_size*=32), which is quite trivial.

Regarding the timing impact, the increase in the warp ready signal generation delay observed by the default warp scheduler is only that of one *AND* gate, as shown in Figure 7.

5. EXPERIMENTAL EVALUATION

Experimental Setup: We use a modified version of GPGPU-Sim[5], which is a cycle level full system simulator for GPUs implementing ptx ISA[20]. We model GPU cores with a 24-stage pipeline similar to contemporary implementations[18, 25]. The detailed configuration of the GPU processor is shown in Table 3. The GPU processor with the baseline pipeline ("baseline GPU") and the case with the proposed elastic pipeline ("enhanced GPU") are evaluated in this paper. They differ only in the core pipeline configurations and warp scheduling schemes, as Table 3 shows. The number of memory pipeline stages and the *warp_size/simd_width* ratio are 4 (see Table 3). Therefore the queue depth is set to 4 for the *NONMEM* queue, and 3 for the *PREMEM* queue in the elastic pipeline, according to Equations (2) and (3).

We select 8 shared memory intensive benchmarks from CUDA SDK[19] and other public sources[17]. Table 4 lists the main characteristics of the selected benchmarks. The instruction count in columns *total instructions* and *shared memory instructions* shows two numbers, with the first being the number of dynamic instructions executed by all 128 scalar pipelines

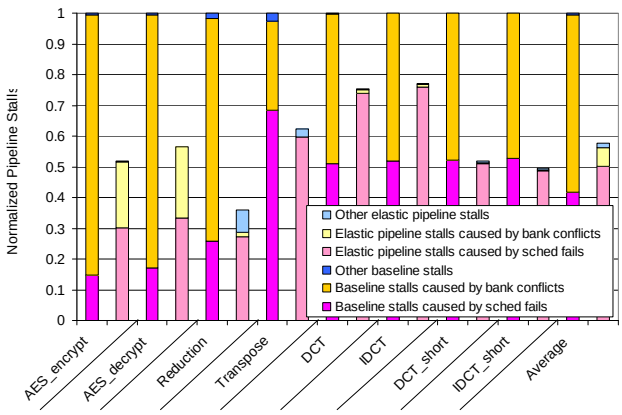

Figure 8: Pipeline stall reduction. In each group: left bar: baseline GPU; right bar: elastic pipeline enhanced GPU

(i.e., SIMD lanes) of 16 GPU cores, and the second number being the ptx instruction count in the compiled program. *Shared memory intensity* is the ratio of dynamic shared memory instructions to total executed instructions. The *average bank conflict degree* field shows the average number of cycles spent on a SIMD shared memory access for each benchmark application. This is collected by running the benchmarks on the baseline GPU. *Theoretic speedup* calculates, assuming IPC=1 (normalized to a single scalar pipeline/SIMD lane) for all instructions except shared memory accesses (i.e., all pipeline inefficiency comes from pipeline stalls caused by shared memory bank conflicts), the speedup that can be gained by eliminating all pipeline stalls. *CTAs per core* denotes the maximal number of concurrent CTAs that can be allocated on each GPU core. A **Y** in the *Irregular shared memory patterns* column indicates kernels with shared memory instructions with irregular access patterns and dynamically varied bank conflict degree.

Note, the kernel names followed by a ⋇ denote the CUDA code which has originally been hand-optimized to avoid shared memory bank conflicts, by changing the layout of the data structures in shared memory (e.g., by padding one additional column to a 2D array). We adopt the code but *undo* such optimizations in our evaluation of elastic pipeline performance in Sections 5.1 and 5.2. There are two reasons for this. First, we found that in practice if the shared memory bank conflict is a problem, the programmer will either remove it (by the above mentioned hand-optimizations), or simply avoid using the shared memory. Due to this we were unable to find many existing code with heavy shared memory bank conflicts. That is why we manually *roll back* the shared memory hand-optimizations for these kernels and use them in our initial evaluation presented in this paper. Second, assuming the elastic pipeline is adopted in the GPU core, we also want to inspect how it performs for these kernels, without specific shared memory optimizations from the programmer.

