
CHALMERS
UNIVERSITY OF TECHNOLOGY

# Addressing GPU on-chip shared memory bank conflicts using elastic pipeline

Int J Parallel Prog (2013) 41:400–429
DOI 10.1007/s10766-012-0201-1

# Addressing GPU On-Chip Shared Memory Bank Conflicts Using Elastic Pipeline

**Chunyang Gou · Georgi N. Gaydadjiev**

Received: 17 August 2011 / Accepted: 14 June 2012 / Published online: 3 July 2012


**Abstract** One of the major problems with the GPU on-chip shared memory is bank conflicts. We analyze 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 causes 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 representative benchmarks, at trivial hardware overhead.

**Keywords** GPU · On-chip shared memory · Bank conflicts · Elastic pipeline

This is an extended version of the paper originally presented in Computing Frontiers 2011 [1].

C. Gou (✉) · G. N. Gaydadjiev
Computer Engineering Laboratory, Faculty of Electrical Engineering,
Mathematics and Computer Science TU Delft, Delft, The Netherlands
e-mail: c.gou@tudelft.nl

G. N. Gaydadjiev
e-mail: g.n.gaydadjiev@tudelft.nl

## 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 numbers of 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 (BW) 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 memories and the hardware caches, are heavily banked, in order to provide high bandwidth for the parallel SIMD lanes. Even with adequate bandwidth provided by the parallel banks, applications can still suffer drastic pipeline stalls, resulting in significant performance losses 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 degree is constrained due to such overhead.

In this paper, we determine 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 propose novel *Elastic Pipeline* design that minimizes the negative impact of on-chip memory bank conflicts on system throughput. More precisely, this paper makes the following contributions:

- careful analysis of the impact of GPU on-chip shared memory bank conflicts on pipeline performance degradation;
- a novel *Elastic Pipeline* design to alleviate on-chip shared memory conflicts and boost overall system throughput;
- co-designed *bank-conflict aware warp scheduling* technique to assist our Elastic Pipeline hardware;
- pipeline stalls reductions of up to 64.0 % leading to overall system performance improvement of up to 20.7 % under realistic scenario.

The remainder of the paper is organized as follows. In Sect. 2, we provide the background and motivation for this work. In Sect. 3, we analyze the GPU shared memory bank conflict problem from latency and bandwidth perspective, and identify the mechanism through which shared memory bank conflicts degrade GPU pipeline

performance. Based on the findings, we discuss our proposed Elastic Pipeline design in Sect. 4. The co-designed bank-conflict aware warp scheduling technique is elaborated in Sect. 5. Performance results of our proposed Elastic Pipeline for GPGPU applications are evaluated in Sect. 6, followed by some general discussions of our simulated GPU core architecture along with the Elastic Pipeline in Sect. 7. The major differences between our proposal and related art are summarized in Sect. 8. Finally, Sect. 9 concludes the paper.

## 2 Background and Motivation

GPU is a manycore platform employing a large number of lightweight cores to achieve high aggregated performance, originally targeting graphics workloads. Nowadays, its utilization has spanned far beyond graphics rendering, covering a wide spectrum of general purpose applications (referred to as GPGPU [2]). GPGPUs often adopt *bulk synchronous programming* (BSP) [3] programming models. BSP programs on GPUs often employ *barrel processing* [4] due to its low pipeline implementation overhead.

### 2.1 Programming Model Properties

The BSP model has been widely adopted in programming languages and language extensions targeting manycore accelerator architectures, e.g., CUDA[5], OpenCL[6] and others [7]. In such languages, the parallelism of the application's compute intensive kernels is explicitly expressed in a *single program multiple data* (SPMD) manner. In such *explicitly-parallel, bulk-synchronous* programming models, the programmer extracts the data-parallel sections of the 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 SPMD *kernel*. Each kernel instance (called *threads* in CUDA) normally operates on one single unit or a relatively small group of units. Multiple kernel instances run independently on the execution cores. During SPMD program execution, all threads execute in parallel, and perform barrier synchronizations only when the results from threads need to be exchanged. Therefore, fast local, shared memories to facilitate communicating the results among execution cores is critical to guarantee high performance [3]. An important example is the GPU shared memories, which are the main embodiment of the local memory in the BSP model, when running GPGPU kernels.

In CUDA, the parallel threads are organized into a two-level hierarchy, in which a kernel (also called *grid*) consists of parallel CTAs (*cooperating thread array*, aka *thread block*), with each CTA composed of parallel threads, as shown in Fig. 1a. Explicit, localized synchronization and on-chip data sharing mechanisms (such as CUDA shared memory) are supported for threads belonging to the same CTA.

### 2.2 Baseline Manycore Barrel Processing Architecture

Figure 1 illustrates our baseline architecture. On the right the high-level system organization is shown. The system consists of a GPU node with $K$ cores and a memory

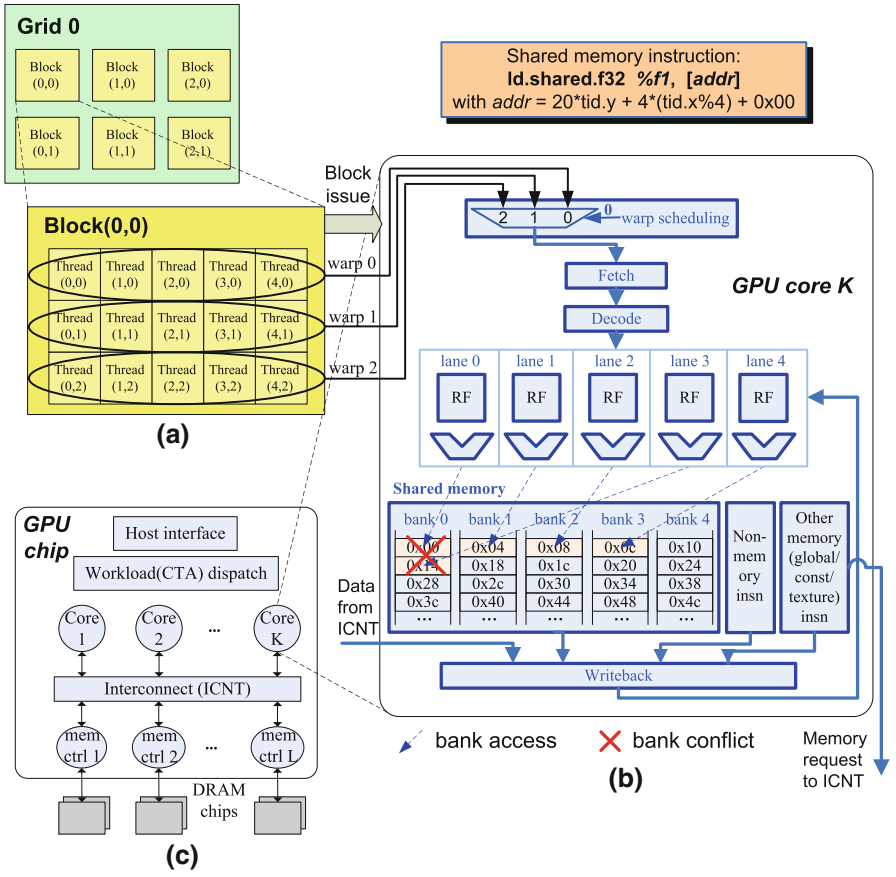


**Fig. 1** **a** CUDA threads hierarchy; **b** thread execution in GPU core pipeline; **c** GPU chip organization

subsystem with $L$ DRAM channels, connected by the on-chip interconnect. Depending on the implementation, the GPU node itself may form a chip (similar to discrete GPUs), or the GPU node(s) and host CPU(s) can be integrated on the same die (e.g., [8]). The host processor offloads compute intensive kernels to the GPU node during execution. The kernel code and parameters are transferred from host processor using the host interface, and the workloads are dispatched at the CTA/block granularity. Concurrent CTAs are executed on GPU cores independent of each other.

