Enabling SIMT Execution Model on Homogeneous Multi-Core System

KUAN-CHUNG CHEN and CHUNG-HO CHEN, National Cheng Kung University

Single-instruction multiple-thread (SIMT) machine emerges as a primary computing device in high-performance computing, since the SIMT execution paradigm can exploit data-level parallelism effectively. This article explores the SIMT execution potential on homogeneous multi-core processors, which generally run in multiple-instruction multiple-data (MIMD) mode when utilizing the multi-core resources. We address three architecture issues in enabling SIMT execution model on multi-core processor, including multithreading execution model, kernel thread context placement, and thread divergence. For the SIMT execution model, we propose a fine-grained multithreading mechanism on an ARM-based multi-core system. Each of the processor cores stores the kernel thread contexts in its L1 data cache for per-cycle thread-switching requirement. For divergence-intensive kernels, an Inner Conditional Statement First (ICS-First) mechanism helps early re-convergence to occur and significantly improves the performance. The experiment results show that effectiveness in data-parallel processing reduces on average 36% dynamic instructions, and boosts the SIMT executions to achieve on average 1.52× and up to 5× speedups over the MIMD counterpart for OpenCL benchmarks for single issue in-order processor cores. By using the explicit vectorization optimization on the kernels, the SIMT model gains further benefits from the SIMD extension and achieves 1.71× speedup over the MIMD approach. The SIMT model using in-order superscalar processor cores outperforms the MIMD model that uses superscalar out-of-order processor cores by 40%. The results show that, to exploit data-level parallelism, enabling the SIMT model on homogeneous multi-core processors is important.

CCS Concepts: • **Computer systems organization → Multicore architectures**; **Single instruction, multiple data**;

Additional Key Words and Phrases: Control divergence, data-level parallelism, openCL, SIMD processors, spatiotemporal SIMT

ACM Reference format:
Kuan-Chung Chen and Chung-Ho Chen. 2018. Enabling SIMT Execution Model on Homogeneous Multi-Core System. *ACM Trans. Archit. Code Optim.* 15, 1, Article 6 (March 2018), 26 pages.
https://doi.org/10.1145/3177960

6

1 INTRODUCTION

Single-instruction multiple-thread (SIMT) machine, such as a GPGPU, is an effective architecture for data-parallel workloads. Parallel programming models, such as OpenCL [17] and CUDA [29], are typically employed for these SIMT machines. In OpenCL programming model, a data-parallel kernel is a function that can be executed on computing devices. An instance to run the function

This work was supported by Ministry of Science and Technology, Taiwan, under Grant MOST 103-2221-E-006-266-MY3.
Authors' addresses: K.-C. Chen and C.-H. Chen, National Cheng Kung University, Department of Electrical Engineering, 1 Da-Tsuen Rd. Tainan, 70101, Taiwan; emails: edi751001@gmail.com, chchen@mail.ncku.edu.tw.

https://doi.org/10.1145/3177960

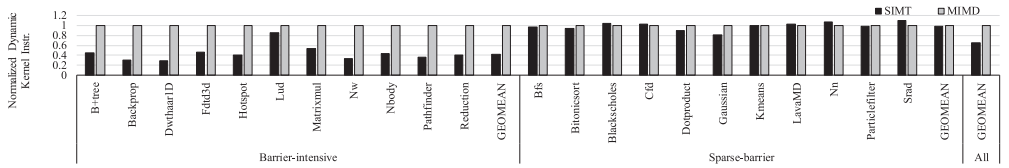


Fig. 1. Dynamic kernel instruction counts in SIMT or MIMD execution for OpenCL applications.

is called a work-item or a kernel thread. A number of work-items form a work-group, which is dispatched to a computing unit, e.g., a GPGPU streaming multiprocessor (SM) or a CPU core, for execution. A kernel can use the barrier built-in function for the synchronization of the work-items within the same work-group. By this abstracted execution model, the OpenCL programming model supports a unified programming interface for heterogeneous system architectures, including not only GPUs but also multicore CPUs and other types of processors.

For the OpenCL programming model, an SM can execute one or more work-groups and typically runs each of the work-items by a thread. A set of threads are grouped into a warp [28] or a wavefront [40] for execution in a SIMD lockstep fashion where the warps are scheduled by a hardware scheduler. This hardware warp scheduler also handles the barrier synchronizations among the threads. In this way, an SM can directly run the data-parallel threads, i.e., work-items, in fine-grained parallel computation.

On the other hand, a processor core of a homogeneous multi-core processor generally supports a single thread, assuming that simultaneous multithreading (SMT) is not used. In this way, multiple threads are executed by the processor cores in parallel to perform the multiple-instruction multiple-data (MIMD) operations. As the number of the threads often outnumbers the processor cores, thread context switching overhead occurs. The granularity for a thread execution must be sufficiently large to reduce these overheads. To this end, prior works for data-level parallelism applications typically use a control thread to serialize the work-item executions in a coarse-grained fashion [11, 14, 18, 21, 37, 38], such as work-item coalescing [21]. The granularity of this control thread often equals the size of a work-group in OpenCL model.

To examine the impact of running data-parallel applications in different processor execution models, we use a quad-core processor, referring to Table 2 for the system configuration, to run OpenCL kernels in both the SIMT and the MIMD operations, respectively. For the MIMD execution mode, the kernel functions, i.e., work-items, are serialized for executions by the control threads as described previously. For the SIMT mode, the concurrent work-items are executed in a warp-based fine-grained multithreading. The evaluated applications are from the OpenCL benchmark suits [1, 4, 29].

Figure 1 shows the dynamic kernel instruction counts of the SIMT execution normalized to the MIMD mode. For the applications that are barrier-intensive, we observe that switching among work-item executions in the MIMD mode has produced excessive dynamic kernel operations, by about 2.37 times more dynamic kernel instructions than the SIMT executions. For the other applications, SIMT and MIMD approaches have similar dynamic kernel instruction counts. The result indicates that a homogeneous multi-core processor running in SIMT execution model can eliminate the software approach overheads in terms of the dynamic kernel instructions, especially for the barrier-intensive applications.