5.1 Effect on Pipeline Stall Reduction

Figure 8 shows the proposed elastic pipeline and the bank-conflict aware warp scheduling effect on reducing pipeline stalls. The results are per kernel, with the left bar of each group showing the number of pipeline stalls in the baseline GPU, and the right bar showing the stalls in the enhanced GPU. The number of stalls are normalized to the baseline GPU. Inside each bar,

Table 3: The GPU processor configurations

Number of Cores	16
Core Configuration	8-wide SIMD execution pipeline, 24 pipeline stages (with 4 memory stages) 32 threads/warp, 1024 threads/core, 8 CTAs/core, 16384 registers/core execution model: strict barrel processing (Section 6) warp scheduling policy: Round-robin (**baseline GPU**) **vs** bank-conflict aware warp scheduling (**enhanced GPU**) pipeline configuration: baseline pipeline (**baseline GPU**) **vs** elastic pipeline (**enhanced GPU**)
On-chip Memories	16KB software managed cache (i.e., shared memory)/core, 8 banks, 1 access per core cycle per bank
DRAM	4 GDDR3 memory channels, 2 DRAM chips per channel, 2KB page per DRAM chip, 8 banks 8 Bytes/channel/transmission (51.2GB/s bandwidth in total), 800 MHz bus freq, 32 DRAM request buffer entries memory controller policy: out-of-order (FR-FCFS)[21]
Interconnect Network	crossbar, 32-Byte flit size

Table 4: Benchmark Characteristics

Benchmark name	Source	Grid Dimensions	CTA Dimensions	CTAs per core	Total instructions	Shared memory instructions	Shared memory intensity	Average bank conflict degree	Theoretic speedup	Irregular shared memory pattern
AES_encypt	Other[17]	(257,1,1)	(256,1,1)	2	35132928/534	12697856/193	36.1%	1.54	1.19	Y
AES_decypt	Other[17]	(257,1,1)	(256,1,1)	2	35527680/540	12763648/194	35.9%	1.53	1.19	Y
Reduction	CUDA SDK	(16384,1,1)	(256,1,1)	4	415170560/49	16744448/5	4.0%	3.07	1.09	Y
Transpose*	CUDA SDK	(16,16,1)	(16,16,1)	4	3538944/54	131072/2	3.7%	4.50	1.13	N
DCT*	CUDA SDK	(16,32,1)	(8,4,2)	7	7274496/222	1572864/48	21.6%	3.33	1.50	N
IDCT*	CUDA SDK	(16,32,1)	(8,4,2)	7	7405568/226	1572864/48	21.2%	3.33	1.50	N
DCT_short*	CUDA SDK	(16,16,1)	(8,4,4)	7	10223616/337	1048576/40	10.3%	2.75	1.18	N
IDCT_short*	CUDA SDK	(16,16,1)	(8,4,4)	7	10190848/336	1048576/40	10.3%	2.75	1.18	N

the pipeline stalls are broken down into three categories (from bottom to top): warp scheduling fails, shared memory bank conflicts, and other reasons (i.e., writeback conflicts incurred by data returned from interconnect).

As Figure 8 shows, the number of pipeline stalls are significantly reduced by the elastic pipeline. In all kernels except AES encrypt/decrypt, the pipeline stalls caused by bank conflicts are almost completely removed in the enhanced GPU. Remember, the bank conflict stalls in the elastic pipeline may occur, only if the conflict degree prediction made by the bank conflict history cache is incorrect (Section 4.2). The bank conflict history cache was unable to produce constantly precise conflict degree prediction for the highly irregular shared memory access patterns in the AES kernels. This results in a large number of bank conflict stalls.

On the other hand, the number of pipeline stalls due to warp scheduling failures are increased for some kernels. This is expected, since the bank-conflict aware warp scheduling masks off the ready warps which violate the constraint of safe memory warp schedule distance. Contrary to our expectation, the number of warp scheduling fails is actually reduced for Transpose and DCT/IDCT_short kernels. Detailed investigation reveals that this is related to the inter operation between our elastic pipeline design and the rest of the GPU processor, such as the on-chip synchronization and control flow re-convergence mechanisms, and off-chip DRAM organizations. For example, drastic DRAM channel conflicts are observed during the Transpose kernel execution on the baseline GPU. Whereas in the GPU enhanced by the elastic pipeline and the bank-conflict aware warp scheduling, such channel conflicts are substantially reduced and DRAM efficiency is improved.