A microarchitectural illustration of the GPU pipeline is shown in Fig. 1b. During execution, a batch of threads from the same CTA are grouped into a *warp*, the smallest unit for the pipeline front-end processing. Each core maintains a set of on-chip hardware execution contexts and switches at the warp granularity. The context switching, also called *warp scheduling*, is done in an interleaved manner, also known as *barrel processing* [4]. Warps are executed by the core pipelines in a SIMD fashion for improved pipeline front-end processing efficiency. In practical designs, the threads number in a warp can be a multiple of the SIMD width. In this case, a warp is composed

of multiple slices (called *subwarps* here), with each subwarp size equal to the SIMD width. Subwarps are processed by the SIMD pipeline back-to-back.

### 2.3 Shared Memory Access on GPGPU

GPUs rely mainly on massive hardware multithreading to hide off-chip memory latencies. In addition, on-chip memory hierarchies are also deployed in GPUs in order to provide high bandwidth and low latency, particularly for data sharing among SPMD threads employing the BSP model (as discussed in Sect. 2.1). Such on-chip memories include, software managed caches (shared memory), or hardware caches, or a combination of both [9]. 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 Fig. 1b, data needed by both lanes 0 and 4 reside in the same shared memory bank 0. In this case a *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. Another motivating example is provided in [1], the original version of this paper.

## 3 Problem Analysis

In this section, we will first analyze the latency and bandwidth implications of GPU shared memory bank conflicts, then identify and analyze the mechanism how shared memory bank conflicts degrade GPU pipeline performance.

### 3.1 Latency and Bandwidth Implications

GPUs use a large number of hardware threads to hide both function unit and memory access latency. Such extreme multithreading requires a large amount of parallelism. The needed parallelism, using "Little's law" [10], can be calculated as follows:

$$Needed\_parallelism = Latency \times Throughput \tag{1}$$

For GPU throughput cores, this means the required number of in-flight operations to maintain peak throughput equals the product of pipeline latency and SIMD width. For the *strict* barrel processing (Sect. 4.2) where all in-flight operations are from different hardware thread contexts, this directly determines the required amount of concurrent threads. For other cases in general, the $needed\_parallelism$ is proportional to the concurrent threads number. As a result, a moderate increase in pipeline latency can be effectively *hidden* by running more threads. For example, the GPU core configuration used in our evaluation employs a 24-stage pipeline with SIMD width of 8

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

(This is in line with a contemporary NVIDIA GTX280 architecture [11]. Similar pipeline configurations are also widely used in research GPU models [12,13].). Hence, assuming 4 extra stages for tolerating shared memory bank conflicts, the pipeline depth is increased from 24 stages to 28. In this case, the number of threads to hide the pipeline latency (function unit/shared memory) is penalized, by an increment from 192 to 224, according to Eq. 1. This is normally not a problem, for both the GPU cores (where there are adequate hardware thread contexts), and the application domains targeted by GPUs (with enough parallelism available).

It is also worth noting that, unlike CPUs, program control flow speculation is not needed in GPUs thanks to the barrel processing model [4]. Therefore, the increase in pipeline latency will not incur pipeline inefficiency associated to deeper pipelines [14].

On the other hand, the peak bandwidth of the shared memory is designed to feed the SIMD core, as illustrated by the pipeline model shown in Fig. 1b, where the number of shared memory banks equals $simd\_width$.[2] Therefore, bandwidth is naturally not a problem when GPU shared memory works at peak bandwidth. However, intuitively, when the shared memory bank conflicts are severe, the *sustained* bandwidth can drop dramatically. Moreover, it may eventually become the bottleneck of the entire kernel execution. In such cases no microarchitectural solution exists without increasing the shared memory raw bandwidth.

To facilitate our discussion, we use the following definition of *bank conflict degree* of SIMD shared memory access:

**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 Fig. 1b has a conflict_degree of 2. In general, it takes $\left\lceil \frac{conflict\_degree}{nr\_of\_shared\_memory\_ports} \right\rceil$ cycles to read/write all data for a SIMD shared memory access.

Naturally, it depends on an application's shared memory *access intensity* and *conflict degree* whether or not it is shared memory bandwidth bound. Assume the available shared memory bandwidth, $BW_{avail}$, allows access of one data element per core-cycle for all SIMD lanes. In this case, a single shared memory instruction's bandwidth requirement equals its $conflict\_degree$. This is due to the fact that this instruction occupies the shared memory for *conflict_degree* cycles during the execution. Suppose the ratio between number of executed shared memory access instructions and all instructions is $r$. Then the shared memory bandwidth can become a bottleneck *if and only if* the available bandwidth is smaller than required, for the entire GPU kernel execution:

$$BW_{avail} < BW_{req}$$

[2] Note, in practical implementations, the number of shared memory banks can be a multiple of (e.g., $2X$) the SIMD width, all running at a lower clock frequency compared to the core pipeline. The bottom line is, the peak bandwidth of the shared memory is at least be capable of feeding the SIMD core [11].

**Table 1** Benchmark shared memory bandwidth requirement

| Benchmark name | $r(\%)$ | $conflict\_degree_{avg}$ | $r \times conflict\_degree_{avg}$ |
|---|---|---|---|
| AES_encypt | 36.1 | 1.54 | 0.56 |
| AES_decypt | 35.9 | 1.53 | 0.55 |
| Reduction | 4.0 | 3.07 | 0.12 |
| Transpose | 3.7 | 4.50 | 0.17 |
| DCT | 21.6 | 3.33 | 0.72 |
| IDCT | 21.2 | 3.33 | 0.71 |
| DCT_short | 10.3 | 2.75 | 0.28 |
| IDCT_short | 10.3 | 2.75 | 0.28 |

i.e.,

$$1 < r \times IPC_{nor} \times conflict\_degree_{avg}$$

Considering the normalized IPC per SIMD lane, $IPC_{nor}$, is no larger than 1, we see that the shared memory bandwidth is a bottleneck *iff*

$$r \times conflict\_degree_{avg} > 1 \tag{2}$$

Table 1 shows the values of $r$ (denoted as "shared memory intensity" in Table 5) and $conflict\_degree_{avg}$ ("average bank conflict degree" in Table 5) in real kernel execution. As can be observed in Table 1, Eq. 2 holds for none of the GPU kernels in our experiments. This indicates we have large shared memory *bandwidth margin*, as far as bank conflicts are concerned. In other words, we have sufficient shared memory bandwidth to sustain peak IPC, even if bank conflicts.