The SIMT performance benefits can be gained for contemporary homogeneous multi-core processors, for instance, the ThunderX ARM processor that has 48 cores [2], the Tilera TILE64 processor [45] that has 64 cores, and the 61-core Intel Xeon Phi co-processor [15]. However, enabling

SIMT operation on a homogeneous multi-core processor faces several architecture level challenges. The first comes from the difference of the multithreaded execution mechanisms. Generally, an SM runs the threads concurrently by a hardware-assistance fine-grained multithreading scheme. In contrast, a homogeneous processor core switches executions among the threads based on a software approach. Like an SIMT GPU, a hardware support that allows the processor cores to run the concurrent threads in fine-grained multithreading is necessary to enable the SIMT execution model.

The next challenge is due to the fine-grained multithreading, which switches thread executions on each cycle [12]. To meet this cycle constraint, an SM often uses a vast register file to store all the concurrent thread contexts, allowing to access the thread contexts in time. However, the register file in a homogeneous processor core is generally employed for a single thread execution while the on-chip memories are used as caches. Hence, on a homogeneous multi-core system, an effective mechanism to access concurrent thread contexts in time is the key to achieve per-cycle thread-switching for SIMT operations.

The third challenge is to resolve the branch divergence problem, which decreases the SIMT efficiency. In an SM, when the warp threads take different execution paths due to a divergent branch, the warp goes through all the divergent paths sequentially. Each of the warp threads only commits the results of its taken execution path. On the other hand, this branch divergence problem never occurs in the MIMD execution mode. Thus, a solution that copes with the branch divergence is required when the processor runs in the SIMT mode.

In this article, we propose an SIMT execution model, which is integrated into an ARMv7 multi-core system, to achieve an SIMT/MIMD dual-mode architecture. This architecture can overcome the above challenges and alternatively runs the concurrent threads either in the SIMT or the MIMD mode, whichever is best for the performance. The proposed SIMT approach includes a spatiotemporal warp scheduling policy, an effective kernel thread context placement method, and a compiler framework with hardware support for thread re-convergences. Overall, we have made the following contributions in the system and micro-architecture design of a new multi-core application:

(1) The spatiotemporal warp scheduling policy enables the SIMT execution paradigm on a homogeneous multi-core processor. This SIMT execution model achieves about 2× speedup over the MIMD mode for the barrier-intensive kernels.
(2) The thread context placement method proposes an addressing scheme that allows to store the contexts of simultaneous kernel threads in the L1 data cache in SIMT execution mode. This mechanism greatly reduces the kernel-level context switch overhead; for the barrier-intensive kernels, only 20% of the memory instructions are needed for the SIMT executions compared to the MIMD mode.
(3) The proposed compiler-assisted thread scheduling mechanism, Inner Conditional Statement First (ICS-First), prevents the SIMT model from heavy performance loss caused by divergence executions and saves an average of 17% execution time compared to that without using the re-convergence methods.

The rest of this article includes the following sections. Section 2 presents the related works. Section 3 introduces the implementation details of the proposed dual-mode architecture. Section 4 presents the framework of the ICS-First mechanism, including a compiler framework based on an ARM-based system. Section 5 shows the evaluation results. Finally, Section 6 concludes this article.

2 RELATED WORK

Previous works [11, 14, 18, 21, 37, 38] exploring data-level parallelism on multi-core processors typically serialize the work-item executions in control threads. AMD's Twin Peaks [11] uses a

user-level thread to execute a work-item and runs these threads in sequence. Other approaches [14, 18, 21, 37, 38] dictate the work-item scheduling to the compiler framework. During compile-time, a kernel source code is divided into multiple code regions according to the synchronization barriers where each region is wrapped with a work-item loop for iteration execution. This method is called work-item coalescing in SNUCL framework [21]. Lee et al. have shown that the work-item coalescing technique is effective in several multi-core architectures [20, 21].

For multiple work-item executions, the control thread needs to replicate kernel variables for each of the work-items. This incurs kernel-context switching for accessing the replicated variables. Stratton et al. [38] demonstrated a selective replication concept to reduce the replicated variables when serializing CUDA kernels. Later, they proposed a variance vector approach for redundancy removal of kernel variables [37]. Jääskeläinen et al. [14] proposed to examine the kernel variable lifespans for replications by Single Static Assignment Control Flow Graphs, i.e., a compiler intermediate representation. Lee et al. [21] presented a web-based variable expansion to deal with variable expansions across different loop nests.

In contrast to the prior works, this article proposes to execute the bulk concurrent threads on a homogeneous multi-core processor in a hardware-assistance manner to avoid these software approach overheads. To the best of our knowledge, this work is the first to investigate how the kernel thread execution fashions, i.e., by the SIMT or the MIMD execution models, affect the performance on a homogeneous multi-core system. On the other hand, performance enhancement in exploiting data-level parallelism by vector instructions is investigated in several prior works. Stanic et al. proposed an integrated vector-scalar design on an ARM-based processor [36]. Jo et al. proposed a compiler optimization technique that employs both the implicit and explicit vectorization to improve the OpenCL kernel executions [16]. These works optimize the intra-thread execution by vector instructions or SIMD extensions. In addition to SIMD extensions, many CPUs, SPARC T5 [41] and Intel i7 [25] for instance, exploit thread-level parallelism by SMT technique [39], which enables processor cores to execute multiple independent instructions from different threads in each cycle.

Furthermore, several prior works studied the execution of applications on heterogeneous system with CPUs and GPUs [13, 30] or task migration [44] between them. Traditionally, the CPUs execute control tasks that offload the data-parallel workloads to the GPUs for higher throughputs. Wen et al. [44] presented a task scheduling scheme with a predication model to exploit all the heterogeneous computing resources, i.e., CPUs and GPUs. In contrast to these works, this article aims to improve the data-parallel executions on the homogeneous multi-core processor and proposes solutions to address the architecture issues in enabling SIMT execution model.