The last type of pipeline stalls (*other pipeline stalls* in Figure 8) is caused by writeback conflicts incurred by data returned from interconnect. The number is slightly increased in the elastic pipeline as shown in Figure 8. This is because the number of such conflicts is relatively small, and a large portion of them are well *hidden* by the large amount of bank conflict stalls at the upstream of the pipeline, in the baseline GPU.

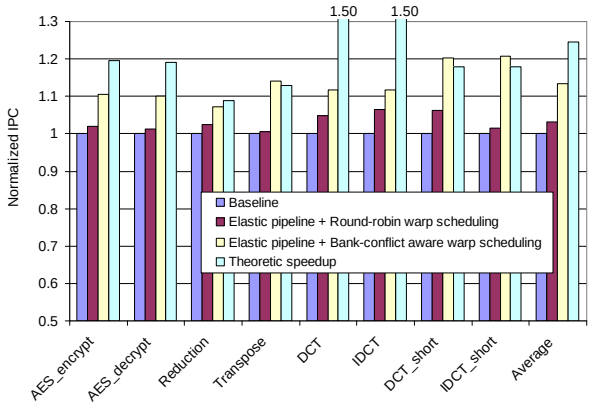

Figure 9: Performance improvement

5.2 Performance Improvements

Figure 9 compares the performance of the baseline GPU, the enhanced GPU with pure elastic pipe design (with default warp scheduling), the enhanced GPU with elastic pipeline augmented by bank-conflict aware warp scheduling, and the theoretic speedup. We can see that the performance is improved by the pure elastic pipeline only slightly (3.2% on average), without the assistance of proper warp scheduling. While with the co-designed bank-conflict aware warp scheduling, an additional 10.1% improvement is gained, leading to the average performance improvement of the elastic pipeline design by

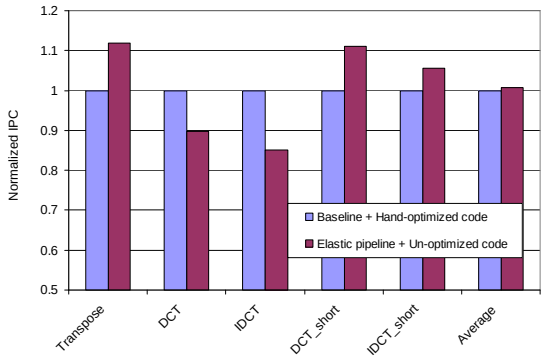

Figure 10: Elastic pipeline vs hand-optimized code for conflicting kernels

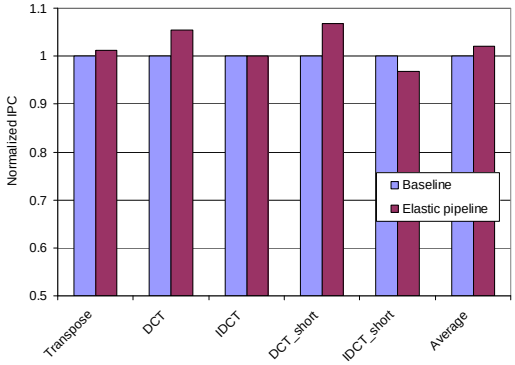

Figure 11: Elastic pipeline performance for non-conflicting kernels

13.3%, as compared to the baseline. This confirms our analysis in Section 4. For AES encrypt/decrypt, the achieved speedup by the elastic pipeline is substantially smaller than the theoretical bound, mainly because a large portion of bank conflict stalls still remain in the elastic pipeline, as shown in Figure 8. DCT and IDCT see a *huge* gap between the actually achieved performance gain by elastic pipeline and the theoretic bound. This is due to the number of pipeline stalls caused by warp scheduling fails is significantly increased, also shown in Figure 8. It is interesting to see that the speedup of our elastic pipeline design exceeds the theoretic bound, for kernels Transpose and DCT/IDCT_short. This results from the fact that the number of warp scheduling fails is reduced, thanks to the positive interaction between the elastic pipeline and the rest of the system in these cases, as discussed in Section 5.1.