This insight is very important, since it reveals the opportunity to improve overall performance, without increasing the shared memory raw bandwidth. In the rest of the paper, we will see how moderate microarchitectural refinement can be created to solve the problem.

### 3.2 Pipeline Performance Degradation Due to Bank Conflicts

The baseline in-order, single-issue GPU core pipeline configuration is illustrated on the top of Fig. 2a. The warp scheduling stage is not shown, and only one of the parallel SIMD lanes of the execution/memory stages is depicted in Fig. 2a for simple illustration.[3] 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. $t_i$ denotes execution time in cycle $i$, and $W_i$ denotes warp instruction fetched in cycle $i$.

[3] Please refer to Fig. 1b for the pipeline details.

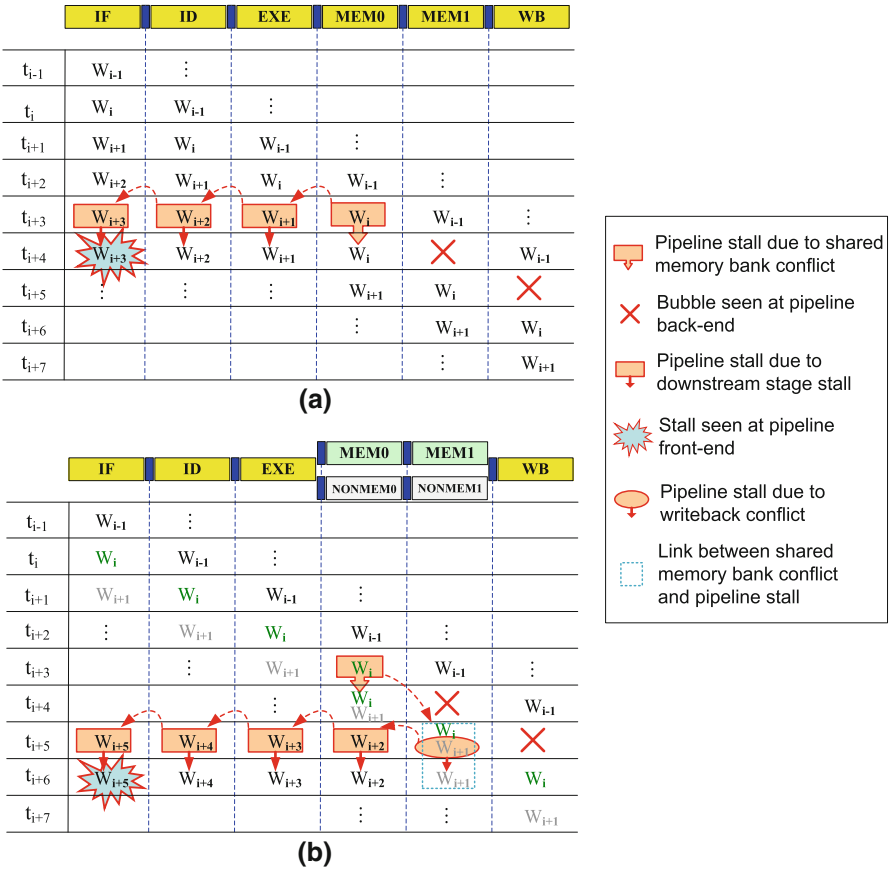


**Fig. 2** Baseline in-order pipeline: **a** unified memory stages and **b** split memory stages

As Fig. 2a 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 incurring 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 writeback bubble in cycle $i+5$.

Notice the fact that $W_{i+1}$ execution does not have 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) is possible, as Fig. 2b 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 Fig. 2b 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 Fig. 2b.

**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 blocking memory bank conflicts. The variation in execution latency incurs writeback bubbles and writeback conflicts, which further causes pipeline stalls in the in-order pipeline. As a result of the above the system throughput is decreased.

## 4 Elastic Pipeline Design

To address the conclusions of the analysis of GPU shared memory bank conflicts, we will introduce a novel Elastic Pipeline design in this section. Its implementation will be presented, with emphasis on the conflict tolerance, hardware overhead and pipeline timing impact.

The Elastic Pipeline design is able to eliminate the negative impact of shared memory bank conflicts on system throughput, as shown in Fig. 3. Compared to the baseline pipeline with split memory stages in Fig. 2b, 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 Fig. 3 shows, by forwarding matured instructions in the *NONMEM* queue to the writeback stage, the writeback conflict is removed, and thus the link between bank- and writeback conflicts is cut and the associated pipeline stall is eliminated.

### 4.1 Safe Scheduling Distance and Conflict Tolerance

For ease of discussion, we first define the following warp types:
**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.

In Fig. 3 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

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

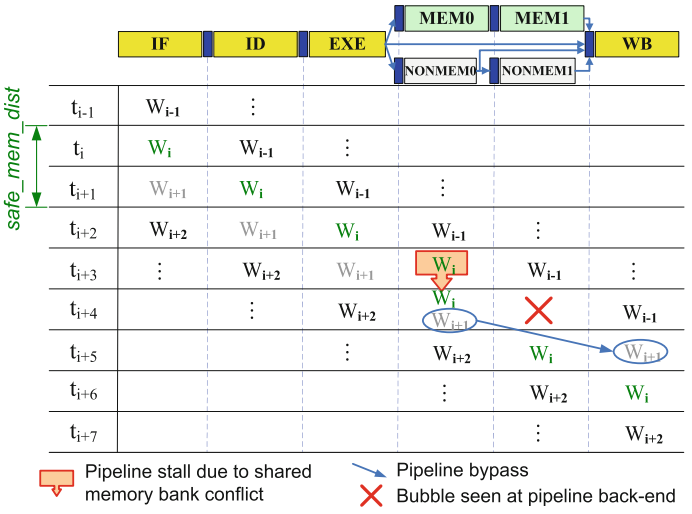


**Fig. 3** Elastic pipeline

cycle, due to its shared memory bank conflict. Such a problem exists even if $W_{i+1}$ is not a shared memory access.[5] To avoid this, 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:

$$safe\_mem\_dist = \left\lceil \frac{conflict\_degree}{nr\_of\_shared\_memory\_ports} \right\rceil \tag{3}$$

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 Fig. 3 is $\lceil \frac{2}{1} \rceil = 2$, which means that in the next cycle, only non-memory warps 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 long as the $safe\_mem\_dist$ constraint is satisfied. We will discuss this in more detail in Sect. 5.

[5] Note, in such case, even if there exists a third path (with fixed number of stages) for that memory access type, writeback bubbles cannot be avoided, due to the same phenomenon illustrated in Fig. 2b.

### 4.2 Out-of-Order Instruction Commitment

In Fig. 3, the Elastic Pipeline shows the behavior of out-of-order instruction commitment. This has important impact on the system design, as elaborated in the following.

As examined in Sect. 2.2, *barrel processing* [4] lays the basis for contemporary GPGPUs execution models [15]. 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 the 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 [16]. 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.