As mentioned in Section 1, control divergence problem needs to be considered in an SIMT execution model. Previous works resolving this problem can be classified into two categories according to the methods used: thread re-convergence and divergence mitigation. For the re-convergence mechanism, Fung et al. [10] used immediate post-dominator (IPDOM) to identify the potential re-convergence points at compile time. At runtime, divergent threads will re-converge at the IPDOM points through the stack-based execution. Diamos et al. proposed the thread frontier concept [8] for earlier re-convergence of divergent warp threads than the IPDOM approaches. Their work was based on the Min-PC approach, which is an implicit thread frontier re-convergence. However, these methods need either stack storages for IPDOM approaches or multi-layer PC comparisons for Min-PC approaches. On the other hand, this article proposes Inner Condition Statement First (ICS-First) mechanism to achieve re-convergence, which requires simpler hardware logic implementation.

Other previous works mitigate the divergent execution cycles by either compaction techniques [9, 27, 32, 42] or warp subdivision methods [26, 31, 33]. Compaction techniques dynamically

compact or re-form warps according to the active masks of the schedulable warps. This improves the SIMD lane utilization and then shortens the divergence execution. Fung et al. [9] proposed a thread block compaction mechanism where a thread block consists of threads within a work-group. This mechanism re-groups the threads of a thread block into warps for execution. Narasiman et al. [27] proposed a large warp microarchitecture that groups threads into large-size warps and dynamically creates SIMD width sized sub-warps from the active threads in a large warp. Vaidya et al. [42] presented an intra-warp compaction method to reduce the execution cycles of a divergent warp. Rhu et al. [32] introduced SIMD lane permutation technology that achieves a balanced permutation for warp threads and offers effective compaction.

In contrast, warp subdivision approaches split a warp when the warp encounters a divergent situation, which may be either a branch divergence or a memory divergence. In this way, the SIMT machine has more schedulable warps, which help to increase latency tolerance. Meng et al. [26] proposed a dynamic warp subdivision method that subdivides a warp upon divergences, including branch divergences and memory divergences. Rhu et al. [31] introduced a dual-path execution model that interleaves warp executions for intra-warp parallelism when the warp encounters a structured divergent control flow. Rogers et al. [33] proposed variable warp sizing to improve the performance of divergent kernels using a small warp size. Wang et al. [43] presented a multiple SIMD, multiple data architecture that permits the SIMD processor executing multiple divergent paths simultaneously. These divergent mitigation methods can be integrated with the proposed ICS-First mechanism to further improve SIMT efficiency, as we will suggest in Section 5.3.

In addition to control divergence problem, the warp scheduling policy also affects the performance of SIMT machines significantly. Most of the prior works studying the warp scheduling tackled the performance loss from long latency memory instructions. Rogers et al. [34] introduced a cache-conscious wavefront scheduling scheme to improve the cache efficiency by throttling the active threads or warps. Li et al. [22] presented a priority-based cache allocation method to reduce the cache pollution caused from massive active threads without throttling them. On the other hand, Liu et al. [23] proposed a barrier-aware warp scheduling scheme to reduce the warp-induced stalls rather than reduce the memory latency. In this article, we focus on the architecture supports in enabling dual-mode processor and only use the loosely-round-robin for the warp scheduling scheme. More comprehensive studies about the warp scheduler in homogeneous multi-core system can be investigated in the future.

3 SPATIOTEMPORAL SIMT EXTENSION ON A HOMOGENEOUS MULTI-CORE PROCESSOR

In this section, we elaborate on the required micro-architecture to enable SIMT execution based on a homogeneous multi-core system. This dual-mode architecture can run the data-parallel threads in either the SIMT or the MIMD mode.

3.1 Dual Mode CPU Architecture

Figure 2 shows the proposed SIMT/MIMD dual-mode architecture, which includes an SIMT co-processor for warp scheduling, logic for warp PC arbitration, and a warp context dispatching unit. To sustain multiple concurrent threads of execution, a per-core CPU thread descriptor table is used to store the execution states of all the sustainable threads running on the processor core.

We propose to use the private first-level data cache to store the concurrent thread contexts. In this way, a processor core is able to support multiple simultaneous threads for SIMT operations by re-using its existing cache memory. The threads scheduled on a processor core simultaneously are called CPU threads in this article.

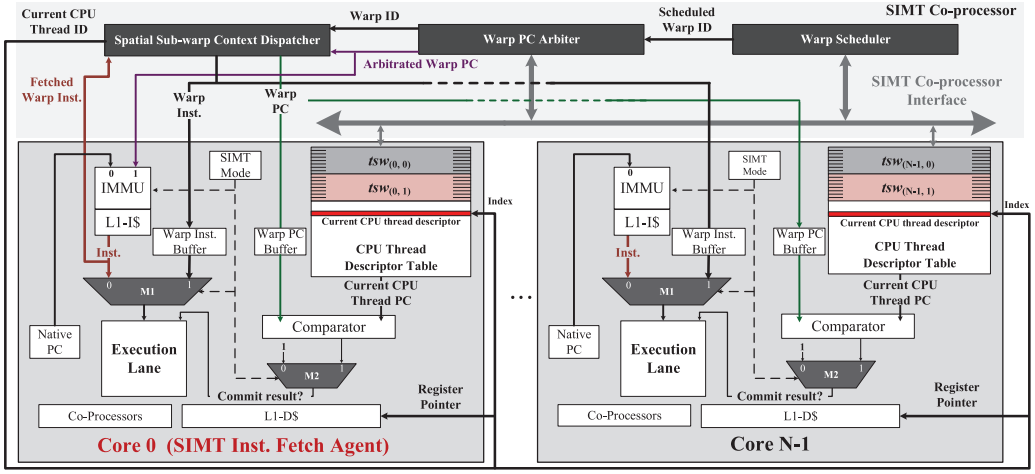


Fig. 2. SIMT/MIMD Dual-mode CPU architecture for a cluster of N processors.

We assume that processor cores are grouped into clusters and that a cluster is the basis of an SIMT unit, i.e., the analogy of an SM used to serve a work-group. In this way, a processor core can work as an SIMD lane, where the instruction to be executed is dispatched from the SIMT co-processor.