In order to find out how our elastic pipeline performs in relieving the overhead of reducing shared memory bank conflicts from software side, we compared the performance of un-optimized code (i.e., CUDA SDK code with shared memory bank conflict optimizations removed by us) running on the enhanced GPU versus the hand-optimized code (i.e. the original CUDA SDK code) running on the baseline GPU, as shown in Figure 10. As we can see from the figure, on average the performance of un-optimized kernel running on elastic pipeline cores is on par with the optimized kernel running on baseline cores. However, we also found that for DCT/IDCT kernels, the performance gap is quite large. In-depth analysis reveals that this is due to the change of warp execution order by the elastic pipeline interacts poorly with the global memory access which results in degraded DRAM access efficiency. This actually leaves the room for further optimizations. For example, a simple variant of our bank-conflict aware warp scheduling allows issuing of memory instructions *violating* the safe memory warp schedule distance, if there is no ready warps to execute non-memory instruction[10]. This variant essentially trades more bank conflict stalls for less scheduling fails. Theoretically, the performance should not change since the number of total pipeline stalls is kept the same. However, the performance of IDCT with the variant is increased by 5.1% as compared with the original bank-conflict aware warp scheduling, simply due to the change of warp execution order[11].

Nonetheless, the results in Figure 10 show the strong potential of our elastic pipeline design to relieve the burden of avoiding shared memory bank conflicts from the programmer. Note also, static program analysis and optimizations are unable to avoid bank conflicts caused by irregular conflict patterns, which can be effectively handled by our proposal as demonstrated by the substantial performance improvement by elastic pipeline for the AES and Reduction kernels in Figure 9. Therefore, we can safely draw the conclusion that, our elastic pipeline proposal is capable of relieving the shared memory bank conflict issue for both regular and irregular access patterns, and thus enables more GPGPU applications to exploit the on-chip shared memory for improved performance and efficiency which is otherwise not possible without our proposal.

5.3 Performance of Non-Conflicting Kernels

Besides bank-conflicting kernels, we also would like to find out to which extent the proposed elastic pipeline will affect the execution of normal kernels without on-chip shared memory bank conflicts. Note, in this case, the bank-conflict aware warp scheduling behaves exactly the same as the default warp scheduling, since the conflict degree predicted by the bank conflict history cache is constantly zero (Figure 7).

The performance of the non-conflicting kernels (i.e. the original CUDA SDK source code) execution on both the baseline and elastic pipeline cores is shown in Figure 11. As we can see in the figure, the difference in performance is negligible. The performance difference between the baseline pipeline and the elastic pipeline for kernels without any bank conflict is due to: (1) the elastic pipeline can hide some of the writeback conflicts caused by the competition between core pipeline instructions and global memory loads (Figure 1); (2) the writeback MUX in the elastic pipeline (Figure 6) changes the default warp completion order of baseline in some cases (e.g., when the *MEM* and *NONMEM* paths compete for writeback, or, when there is a pipeline bypass in the *NONMEM* path (Figure 5)), which will further affect warp scheduling and execution order later. Factor (1) is always beneficial while factor (2) can contribute either positively or negatively to overall performance, depending on other subtle conditions (e.g. varied global memory ac-

[10]In the original bank-conflict aware warp scheduling, the ready warp pool is masked to empty in this case and pipeline will be stalled due to scheduling fails.

[11]We did not adopt this variant as it degrades the performance for other kernels.

cess efficiency, synchronization efficiency and control flow reconvergence efficiency, under different warp execution orders).

6. DISCUSSIONS

In this paper we assume *barrel processing* [22], which lays the basis for contemporary GPGPUs execution models [11]. In barrel processing, an instruction from a different hardware execution context is launched at each clock cycle in an *interleaved* manner. Consequently, there is no interlock or bypass associated with the barrel processing, thanks to the non-blocking feature of the execution model. Despite its advantages, *strict* interleaved multithreading has the drawback of requiring large on-chip execution contexts to hide latency, which can be improved in some ways. One such improvement is to allow multiple *independent* instructions to be issued into pipeline from the same execution context. In GPU cores, that is to allow multiple independent instructions from the same warp to be issued back-to-back (instead of the *strict* barrel execution model in which consecutively issued instructions are from different warps). Such execution is also adopted by some contemporary GPUs[24]. We call this extension "relaxed barrel execution model". The intention of the *relaxed barrel processing* in GPUs is to exploit ILP inside the thread, in order to reduce the minimal number of independent hardware execution contexts (active warps) required to hide pipeline latency.