For GPU cores with *strict barrel processing*, 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 the same warp (but without data dependence), out-of-order instruction commitment within the same execution context may occur. This breaks the sequential semantics of the code and is penalized by the pipeline being unable to support precise exception. In such case, 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 re-order 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.

### 4.3 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 (as discussed in Sect. 2.2). 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.

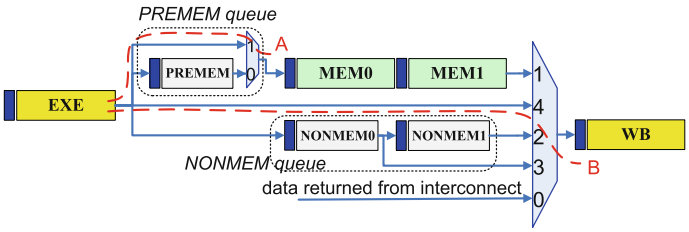


**Fig. 4** Elastic pipeline logic diagram

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 Fig. 3 will have to execute the same shared 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.

To solve this problem, an extension has to be adopted in the Elastic Pipeline shown at the top of Fig. 3. With this 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 Fig. 4). 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 Fig. 4, 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). For example, the data returned from interconnect is assigned with the highest priority (it is loaded from the external main memory after hundreds of cycles delay); whereas only when there is no available data from elsewhere can the data output from stage *EXE* be written back (if it is not a memory access).

With the Elastic Pipeline configuration of Fig. 4, $W_{i+1}$ in Fig. 3 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

$$M = nr\_of\_MEM\_stages \tag{4}$$

$$N = \left\lceil \frac{warp\_size}{simd\_width} \right\rceil - 1 \tag{5}$$

**Table 2** Elastic pipeline hardware 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 |

### 4.4 Hardware Overhead and Impact on Pipeline Timing

The additional hardware overhead as compared with the baseline pipeline is summarized in Table 2. 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 2, the area consumption of the additional pipeline latches is in the order of $(M + N) \cdot simd\_width$. Considering small $M$s and $N$s 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 Fig. 4, 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 Fig. 4.[6] There are two separate paths in which the *EXE* stage is prolonged: path A and B, as marked by the two dash lines in Fig. 4. A is the (N+1)-to-1 multiplexer and B is the (M+3)-to-1 multiplexer listed in Table 2. With standard critical path optimizations such as the *priority on late arriving signal* technique [17], 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).

## 5 Bank-Conflict Aware Warp Scheduling

As discussed in Sect. 4.1, 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* (Eq. 7) cycles after a bank-conflicting shared memory warp is scheduled for execution. This is called "*bank-conflict aware warp scheduling*", which will be discussed next.

[6] Note, although not shown in Fig. 2a, 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.

### 5.1 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 not known at compile time, or irregular accesses whose bank conflict patterns can not be identified statically (such as the AES example). Second, there is no additional hardware cost directly by the dynamic detection itself, as the shared memory bank conflict detection logic is also needed in the baseline pipeline.[7]

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:

$$warp\_bkconf\_degree = \sum_{i=0}^{\left\lceil \frac{warp\_size}{simd\_width} \right\rceil - 1} conflict\_degree_i \tag{6}$$

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 $\left\lceil \frac{warp\_size}{simd\_width} \right\rceil$ cycles (if there is no pipeline stall during that time).

Accordingly, the safe memory warp schedule distance in Eq. (3) is extended in the following:

$$warp\_safe\_mem\_dist = \left\lceil \frac{\sum_{i=0}^{\left\lceil \frac{warp\_size}{simd\_width} \right\rceil - 1} conflict\_degree_i}{nr\_of\_shared\_memory\_ports} \right\rceil \tag{7}$$

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 last 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

[7] The shared memory has to identify the conflict degree of each SIMD shared memory access (i.e., $conflict\_degree_i$ in Eq. 6) in order to resolve it.

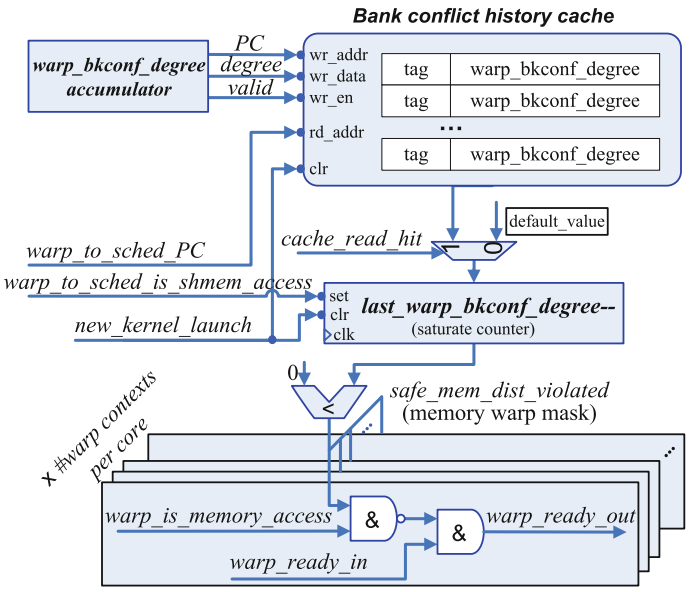


**Fig. 5** Bank-conflict aware warp ready signal generation

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 *last execution* of the same instruction.

## 5.2 Bank Conflict History Cache

In order to maintain the historic conflict degree information, we implement a small private *bank conflict history cache* distributed among the GPU cores, as shown in Fig. 5. 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.[8] Whenever a shared memory warp is scheduled, the *last_warp_bkconf_degree* counter is set to its last warp bank conflict degree in history, by checking out 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 is violated.

Note, in our design we assume in the warp scheduling stage it is known whether or not a ready warp is a shared memory access (the *"warp_to_sched_is_ shmem_access"*

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

signal in Fig. 5), or a memory access (the *"warp_is_memory_ access"* signal). This can be easily achieved with negligible hardware overhead. For example, a bit-vector can be employed to maintain the instruction type of all (static) instructions in a kernel. This type vector can be initialized at kernel launch time, and checked out during warp execution.

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 highly irregular shared memory access patterns and dynamically changing conflict degree (e.g., AES).

Note, 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, immediately after the incorrect prediction, 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 Sect. 6.1.

### 5.3 Proposed Warp Scheduling Scheme

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 Fig. 5 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 Sect. 4. 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 Sect. 6.2. It should be noted that, warp scheduling by itself is unable to reduce pipeline stalls caused by shared memory bank conflicts, without the Elastic Pipeline infrastructure.

### 5.4 Hardware Overhead

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 aware of in-flight bank-conflicting shared memory accesses and enforce the constraint on following warps to be scheduled, as shown in Fig. 5. Table 3

**Table 3** Hardware overhead of bank conflict prediction and warp mask generation (per GPU core)

| Type | Logic complexity | Quantity |
|---|---|---|
| Bank conflict history cache | $nr\_of\_cache\_lines \cdot (log_2(warp\_size) + 14\text{-}log_2(cache\_sets))$ bits, dual port (1R + 1W) | 1 |
| AND/NAND gate | – | 2×nr of 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 |

summarizes the hardware overhead incurred by the bank conflict degree prediction and bank-conflict aware warp ready signal generation. The main contributor in Table 3 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 Eq. (6) takes $log_2(simd\_width)$ bits (Sect. 3.2), 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 3.