To schedule the massive number of CPU threads efficiently, we use a warp-based scheduling approach that groups a number of the CPU threads into a warp and schedules the instructions from different warps for fine-grained multi-threading execution. As shown in Figure 2, when the cluster performs the SIMT operation, the warp scheduler in the SIMT co-processor first selects a warp based on a given policy, and then the warp PC arbiter provides a warp PC for the scheduled warp. The warp PC is decided according to the thread execution states of the scheduled warp. The execution states of the warp are stored in the CPU thread descriptor table and read by the SIMT co-processor interface. The warp PC pointed instruction is fetched by the SIMT instruction fetch agent, a designated processor core in the cluster, and then executed by all the cluster cores to perform the SIMD operations.

Similar to the terminology used in the GPGPU, the warp size denotes the number of concurrent threads running on the same instruction stream while the SIMD width denotes the number of physical execution lanes. In this work, the SIMD width is the number of processor cores in a cluster. Because a typical SIMD width is usually less than the warp size, a warp will be executed in multiple execution rounds. Each execution round runs a SIMD width sized sub-warp; consequently, the number of execution rounds for a warp is the warp size divided by the SIMD width.

Since the threads in a sub-warp are executed by all the SIMD lanes in synchrony, it is also called a spatial sub-warp. On the other hand, those warp threads that are executed by the same SIMD lane in consecutive execution round form a temporal sub-warp [24]. Thus, the SIMT co-processor introduces a third component, a spatial sub-warp context dispatcher, which prepares the current CPU thread ID, warp instruction, and PC for a spatial sub-warp execution, and as a result runs a warp in the consecutive execution of spatial sub-warps.

In summary, the SIMT approach is achieved by adding the per-core thread descriptor table and the SIMT co-processor, which includes a warp scheduler, warp PC arbiter, and spatial sub-warp context dispatcher, to the cluster-based multicore processor. The detailed functionalities of these components are described in the following sections.

Table 1. Notation Used

Symbol	Description
p_i	Cluster processor core i, where $0 \leq i \leq SIMD\_width - 1$.
N_{ct}	Number of CPU threads in a processor core.
n_{act}	Number of active CPU threads in a processor core, counted by $WorkGroupSize/SIMD\_width$ where the work-group size is announced by OpenCL API function.
$t_{(i,j)}$	CPU thread j on processor core p_i, where $0 \leq i \leq SIMD\_width - 1$, $0 \leq j \leq N_{ct} - 1$.
N_{ISA_reg}	Number of general purpose registers that ISA defines.
N_{it}	Iterative execution times of a warp, evaluated by $Warp\_size/SIMD\_width$.
W_m	Warp m, where $0 \leq m \leq N_{ct}/N_{it} - 1$.
Wt_k	Warp thread k, where $0 \leq k \leq warp\_size - 1$.
$ssw_{(v,m)}$	Warp w_m's spatial sub-warp v, where $0 \leq v \leq N_{it} - 1$.
$tsw_{(i,m)}$	Warp w_m's temporal sub-warp i, where tsw_i runs on p_i.
TPM_i	Processor core p_i's on-chip memory, which is used as the thread privatization memory when p_i runs in SIMT mode.
TPO	Thread-privatization objects, which are used by a CPU thread for ISA registers and stack objects.
n_{tpo}	Number of thread-privatization objects allocated to a CPU thread, evaluated by $L1\_cache\_size/(n_{act} * ISA\_register\_width)$.
$TPO_{(i,j,d)}$	CPU thread $t_{(i,j)}$'s dth TPO.
$vTPO_{(i,m,d)}$	Temporal sub-warp $tsw_{(i,m)}$'s dth TPO.

3.2 CPU Thread Descriptor

A CPU thread descriptor stored in an entry of the CPU thread descriptor table is associated with a CPU thread. As shown in Table 1, we use the notation $t_{(i,j)}$ to denote a CPU thread j run on processor core i. As an example, for a table of 128 entries, the maximum number of concurrent CPU threads that can be assigned to a processor core for execution is 128, and the CPU threads of core zero can be denoted from $t_{(0,0)}$ to $t_{(0,127)}$.

To run a kernel program, a processor core cluster is assigned a number of work-items depending on the work-group size announced by the programmer. The work-items in a work-group are evenly distributed to the cluster cores for execution, and are scheduled as the CPU threads sequentially numbered from zero. To do this, a cluster processor core stores the work-item IDs into the CPU thread descriptors and initializes all the other descriptor items as the cluster is assigned a work-group. In the OpenCL approach, a cluster processor core starts executing these initialization functions when a work-group is created and dispatched to the cluster by the OpenCL API function, *clEnqueueNDRangeKernel*. Then, the cluster starts SIMT operation after all of the cluster processor cores finish the initialization functions.

The CPU thread descriptor is shown in Figure 3. The work-item identifier is a three-dimensional value (X, Y, and Z) that identifies which work-item is assigned to the CPU thread. Each CPU thread has a PC to point to the instruction that is going to run, and the condition code is used for conditional execution. The thread descriptor includes a state and priority that indicate the execution state of the corresponding work-item. The state bits identify the work-item execution in one of the following states: idle, ready, barrier, or execution. Figure 4 shows a CPU thread state

Fig. 3. CPU thread descriptor.

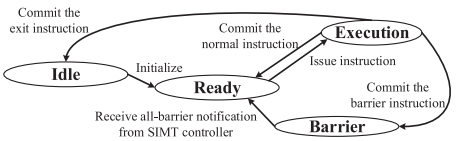


Fig. 4. CPU thread state transition.

transition diagram, where the arrows denote the processor core actions. The state begins with the idle state in which no work is assigned to the CPU thread. The ready state is entered as a work-item is assigned to the corresponding CPU thread, or returns from the execution or barrier state.