In the case of relaxed barrel execution, there can be two choices to make our proposed Elastic Pipeline still work. First, we can still allow elasticity in the pipeline backend, which means that the consecutively issued instructions from the same warp commit out of the program order. This flexibility comes at the cost of the pipeline being unable to support precise exception handling. This can be resolved by adding a reorder buffer (ROB), however at extra hardware cost. A second choice is to forbid out-of-order writeback for instructions from the same warp. In order to make elastic pipeline still effective in reducing pipeline stalls, it is the responsibility of the scheduling logic not to execute any more instruction from the same warp, if current shared memory instruction will cause any bank conflict. This can be easily integrated into our bank-conflict aware warp scheduling technique.

Although only the effect of elastic pipeline for on-chip explicitly managed shared memory is evaluated in this paper, we believe the first level hardware cache can also benefit from our proposal. The reason is that the heavily-banked hardware cache also suffers from the dynamically varied cache access delay due to unbalanced bank accesses, which is similar to the shared memory case. We leave the evaluation of elastic pipeline for L1 cache as future work.

For out-of-order processors, the *pipeline elasticity* realized by our elastic pipeline proposal in this paper is actually enabled by the out-of-order engine. The OoO engine provides a small instruction window, which handles the variation of execution latency similar to a dataflow machine. The associated reorder buffer enforces the in-order instruction commitment. For architectures based on in-order pipelines, our elastic pipeline can be applied for a wide range of designs adopting barrel processing and SIMD data path, besides GPUs. The reason is that the on-chip bank conflict problem exists generally in such architectures. Furthermore, although we target the varied execution latencies caused by shared memory bank conflicts in this paper, elastic pipeline can also be applied to cope with on-chip execution latency variation due to other shared resource conflicts (e.g., accelerator (such as FPU) access, interconnect buffers allocation, miss status holding registers (MSHRs) allocation, etc.). In such cases, pipeline elasticity can be exploited to tolerate the resource conflicts and maximize the SIMD datapath throughput.

7. RELATED WORK

Bank conflict is an important problem in vector processors and it has been studied intensively in the literature. To cope with bank conflicts of vector access across stride families, several techniques have been proposed, including the use of buffers[7], dynamic memory schemes[8, 9, 14, 15], memory modules clustering[7], and intra-stream out-of-order access[23], just to name a few. Some of the existing techniques may be complementary to our elastic pipeline proposal, however subject to certain limitations. For example, one possible solution based on existing techniques is to add buffers in front of each shared memory bank to create a small access window. Subsequently, out-of-order scheduling techniques may be applied to resolve the bank conflicts, within such window. Similar techniques have been successfully used in other scenarios, such as the DRAM memory controller scheduling[21]. However, in the context of GPU fine-grain multithreaded SIMD processing, this technique is not applicable, because distributed our-of-order accesses in parallel shared memory banks create diverged execution orders for threads inside a warp/subwarp, effectively breaking the subwarp boundaries in the SIMD datapath. This often leads to conflicts in the register file banks at the writeback stage, which stall the pipeline in the end. In this case, the problem of shared memory bank conflicts is not resolved but just *postponed* to later pipeline stages.

Regarding bank conflicts in GPU shared memories, there have been some programming practices to avoid or relieve the conflict degree, by changing the data layout at source code level (e.g., zero-padding for data structures in shared memory)[16, 18]. While such optimizations are widely used in practice, they put nontrivial burden on programmers, since detailed knowledge of the shared memory hardware is needed, and sometimes major modifications to the source code are required. Besides, they also create portability issues when the code is deployed on platforms with different shared memory configurations. Moreover, static code optimizations are unable to relieve bank conflicts for conflict patterns which cannot be determined statically (such as the AES and Reduction kernels). Recently we see some work in automating such manual optimizations[10, 26]. Such high-level optimizations from the software side are complementary to our elastic pipeline proposal, albeit with the limitation mentioned above.

Current GPGPUs computing platforms suffer significantly from the relatively low bandwidth between the host CPU and the accelerator GPU attached to the CPU through system bus[16]. The research and development efforts can be classified into two categories: 1) to improve the efficiency of the CPU-GPU communication based on existing loosely-coupled system bus configuration[12]; and 2) to integrate the CPU and GPU into the same die[1, 3]. Our work addresses the problem of GPU on-chip shared memory bank conflicts. While our elastic pipeline design does not assume any particular CPU-GPU coupling configuration, the work on CPU-GPU communication optimization is orthogonal to our proposal.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we analyzed the shared memory bank conflict

problem, and identified how the bank conflicts are translated into pipeline performance degradation. Based on this observation, we proposed a novel elastic pipeline design that minimizes the negative impact of on-chip memory conflicts on system throughput, by decoupling bank conflicts from pipeline stalls. Simulation results show that our elastic pipeline with the co-designed bank-conflict aware warp scheduling significantly reduces the pipeline stalls by up to 64.0% and improves overall performance by up to 20.7%, with trivial hardware overhead.