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 Fig. 5.

## 6 Experimental Evaluation

**Experimental Setup:** We use a modified version of GPGPU-Sim [12], which is a cycle accurate full system simulator for GPUs implementing ptx ISA [19]. We model GPU cores with a 24-stage pipeline similar to contemporary implementations [11,13]. The detailed configuration of the GPU processor is shown in Table 4. 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 4 shows. The number of memory pipeline stages and the *warp_size/simd_width* ratio is 4 (see Table 4). Therefore the queue depth is set to 4 for the *NONMEM* queue, and 3 for the *PREMEM* queue in the Elastic Pipeline, according to Eqs. (4) and (5).

We select 8 shared memory intensive benchmarks from CUDA SDK [20] and other public sources [21]. Table 5 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 (i.e., SIMD lanes) of 16 GPU cores, and the second number

**Table 4** 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 (Sect. 4.2) 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 | 16 KB 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, 2 KB page per DRAM chip, 8 banks, 8 Bytes/channel/transmission (51.2 GB/s BW in total), 800 MHz bus freq, 32 DRAM request buffer entries memory controller policy: out-of-order (FR-FCFS) [18] |
| Interconnect | Crossbar, 32-Byte flit size |

**Table 5** Benchmark characteristics

| Name | Source | Grid Dim | CTA Dim | CTAs/core | Total Insns |
|---|---|---|---|---|---|
| AES_encypt | [21] | (257,1,1) | (256,1,1) | 2 | 35132928/534 |
| AES_decypt | [21] | (257,1,1) | (256,1,1) | 2 | 35527680/540 |
| Reduction | CUDA SDK | (16384,1,1) | (256,1,1) | 4 | 415170560/49 |
| Transpose✳ | CUDA SDK | (16,16,1) | (16,16,1) | 4 | 3538944/54 |
| DCT✳ | CUDA SDK | (16,32,1) | (8,4,2) | 7 | 7274496/222 |
| IDCT✳ | CUDA SDK | (16,32,1) | (8,4,2) | 7 | 7405568/226 |
| DCT_short✳ | CUDA SDK | (16,16,1) | (8,4,4) | 7 | 10223616/337 |
| IDCT_short✳ | CUDA SDK | (16,16,1) | (8,4,4) | 7 | 10190848/336 |

| Name | Sh-mem Insns | Sh-mem Intensity (%) | Avg. conf. degree | Theoretic speedup | Irregular Sh-mem pattern |
|---|---|---|---|---|---|
| AES_encypt | 12697856/193 | 36.1 | 1.54 | 1.19 | Y |
| AES_decypt | 12763648/194 | 35.9 | 1.53 | 1.19 | Y |
| Reduction | 16744448/5 | 4.0 | 3.07 | 1.09 | Y |
| Transpose✳ | 131072/2 | 3.7 | 4.50 | 1.13 | N |
| DCT✳ | 1572864/48 | 21.6 | 3.33 | 1.50 | N |
| IDCT✳ | 1572864/48 | 21.2 | 3.33 | 1.50 | N |
| DCT_short✳ | 1048576/40 | 10.3 | 2.75 | 1.18 | N |
| IDCT_short✳ | 1048576/40 | 10.3 | 2.75 | 1.18 | N |

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. Letter **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 kernels marked 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 the Elastic Pipeline performance in Sects. 6.1 and 6.2. 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 codes 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.

We use the NVCC toolchain [19] to compile the CUDA application code. The toolchain first invokes *cudafe* to extract and separate the host C/C++ code and device C code from the CUDA source, then it invokes two stand-alone compilers: *gcc* to compile the host C/C++ code running on the CPU and *nvopencc* to compile the device code running on the GPU. All benchmarks are compiled with *-O3* option.

It has to be pointed out that, although we use CUDA code and the corresponding toolchain in our experiments, our proposed Elastic Pipeline and bank-conflict aware warp scheduling do not rely on any particular GPGPU programming model. Further, the application of our proposal is not limited to GPGPU applications—graphics kernels can also benefit from it where on-chip shared memory bank conflict is a concern.

### 6.1 Effect on Pipeline Stall Reduction

Figure 6 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 is normalized to the baseline GPU. Inside each bar, 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). Note, GPU core warp scheduling fails if the ready warp pool is empty, which can be incurred by: (1) the core pipeline latency or other long latencies (e.g., due to main memory access) which are not hidden by the parallel warp execution (i.e., not enough concurrent CTAs active on chip); (2) the barrier synchronization; (3) warp control-flow re-convergence mechanisms [15].

As discussed in Sect. 3, shared memory bank conflicts create writeback bubbles which finally incur pipeline stalls. Note, although the portion of shared memory

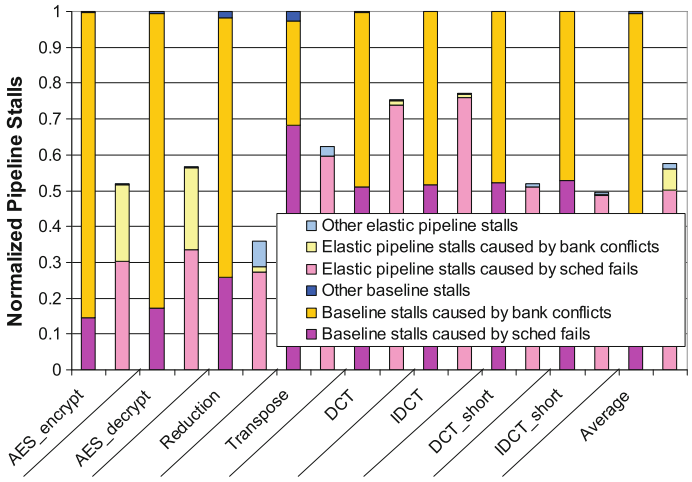


**Fig. 6** Pipeline stall reduction. In each group, *left bar* baseline GPU. *right bar* Elastic pipeline enhanced GPU

instructions is small for some kernels (such as Reduction and Transpose, with less than 5 % as shown in Table 5), some of the involved SIMD shared memory accesses result in very high bank conflict degrees (up to 8). Therefore the total number of bank conflicts and pipeline stalls is quite significant in the baseline, as shown in Fig. 6.

As Fig. 6 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 (Sect. 5.2). 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. Figure 6 also confirms that the impact of compulsory misses in bank conflict history cache is negligible in the Elastic Pipeline.

On the other hand, the number of pipeline stalls due to warp scheduling failures is 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.