As a CPU thread is scheduled to execute a warp instruction, the state transfers to the execution state. Whenever the executed instruction reaches the end of the pipeline to commit its result, the state returns to ready if it is a non-barrier instruction or transfers to the barrier state if it is a barrier instruction. The barrier state indicates the requirement for bulk thread synchronization. When all of the associated CPU threads reach the barrier state, their CPU thread states become ready again for execution. Also, as a thread exit instruction is reached, this means the thread has run to completion and the state returns to the idle state. When all the threads have run to completion, the cluster processor cores return to the MIMD mode. While state transition to the ready or execution state can be signaled by the execution of an instruction or by processor core actions, the transition to the barrier or idle state is triggered by a specific instruction.

3.3 Warp Scheduling and Warp PC Arbitration

Once all of the cluster processor cores set up their respective CPU thread descriptor tables, they raise their SIMT mode registers to signal the SIMT co-processor that they are ready to enter the SIMT mode. The cluster processor cores then begin executing the warp instructions received from the SIMT co-processor.

A complete warp is divided into multiple temporal sub-warps, *tsw*, (see notation used in Table 1) according to the cluster processor core number, where a temporal sub-warp is formed by a chain of CPU threads in a cluster processor core. Given a warp size of 32 and an SIMD width of four, for instance, the first eight CPU threads (CPU thread ID from 0 to 7) of each processor core form the first temporal sub-warp while the second eight CPU threads (CPU thread ID from 8 to 15) of each processor core form the second temporal sub-warp, and so on. Consequently, the *m*th warp consists of the *m*th temporal sub-warps from the cluster processor cores. Hence, with 128 CPU threads per core, each processor core accommodates at most 16 temporal sub-warps; namely, there are at most 16 active warps that can be scheduled for execution each cycle in a cluster.

As a cluster runs in the SIMT mode, the warp scheduler reads the warp state bits through the hardwired logic from the CPU thread descriptor tables to decide the ready-to-run warps. A ready-to-run warp contains threads in either ready or barrier state with at least one thread in the ready state. One of the ready-to-run warps will become the candidate warp for execution according to the warp scheduling policy used, for example, a round robin policy. The warp PC arbiter determines a warp PC to use for the ready-state threads based on the given thread priority. We develop a compiler-managed thread priority scheme, which will be discussed in Section 4, to set the thread priority at runtime.

3.4 Instruction Fetch and Sub-Warp Scheduling

One way to fetch a warp instruction is by broadcasting the warp PC to the cluster processor cores where each uses this PC to fetch the instruction for execution, respectively. To coalesce the

redundant fetches of the same instruction, an alternative is to designate one of the processor cores to fetch the warp instruction on behalf of the other processor cores, as illustrated in Figure 2. As an example, the fetch unit associated with the instruction memory management unit (IMMU) of core zero is devoted to SIMT instruction fetching while the rest of the IMMUs have no operation in the SIMT mode. When the IMMU of core zero receives the warp PC, the fetch unit uses this PC to fetch the instruction, as in the conventional mode. The fetched instruction is returned to the warp context dispatcher for broadcasting later.

When the warp size is greater than the SIMD width, a warp is decomposed into multiple spatial sub-warps (*ssw*) for iterative executions. The number of iterative times is denoted as N_{it}. A spatial sub-warp, $ssw_{(v,m)}$, is identified by the iteration sequence number v and the warp ID m. In addition, the warp threads of warp m executed in the same SIMD lane form the temporal sub-warp, denoted as $tsw_{(i,m)}$, where i is the SIMD lane ID and m is the warp ID. Note that a fetched warp instruction will be executed iteratively N_{it} times for a warp execution, that is, the tsw size equals N_{it}.

In each execution round, all of the cluster processor cores execute the same warp instruction according to the current CPU thread ID, which associates with an *ssw*, in synchrony. Namely, an *ssw* (spatial sub-warp) consists of those threads running on each cluster processor core of the same CPU thread ID. Given a CPU thread $t_{(i,j)}$, which is associated with the spatial sub-warp $ssw_{(v,m)}$, the current CPU thread ID, j, can be obtained by Equation (1):

$$j = m * N_{it} + v, \tag{1}$$

where v ranges from 0 to 7, m ranges from 0 to 15 and N_{it} equals 8; given a warp size of 32, an SIMD width of 4, and a 128-entry CPU thread descriptor table.

Thus, an *ssw* (spatial sub-warp) context is a three-tuple value consisting of the warp instruction, the warp PC, and the current CPU thread ID (j in Equation (1)). The current CPU thread ID is broadcast from the SIMT co-processor for each *ssw* execution, and is used to index the CPU thread descriptor and to address the thread contexts in the on-chip memory, the L1 data cache in this work. The warp instruction and PC are broadcast for the first *ssw* and then buffered by the cluster processor cores for the remaining $N_{it} - 1$ spatial sub-warps until the warp is completed.

To support the SIMT approach while retaining the conventional processor execution model, we design two instruction fetch and commit paths for MIMD and SIMT modes, respectively. As shown in Figure 2, each processor core holds an SIMT mode register that indicates the operation mode. The M1 mux is simply used to select the instruction source for either mode of operation while the output of the M2 mux is an additional flag for the instruction commit. In the SIMT mode, if the warp PC and the current CPU thread PC, which is read from the CPU thread descriptor of an SIMD lane, are different, this means the SIMD lane diverges from the warp execution path. In this case, the output of M2 is used to disable the instruction commit operation.

3.5 Implementation of Thread-Privatization Memory

Figure 5 gives an example of using the processor core's L1 data cache for storing the concurrent threads' contexts when the core runs in the SIMT mode. In SIMT mode, the L1 data cache is called a thread-privatization memory (TPM), which is evenly shared among all the active CPU threads. The number of active CPU threads in a processor core is n_{act}, that is, the work-group size divided by the SIMD width. The thread-privatization memory is accessed in words or ISA-register width, e.g., 32-bit in our example. As a result, a CPU thread can obtain n_{tpo} 32-bit TPO referring to Table 1 where these TPOs are numbered from 0 to $n_{tpo} - 1$.