In future work, we will evaluate the effect of our proposed elastic pipeline design on GPU cores with on-chip hardware caches. Besides, we also want to improve the accuracy of bank conflict degree prediction for irregular shared memory access patterns, which is desirable for improved warp scheduling efficiency. Since the core elastic pipeline design also interacts with the off-chip DRAM access, we would also like to investigate potential joint optimizations to improve the efficiency of both on-chip core pipeline and off-chip DRAM bandwidth, for memory intensive applications constrained by on-chip bank conflicts and off-chip bandwidth.

9. ACKNOWLEDGMENTS

This work was supported by the European Commission in the context of the SARC project (FP6 FET Contract #27648) and was continued by the ENCORE project (FP7 ICT4 Contract #249059).

References

[1] AMD FUSION, http://sites.amd.com/us/fusion/apu/pages/fusion.aspx.

[2] GPGPU home page. http://gpgpu.org.

[3] Intel Sandy Bridge, Intel processor roadmap, 2010.

[4] OpenCL home page. http://www.khronos.org/opencl/.

[5] Bakhoda, A., Yuan, G., Fung, W., Wong, H., and Aamodt, T. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (2009), pp. 163 –174.

[6] Bhatnagar, H. *Advanced ASIC Chip Synthesis Using Synopsys Design Compiler Physical Compiler and PrimeTime*, 2nd ed. Kluwer Academic Publishers, 2001.

[7] D. T. Harper III. Block, multistride vector and FFT accesses in parallel memory systems. *IEEE Trans. Parallel and Distributed Systems 2*, 1 (1991), 43–51.

[8] D. T. Harper III. Increased memory performance during vector accesses through the use of linear address transformations. *IEEE Trans. Computers 41*, 2 (1992), 227–230.

[9] D. T. Harper III, and Linebarger, D. A. Conflict-free vector access using a dynamic storage scheme. *IEEE Trans. Computers 40*, 3 (1991), 276–283.

[10] Diamos, G. F., Kerr, A. R., Yalamanchili, S., and Clark, N. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 353–364.

[11] Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2007), MICRO 40, IEEE Computer Society, pp. 407–420.

[12] Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N., and Hwu, W.-m. W. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (New York, NY, USA, 2010), ASPLOS '10, ACM, pp. 347–358.

[13] Glaskowsky, P. N. Nvidia's Fermi: The first complete GPU computing architecture.

[14] Gou, C., Kuzmanov, G., and Gaydadjiev, G. N. SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In *Proceedings of the 24th ACM International Conference on Supercomputing* (New York, NY, USA, 2010), ICS '10, ACM, pp. 179–188.

[15] Gou, C., Kuzmanov, G. K., and Gaydadjiev, G. N. SAMS: single-affiliation multiple-stride parallel memory scheme. In *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?* (New York, NY, USA, 2008), MAW '08, ACM, pp. 350–368.

[16] Kirk, D. B., and mei W. Hwu, W. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[17] Manavski, S. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on* (2007), pp. 65 –68.

[18] NVIDIA. CUDA best practice guide, edition 3.0.

[19] NVIDIA. CUDA SDK, version 2.2.

[20] NVIDIA. The CUDA compiler driver NVCC, edition 2.2.

[21] Rixner, S., Dally, W., Kapasi, U., Mattson, P., and Owens, J. Memory access scheduling. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on* (2000).

[22] Thistle, M. R., and Smith, B. J. A processor architecture for horizon. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 1988), Supercomputing '88, IEEE Computer Society Press, pp. 35–41.

[23] Valero, M., Lang, T., Peiron, M., and Ayguade, E. Conflict-free access for streams in multimodule memories. *IEEE Trans. Computers 44* (1995), 634–646.

[24] Volkov, V. Better performance at lower occupancy. In *GPU Technology Conference 2010* (2010), GTC '10.

[25] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on* (2010), pp. 235 –246.

[26] Yang, Y., Xiang, P., Kong, J., and Zhou, H. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 86–97.