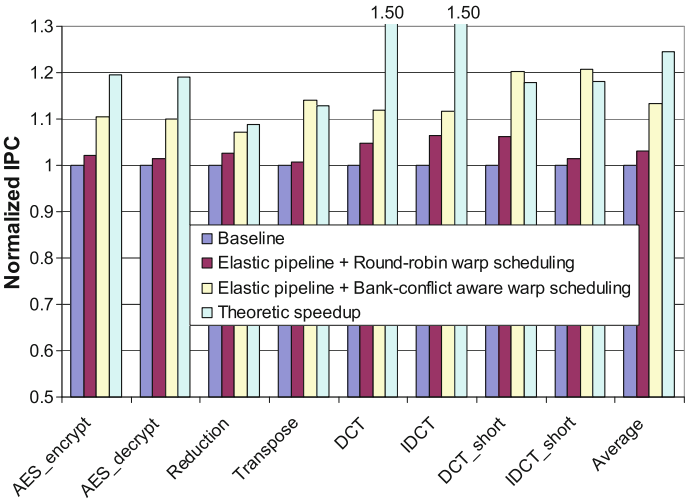


**Fig. 7** Performance improvement

The last type of pipeline stalls (*other pipeline stalls* in Fig. 6) is caused by writeback conflicts incurred by data returned from interconnect. The number is slightly increased in the Elastic Pipeline as shown in Fig. 6. 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.

## 6.2 Performance Improvements

Figure 7 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 theoretical 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 13.3 %, as compared to the baseline. This confirms our analysis in Sect. 5. 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 remains in the Elastic Pipeline, as shown in Fig. 6. DCT and IDCT see a *huge* gap between the actually achieved performance gain by Elastic Pipeline and the theoretical bound. This is due to the number of pipeline stalls caused by warp scheduling fails is significantly increased, also shown in Fig. 6. It is interesting to see that the speedup of our Elastic Pipeline design exceeds the theoretical 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 Sect. 6.1.

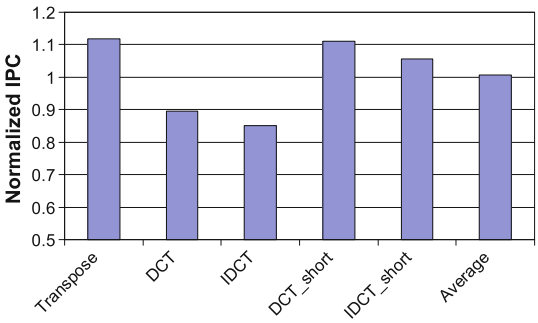


**Fig. 8** Performance of un-optimized code execution on Elastic pipeline (IPC normalized to hand-optimized code execution on baseline)

In order to find out how our Elastic Pipeline performs in relieving the overhead of reducing shared memory bank conflicts from the 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 Fig. 8. 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 (the average normalized IPC is close to 1). However, we also found that for the DCT/IDCT kernels, the performance gap is quite large (e.g., 14.9 % less performance with the Elastic Pipeline/un-optimized kernel combination for IDCT, as compared to the baseline execution of hand-optimized code).

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 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 are no ready warps to execute non-memory instruction.[9] This variant essentially trades more bank conflict stalls for fewer 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.[10] This could be further improved by taking into account the main memory bandwidth efficiency in our design. For example, it may be possible to create more efficient warp scheduling schemes which are aware of not only on-chip shared memory bank-conflict, but also global memory efficiency. More details about such interactions between on/off-chip memory accesses at system level are discussed in Sect. 6.4.

Nonetheless, the results in Fig. 8 suggest 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

[9] 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.

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

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 Fig. 7. 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.

### 6.3 Performance of Non-Conflicting Kernels

Besides benefiting bank-conflicting kernels, our Elastic Pipeline is also expected (at least) not to degrade the performance of normal kernels without on-chip shared memory bank conflicts. 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 (Fig. 5). The *non-conflicting* kernels examined in this section are the five kernels marked by a ⁎ in Table 5, with the hand-optimization to avoid shared memory bank conflict (i.e., the original CUDA SDK code is used).

The performance of the non-conflicting kernels execution on Elastic Pipeline is shown in Fig. 9 (with IPC normalized to the baseline execution). As we can see in the figure, the performance difference exists but is rather small in general (all normalized IPC values are close to 1). 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 (e.g., non-global memory instructions) and global memory loads (Fig. 1); (2) the writeback MUX in the Elastic Pipeline (Fig. 4) 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 Fig. 3), 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 access efficiency, synchronization efficiency and control flow re-convergence efficiency, under different warp execution orders).

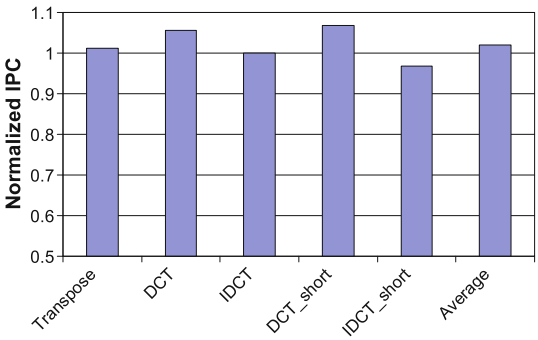


**Fig. 9** Elastic pipeline performance for non-conflicting kernels (IPC normalized to baseline execution)

### 6.4 Interaction with Off-chip DRAM Access

At first glance, it seems that on-chip shared memory access is decoupled from off-chip DRAM access. Counterintuitively, however, we have already observed quite some inter-operation between them, is the warp execution order (and the subsequent DRAM access patterns): as discussed in Sects. 6.1 and 6.2. Indeed, it would be interesting to inspect the relationship between our proposed Elastic Pipeline and the kernel DRAM access behavior. Figure 10 tries to unveil it in a quantitatively way. The graph at the top shows the ratio between theoretical speedup and the speedup actually gained by our Elastic Pipeline design (data from Fig. 7). And the one at the bottom shows the required DRAM bandwidth by each kernel, normalized to GPU DRAM peak bandwidth (Table 4). The required bandwidth is calculated by dividing the total amount of global memory data access by the execution time assuming IPC = 1 for each SIMD lane. We choose the *required* bandwidth as the metric instead of the *actual* bandwidth utilization, since the latter has already been coupled with the interaction between the core pipeline behavior and external DRAM access patterns.

Interestingly, the two graphs in Fig. 10 show quite strong correlation between each other. Roughly speaking, the higher off-chip bandwidth is required, the larger the gap between the speedup of our Elastic Pipeline and the theoretical bound—in other words, the more difficult to reclaim the performance loss due to bank-conflict pipeline stalls. For the benchmarks examined in our experiments and the off-chip DRAM configuration in our GPU processor, we can see that some rough threshold, say, the 50 % DRAM peak bandwidth bar, separates the benchmarks into *high bandwidth* group (DCT/IDCT) and *low bandwidth* group (the other kernels), as illustrated in Fig. 10. For the low bandwidth group, the performance loss due to bank conflict stalls is relatively easy to be reclaimed by our proposed Elastic Pipeline design (indeed the theoretical bound is even surpassed in cases of Transpose and DCT/IDCT_short). While for the high bandwidth group, the Elastic Pipeline performance gain is far from ideal. The reason seems to be that, standalone core pipeline techniques (such as our Elastic Pipeline proposal) oblivious to the main memory access efficiency are unable to exploit the full potential of the hardware and the parallelism inherent in the software. This

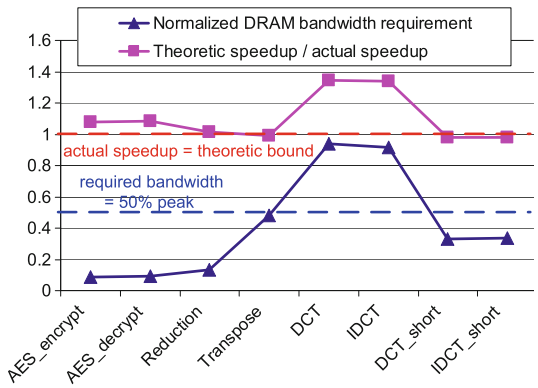


**Fig. 10** DRAM bandwidth impact

also explains the relatively large performance loss for DCT/IDCT kernels in Fig. 8 with our Elastic Pipeline.