In general, a CPU thread's private memory includes ISA-defined registers and stack objects. This work proposes to use thread-privatization objects for the thread context storage as much as possible. In this way, when running in the SIMT mode, the data of CPU thread registers, i.e., the

per-thread ISA registers, are placed in the L1 data cache instead of in the original CPU register file. The data of a CPU thread register are stored in a *TPO* where the register ID d is used to index the *TPO*. Since the tsw threads running on the same processor core execute the same instruction stream, they require the registers of the same IDs for a warp instruction execution. Thus, the *TPO*s, which are indexed by d for the tsw threads in processor core i, are placed in a contiguous memory space for accessing in parallel, as shown in Figure 5.

Let $TPO_{(i,j,d)}$ denote the dth *TPO* allocated to the CPU thread $t_{(i,j)}$. The $TPO_{(i,j,d)}$ is accessed according to Equation (2), assuming that the object width is 32-bit, i.e., 4 bytes. The portion of the L1 data cache of processor core i, which is used as the thread-privatization memory in the SIMT mode, is denoted as TPM_i. Note that the CPU thread $t_{(i,j)}$ is associated with the spatial sub-warp $ssw_{(v,m)}$, referring to Table 1:

$$m = j \gg \log_2(N_{it}), \tag{2a}$$

$$v = j\&(N_{it} - 1), \tag{2b}$$

$$TPO_{(i,j,d)} = TPM_i[((m * n_{tpo} + d) * N_{it} + v)]. \tag{2c}$$

This placement helps to parallel access a set of *TPO*s for the threads in a tsw. The set of *TPO*s numbered d for the threads in $tsw_{(i,m)}$ is a vector *TPO* denoted as $vTPO_{(i,m,d)}$. For instance, the $vTPO_{(i,1,2)}$ in Figure 5 denotes the set of third *TPO*s (numbered from zero) for the threads in $tsw_{(i,1)}$. Each $vTPO$, which is placed in a cache line for parallel accessing, can be addressed by the warp ID m concatenated with the *TPO* ID d, i.e., $(m * n_{tpo} + d)$. Since the threads in a tsw will be executed in N_{it} rounds by a cluster processor core, the operand collectors are added to the cache to hold all the operands of the current instruction in a temporal sub-warp until all the threads are complete. A mux is used to select the operands of the executing thread by the iteration sequence number v of the spatial sub-warp $ssw_{(v,m)}$. This addressing scheme has been verified on a 32KB cache memory with cache line size configured to either 32 or 64 bytes.

3.6 Thread Stack Object Placement

As mentioned above, each CPU thread uses N_{ISA_reg} *TPO* to store the register data. N_{ISA_reg} denotes the number of ISA defined registers. However, the number of per-thread *TPO*s, n_{tpo}, usually outnumbers N_{ISA_reg}. For example, if the cache size is 32KB, and the number of cluster cores is four, a processor core will get 64 active threads out of the 256 work-items. In this case, an active thread can be allocated with 128 32-bit *TPO*s.

To fully utilize the higher speed L1 data cache in the SIMT mode, we use the rest $n_{tpo} - N_{ISA\_reg}$ *TPO*s to store the thread stack objects as shown in Figure 6. In this way, the data of a stack object are stored either in a *TPO* or in the main memory. Before discussing how to achieve this, we introduce the stack memory allocation for all the active CPU threads on a processor core.

As each of the active threads has its own private stack, the stack pointers (SP) of these threads have to be initialized to different values. To achieve this, the OpenCL runtime assigns a stack segment to a processor core where the segment will be distributed to the active threads using a predefined thread stack size. Assuming the SP initially points to the top of stack of the respective threads, which is the highest address in the stack, and its value grows down as the stack objects increase, the SP value of an active thread j is initialized by Equation (3):

$$SP_j = TOS_{seg} - j * Size_{stack}, \tag{3}$$

where the TOS_{seg} is the top of the stack segment; j is the CPU thread ID, and $Size_{stack}$ is the stack size allocated to an active thread.

Figure 7 depicts the mapping between a stack object to a *TPO*. Considering that the predefined thread stack size is a power of two, a stack object address can be decoded into a segment number,

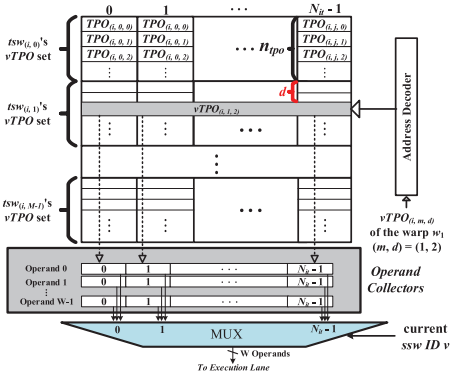


Fig. 5. Data cache used as the register file for processor core p_i.

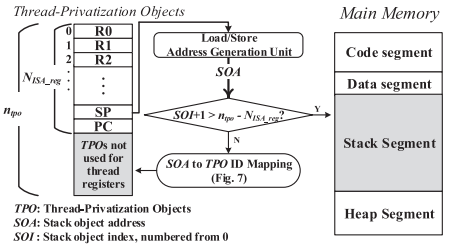


Fig. 6. Stack memory arrangement for a CPU thread.

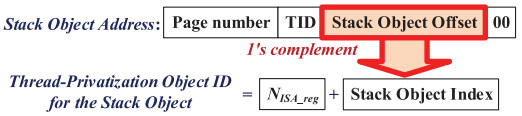


Fig. 7. Conversion from stack object address to thread-privatization object ID.

CPU thread ID, and a stack object offset. Assuming 32-bit word is used, the stack address, which will be redirected to a *TPO*, is aligned to the 32-bit width; thus, the last two bits are ignored when generating the object ID.

Since the stack object address decreases as the number of stack objects increases, the stack object index can be obtained by the one's complement of the stack object offset. As shown in Figure 6, a stack object can be stored in a *TPO* when its index is less than or equal to $n_{tpo} - N_{ISA\_reg}$. A *TPO* for a stack object can be indexed by the addition of the stack object index and the base N_{ISA_reg}. In addition, by adding a memory attribute to the segment descriptor, the memory management unit is able to determine whether a memory address is located in the stack segment or not.