To summarize, it can be anticipated that, cooperative optimization schemes which take care of both core pipeline optimization and off-chip DRAM bandwidth efficiency are highly desirable, to further improve the overall performance for kernels with both heavy on-chip shared memory bank conflicts and off-chip bandwidth requirement.

## 7 Discussions

Typical throughput-oriented architectures (such as GPUs) exploit massive thread- and/or data-level parallelism to hide long latencies and achieve high throughput using in-order pipelines. Introducing out-of-order execution capability to such architectures is generally not preferable due to its limited performance gain at significant hardware cost. This paper proposed Elastic Pipeline (EP) design to avoid pipeline stalls due to GPU shared memory bank conflicts. Although EP also employs an out-of-order (OoO) pipeline, it deviates from OoO schemes in typical superscalar processors in the context, design objective and implementation.

OoO in superscalar processors intends to tolerate both core pipeline and off-core memory latencies by exploiting instruction-level parallelism of a single execution context. On contrast, OoO in EP is designed to tolerate varied SIMD pipeline latencies due to on-chip memory bank conflicts, by exploiting thread/warp-level parallelism in a platform with thousands of hardware execution contexts. The different contexts and design objectives boil down to two major simplifications in EP implementation: (1) it allows instructions from different warps to be committed out of order—therefore no reordering at pipe writeback stage; and (2) out-of-order execution/commitment is enabled only between two instruction categories (memory/non-memory)—effectively forming a *partial out-of-order scheme*. Due to these simplifications, efficient hardware designs with low overhead are achieved (Sects. 4 and 5). To the best of our knowledge, EP is the first partial out-of-order GPU core pipeline scheme proposed in the literature (interestingly, the AMD next generation GPU architecture, named "Graphics Core Next" [22], adopted *limited out-of-order completion* similar to our proposal).

We have shown in prior sections that on-chip bank conflicts can be tolerated if the reduced (effective) memory bandwidth is not a problem. Similarly, it is also possible to cut the number of on-chip memory banks, if the reduced bandwidth due to fewer banks is not a problem for the bandwidth requirement of "common" GPU kernels. Our Elastic Pipeline proposal tolerates on-chip memory latencies with low hardware cost. As such, it has the potential to further simplify the on-chip memory design by reducing the bank number, while still maintaining system performance. This is attractive for GPUs with high memory banking cost (e.g., NVIDIA Fermi employs 32-way banked on-chip memories to feed its SIMD pipelines [9]).

In this paper we have assumed GPU *barrel processing* [4], where out-of-order instruction commitment is not a problem since the in-flight instructions are from different warps. In the case of relaxed barrel execution, our proposed Elastic Pipeline can still work with straightforward microarchitectural extensions, as described in Sect. 4.2.

Alternative solutions for GPU shared memory bank conflicts may include programming practices to avoid conflicts/reduce the conflict degree, by changing the data layout at source code level (e.g., *zero padding* to restructure data in shared memory) [11,23]. While such optimizations are widely adopted, *bank conflict resolution is not always possible in software*. More specifically, when memory access patterns are highly irregular and/or dependent on input data (such as the memory indirection/pointer-chasing pattern in AES), it is very difficult (if possible at all) to identify the conflict pattern by static analysis (being either compile-time GPU kernel code analysis or GPU runtime running on the CPU).

In addition, in such cases both our EP hardware and its corresponding (hypothetic) *GPU core runtime*[11] face the problem similar to branch prediction in a scalar processor: the access pattern of a scheduled warp memory instruction (A) must be used to decide the type of following warps to be scheduled (Sect. 4.1); but A's pattern is available only after it arrives at the pipeline memory stages (few cycles after A is scheduled). That's why the historic information has to be used to predict A's pattern in order to avoid any pipeline bubble. This type of job is best to be done by hardware (similar to CPU branch prediction), rather than runtime. To summarize, software bank conflict resolution is either impossible or inappropriate for highly irregular GPU on-chip shared memory accesses.

Even for GPU kernels with less irregular memory access patterns, manual data layout optimizations put *nontrivial burden on programmers* since (1) knowledge about memory hardware configurations of specific GPU platforms is necessary; (2) carefully restructuring the data and/or changing the access patterns are required. Optimizations based on static code analysis can automate this process, however, source level bank access optimizations also create *portability issues* when the code is to be deployed on platforms with different shared memory configurations. Delegating the optimizations to runtime eliminates the portability problem, but they can still be constrained by the extra memory space often required by data restructuring (e.g., when the kernel uses up GPU shared memory and thus zero padding is not possible). As such, we believe that hardware solutions with low implementation overhead (such as Elastic Pipeline) are superior to software solutions, for the GPU on-chip memory bank conflict problem.

This paper also assumes the execution stages and memory stages are not overlapped, therefore our proposed Elastic Pipeline design can make use of the existing "spare" (for non-memory instructions) MEM pipeline latches as the source of *elasticity* to tolerant the varied pipeline latency due to shared memory bank conflicts. However, this is not a mandatory requirement of our Elastic Pipeline proposal. For example, in the pipeline configuration with parallel execution/memory stages, we can insert some additional *spare pipeline stages* between the end of the parallel execution/memory stages and the beginning of writeback stages. With the extra stages as the source of elasticity our proposal can still work. Note the extra spare stages do not introduce any additional pipeline latency for ordinary execution without bank conflicts, thanks to the bypass buses (Fig. 2b). The only overhead is the hardware cost of the pipeline stage latches of the additional pipeline stages.

[11] In fact, current GPU SIMD cores do not support such low-level "GPU runtime" and they are not likely to support it due to their inefficiency in running scalar codes.

Besides GPUs, our Elastic Pipeline can also be applicable to a wide range of in-order pipeline designs adopting barrel processing and SIMD data path. The motivation is that the bank conflict problem exists generally in such architectures. Furthermore, although we target the varied execution latencies caused by on-chip shared memory bank conflicts in this paper, Elastic Pipeline can also be applied to cope with execution latency variation due to other shared resource conflicts (e.g., accelerator (such as FPU) port 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 data path throughput.

## 8 Related Work

Bank conflict is an important problem in vector processors and has been studied intensively in the past. To cope with bank conflicts of vector access across stride families, several techniques have been proposed, including the use of buffers [24], dynamic memory schemes [25–28], memory modules clustering [24], and intra-stream out-of-order access [29], 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 [18]. 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 data path. 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) [11,23]. Recently we also see some work in automating such manual optimizations [30,31]. Such high-level optimizations from the software side are complementary to our Elastic Pipeline proposal, albeit with the limitation discussed in Sect. 7.