In this way, we have provided a uniform addressing scheme for accessing the thread's ISA registers as well as the thread stack objects from per-thread thread-privatization objects. By this uniform addressing scheme, the processor cores are able to access the private thread contexts in time for fine-grained multithreading. In addition, by storing the data of stack objects in the thread-privatization objects, the cache memory capacity can be utilized effectively.

4 COMPILE-ASSISTED THREAD SCHEDULING

This section describes the Inner Conditional Statement First thread scheduling algorithm as well as its compiler-assisted framework. We also give an implementation example of the warp PC arbitration based on thread priority.

4.1 Inner Conditional Statement First Strategy

The "inner conditional statement first" scheduling policy achieves thread re-convergence by leveraging the following observation: Since a warp thread divergence generally comes from a condition statement, a re-convergence chance may arise when the warp threads finish the inner conditional statement and then rendezvous with other threads at the outer statement.

Specifically, the proposed mechanism gives an inner conditional statement of depth *i* a higher priority for execution than its outer statement of depth *i-1*. In this way, the priority of a conditional statement is determined by the depth of the nested conditional structure. Thus, the outer statement

can proceed as soon as there is no more inner conditional statement left for execution. As a result, re-convergence may occur, since the faster threads will wait at the outer statement.

The ICS-First mechanism consists of compiler insertion framework and adjustment of CPU thread priority. The compiler has to insert the priority setting instructions at the proper locations for the priority-level adjustment. The warp PC arbiter then references the CPU thread priorities to prepare the warp PC. The following sections introduce the compiler framework and present a case study of the priority-based thread scheduling mechanism.

4.2 Compiler-Assisted Insertion Framework

To achieve ICS-First execution, we introduce three priority setting operations: *raise-the-priority*, *lower-the-priority*, and *set-outmost-priority*. The *raise-the-priority* operation increments the CPU thread priority level by one while the *lower-the-priority* operation downgrades one level of the priority. The *set-outmost-priority* operation defines a thread not entering any conditional statements, which in default has the lowest priority level.

To insert priority setting instructions, the kernel source code is first transformed into an abstract syntax tree (AST), which consists of data structure nodes denoting the declarations of functions or variables, and statements. Algorithm 1 depicts the rundown of finding the insertion points in the AST nodes through checking the condition statements. Clang, a C language family front-end compiler of LLVM framework, is employed to generate the AST structure and the kernel source code with the insertion tags. To insert these tags at the correct location, each AST node uses four member functions for position identification: *begin()*, *end()*, *before()*, and *after()*.

The *begin()* and *end()* functions identify the beginning and end of the statement, respectively. The *before()* function indicates the position just in front of the statement while the *after()* function locates the successor points where the AST tracer leaves the statement. A priority instruction pragma is inserted at one of these positions by adding an extra corresponding AST node to the tree structure. In addition, we further use member functions: *isConditionStatement()* and *isConditionalStatementBody()*, to identify whether an AST node is a condition statement or a conditional statement body.

At first, the "*set-outmost-priority*" pragma is inserted to indicate the beginning of the kernel function, which has the lowest priority. A kernel function is found as a function declaration node, which has kernel attribute. As tracing the underlying AST node of the kernel function, whenever

ALGORITHM 1: Priority pragma insertion

```
Input: AST-Tree = KernelSourceCode.GetAST-Tree();
pragma-stack = NULL;
for every Decl *D in AST-Tree do
    if D is a FunctionDecl FD && FD.isKernelFunc() then
        FD.begin().set-outmost-priority();
        for every Stmt S in FunctionDecl FD do
            if S.isConditionStatement() then
                S.before().raise-the-priority();
                pragma-stack.push();
            else if S.isConditionalStatementBody() then
                if AST tracer reaches S.end() then
                    pragma-stack.pop();
                    if pragma-stack == NULL then
                        S.after().set-outmost-priority();
                    else
                        S.after().lower-the-priority();
```

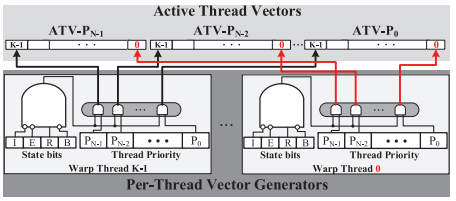


Fig. 8. Active thread vector generation logic, N priority level, where warp size is K.

```
Procedure HighestPriorityActiveThreadArbitration
Inputs: ATV-P_n, n is from 0 to N-1 where N is support priority level
Output: Highest priority Warp Thread ID, denoted as WTID
1.    for(n = N-1 to 0){
2.        if(ATV-P_n != 0){
3.            WTID = ((~(ATV-P_n)) + 1) & ATV-P_n
4.            break;
5.        }
6.    }
7.    return WTID;
```

Fig. 9. Pseudo-code of highest priority warp thread arbitration.

the AST tracer sees a conditional statement, such as an if-statement or a loop-statement, the algorithm inserts the "*raise-the-priority*" pragma before that statement.

As the AST tracer finishes tracing a conditional statement, there are two candidate pragmas, "*lower-the-priority*" and "*set-outmost-priority*," will be inserted according to whether the inserted location is out of any conditional statement or not. We use a stack, which is initialed to NULL at beginning, to determine the inserted pragma. A flag is pushed onto the stack each time "*raise-the-priority*" is inserted. A pop operation of the stack is performed each time the AST tracer finishes tracing a conditional statement. For a null stack, "*set-outmost-priority*" is inserted, otherwise, "*lower-the-priority*" is inserted. Finally, all pragmas are translated into the corresponding instructions when generating the binary code, i.e., co-processor instructions MCR in ARM-based ISA.

4.3 Case Study: Per-Thread Priority Adjustment and Vector-Based Warp PC Arbitration

Referring to Figure 3, a CPU thread state consists of a thread priority and a four-state bit vector, including *Idle*, *Ready*, *Execution*, and *Barrier*, respectively. Furthermore, an N-level thread priority is also implemented as an N-bit vector, where each bit indicates a priority level. Whenever a processor core encounters a priority setting instruction, the core uses the current thread ID to index the CPU thread descriptor to update the thread priority level.