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 [23]. 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 [32]; and (2) to integrate the CPU and GPU into the same die [8,33]. Besides, the DRAM memories attached to GPUs become a bottleneck for memory intensive kernels, and plenty of recent work on optimizing GPU off-chip memory access exists [34–38]. Our work addresses the problem of GPU on-chip shared memory bank conflicts, which seems largely orthogonal to the GPU

off-chip memory access work. However, the interaction and coordination between both on- and off-chip memory accesses may deserve investigation for further performance improvement, as suggested in Sect. 6.4.

## 9 Conclusions and Future Work

In this paper, we analyzed shared memory bank conflicts, and indicated how bank conflicts are translated into pipeline performance degradation. Based on our observations, 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 our future work, we will evaluate the effect of the proposed Elastic Pipeline on GPUs with L1 hardware caches. The heavily-banked caches also suffer from the dynamically varied access latencies in case of irregular memory access patterns, similar to the shared memory case. Besides, we also intend to improve the accuracy of bank conflict degree prediction for irregular memory access patterns. Since the Elastic Pipeline also interacts with off-chip DRAM accesses, we would also like to investigate co-optimization opportunities to improve the efficiency of both on- and off-chip memory accesses, for applications constrained by both on-chip bank conflicts and off-chip bandwidth.

**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). We would also like to thank the reviewers for their insightful comments.



## References

1. Gou, C., Gaydadjiev, G.N.: Elastic pipeline: addressing GPU on-chip shared memory bank conflicts. In: Proceedings of the 8th ACM International Conference on Computing Frontiers (CF'11) (May 2011), pp. 1–11 (2011)
2. GPGPU home page. http://gpgpu.org
3. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**, 103–111 (1990)
4. Thistle, M.R., 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 (1988)
5. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. Queue **6**, 40–53 (2008)
6. OpenCL home page. http://www.khronos.org/opencl/
7. Hou, Q., Zhou, K., Guo, B.: BSGP: bulk-synchronous GPU programming. In: ACM SIGGRAPH 2008 papers, SIGGRAPH '08. ACM, pp. 19:1–19:12 (2008)
8. AMD fusion. http://sites.amd.com/us/fusion/apu/pages/fusion.aspx
9. Glaskowsky, P.N.: NVIDIA's Fermi: the first complete GPU computing architecture

10. Little, J.D.C.: A proof for the queuing formula: L = w. Oper. Res. **9**(3), 383–387 (1961)
11. NVIDIA. CUDA best practice guide, edition 3.0
12. Bakhoda, A., Yuan, G., Fung, W., Wong, H., Aamodt, T.: Analyzing CUDA workloads using a detailed GPU simulator. In: IEEE International Symposium on, Performance Analysis of Systems and Software, 2009 (ISPASS 2009), pp. 163–174 (2009)
13. Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., Moshovos, A.: Demystifying GPU microarchitecture through microbenchmarking. In: IEEE International Symposium on, Performance Analysis of Systems Software (ISPASS, 2010), pp. 235–246 (2010)
14. Stark, J., Brown, M.D., Patt, Y.N.: On pipelining dynamic instruction scheduling logic. In: Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 33. ACM, pp. 57–66
15. Fung, W.W.L., Sham, I., Yuan, G., 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, MICRO 40. IEEE Computer Society, pp. 407–420 (2007)
16. Volkov, V.: Better performance at lower occupancy. In: GPU Technology Conference 2010 GTC '10 (2010)
17. Bhatnagar, H.: Advanced ASIC chip synthesis using synopsys design compiler physical compiler and primeTime. 2nd edn. Kluwer, Dordrecht (2001)
18. Rixner, S., Dally, W., Kapasi, U., Mattson, P., Owens, J.: Memory access scheduling. In: Proceedings of the 27th International Symposium on, Computer Architecture, 2000 (2000)
19. NVIDIA. The CUDA compiler driver NVCC, edition 2.2
20. NVIDIA. CUDA SDK version 2.2
21. Manavski, S. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: IEEE International Conference on, Signal Processing and Communications, 2007 (ICSPC 2007), pp. 65–68 (2007)
22. AMD Graphic Core Next Architecture. http://developer.amd.com/afds/assets/presen-tations/2620_final.pdf
23. Kirk, D.B., Mei, W., Hwu, W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, Los Altos (2010)
24. Harper, D.T. III.: Block, multistride vector and FFT accesses in parallel memory systems. IEEE Trans. Parallel Distrib. Syst. **2**(1), 43–51 (1991)
25. Harper, D.T. III.: Increased memory performance during vector accesses through the use of linear address transformations. IEEE Trans. Comput. **41**(2), 227–230 (1992)
26. Harper, D.T. III., Linebarger, D.A.: Conflict-free vector access using a dynamic storage scheme. IEEE Trans. Comput. **40**(3), 276–283 (1991)
27. Gou, C., Kuzmanov, G., 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, ICS '10. ACM, pp. 179–188 (2010)
28. Gou, C., Kuzmanov, G.K., 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? MAW '08. ACM, pp. 350–368 (2008)
29. Valero, M., Lang, T., Peiron, M., Ayguade, E.: Conflict-free access for streams in multimodule memories. IEEE Trans. Comput. **44**, 634–646 (1995)
30. Diamos, G.F., Kerr, A.R., Yalamanchili, S., 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, PACT '10. ACM, pp. 353–364 (2010)
31. Yang, Y., Xiang, P., Kong, J., 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, PLDI '10. ACM, pp. 86–97 (2010)
32. Gelado, I., Stone, J.E., Cabezas, J., Patel, S., Navarro, N., 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, ASPLOS '10. ACM, pp. 347–358 (2010)
33. Intel Sandy Bridge, Intel processor roadmap (2010)

34. Fung, W.W.L., Aamodt, T.M.: CPU-assisted GPGPU on fused CPU-GPU architectures. In: Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture HPCA '12 (2012)
35. Gou, C., Gaydadjiev, G.: Exploiting spmd horizontal locality. IEEE Comput. Archit. Lett. **10**(1), 20–23 (2011)
36. Lee, J., Lakshminarayana, N.B., Kim, H., Vuduc, R.: Many-thread aware prefetching mechanisms for GPGPU applications. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on, Microarchitecture, MICRO '43. IEEE Computer Society, pp. 213–224 (2010)
37. Narasiman, V., Lee, C.J., Shebanow, M., Miftakhutdinov, R., Mutlu, O., Patt, Y.N.: Improving GPU performance via large warps and two-level warp scheduling. In: Proceedings of the 44th Annual ACM/IEEE International Symposium on Microarchitecture MICRO 44 (2011)
38. Yuan, G., Bakhoda, A., and Aamodt, T.: Complexity effective memory access scheduling for many-core accelerator architectures. In: 42nd Annual IEEE/ACM International Symposium on, Microarchitecture, 2009 (MICRO-42), pp. 34–44 (2009)