The current warp PC is selected from the warp threads that have the highest execution priority level. In an N priority-level architecture, there are N active thread vectors (*ATV*) to show the active threads for each priority level. If a warp consists of K CPU threads, then each *ATV* is a K-bit vector, where a bit shows whether the corresponding CPU thread is active or not at a given priority level. Since a processor core provides N_{it} CPU threads to form a warp, a CPU thread $t_{(i,j)}$ can be identified as a warp thread wt_k by Equation (4), where $t_{(i,j)}$ is associated with the spatial sub-warp $ssw_{(v,m)}$, and v is evaluated by Equation (2b):

$$k = i * N_{it} + v = i * N_{it} + (j\&(N_{it} - 1)). \tag{4}$$

The warp thread ID k, ranging from 0 to K-1, is used to address the *ATV*'s bit position of a given priority level. At priority level n, the kth bit of the active thread vector, $ATV\text{-}P_n[k]$, stands for whether wt_k is active. $ATV\text{-}P_n[k]$ is determined by Equation (5):

$$ATV\text{-}P_n[k] = (I' \cdot E' \cdot R \cdot B') \cdot P_n, \tag{5}$$

where I, E, R, and B are thread state bits; P_n is the nth bit of the thread priority; k is from zero to K - 1, and n is from zero to N - 1.

$ATV\text{-}P_n[k]$ has a logic value of one if and only if the wt_k is active (R is true) as well as turned on at priority level n. Figure 8 illustrates the logic diagram for the *ATV*s of N priority levels. Figure 9 illustrates the pseudo-code that arbitrates the highest priority warp thread. This procedure takes

Table 2. Dual-Mode Multi-Core Platform Configurations

Single Issue In-Order Platform Configuration	
Platform Component	**Baseline Configuration**
core cluster	a cluster has 4 cores
core model	in-order, ARMv7-A ISS, VFP 3.0, 1GHz
L1 I/D-Cache (private)	4-way, 64B line, 32KB, 1 cycle delay, write through, perfect write buffer
L2 Unified Cache (private)	8-way, 64B line, 256KB, 8 cycles delay, MOESI cache coherence, write back
I/D TLB	32 entries, 20 cycles miss penalty
Main memory	200 cycle latency
SIMT Architecture Configuration	
Simultaneous CPU threads	Up to 128 CPU threads per core
Warp scheduling	Round robin, warp size is 32
Warp thread scheduling	ICS-First
Compiler Framework	
front-end	Clang 3.7
back-end	LLVM 3.7 & GCC 4.9.2
work-item coalescing	SnuCL with LLVM2.9

Table 3. The Superscalar Processor Core Variances

	MIMD mode Out-of-Order	**MIMD mode In-Order**	**Dual-mode**
Execution units	2 Integer units, 2 Floating point units, and 1 Load-store unit, Compatible with ARM v7 and VFP 3.0, 1 GHz		
Issue width	3	2	3
Commit width	3	2	3
Inst. fetch queue	32		
Instruction widow	128		
Branch predictor	2-bit, 4096 entries. (only used in MIMD mode)		
BTB entry	4,096 entries. (only used in MIMD mode)		
Load-store queue	32, support hit under multiple miss read	32, in-order execution	32, in-order execution

all the $ATV\text{-}P_n$s as its inputs and then outputs the highest priority ***WTID*** using a one-hot bit vector, where the logic one position indicates the warp thread ID.

5 EXPERIMENTAL EVALUATION

In this section, we present a detailed evaluation of the proposed dual-mode architecture.

5.1 Methodology

We model the proposed dual-mode multi-core architecture in SystemC modules. First, we implement a single issue in-order processor core, see Table 2. The approximately timed SystemC instruction set model is fully compatible with the ARM v7 architecture and has been verified by booting the Linux OS [3]. All simulated processor cores have their private cache systems, including an L1 I-cache, an L1 D-cache, and an L2 unified cache. The cache systems and the SIMT co-processor are simulated in a cycle accurate model. Each processor core is designed to support at most 128 concurrent CPU threads for the SIMT mode. The cache coherence is handled with the MOSEI protocol. The cache coherence mechanism is employed to retain the workgroup shared memory consistency specified in the OpenCL framework.

The processor core configuration in Table 2 is a single issue in-order design, which is used for the major discussions of the SIMT/MIMD execution model. We also model a cycle-accurate superscalar processor core in SystemC for further evaluations. The variances of processor core architectures are shown in Table 3. We implement two superscalar architectures including

Table 4. Benchmarks

Barrier-intensive benchmarks				Sparse-barrier benchmarks			
Benchmark	Static Barriers	Dynamic Barriers	BPKI	Benchmark	Static Barriers	Dynamic Barriers	BPKI
B+tree (R)	5	13056000	45.85	Bfs (R)	0	0	0
Backprop (R)	6	10485760	46.26	Bitonicsort (A)	0	0	0
Dwthaar1d (A)	3	623552	62.09	Blackscholes (N)	0	0	0
Fdtd3D (N)	2	4194304	16.19	Cfd (R)	0	0	0
Hotspot (R)	3	7573504	13.50	DotProduct (A)	0	0	0
Lud (R)	6	333056	4.32	Gaussian (R)	0	0	0
MatrixMul (N)	2	128000	11.40	Kmeans (R)	0	0	0
Nbody (A)	2	81920	2.50	LavaMD (R)	0	0	0
Nw (R)	12	8587264	43.29	Nn (R)	0	0	0
Pathfinder (R)	3	4569660	38.03	Particlefilter (R)	10	1092096	0.27
Reduction (A)	4	21201920	38.28	Srad (R)	0	0	0

R:Rodinia, N:NVIDIA SDK, A:AMD SDK.

in-order and out-of-order processors for the MIMD approaches. The configurations of the in-order and the out-of-order superscalar processor core are similar to the ARM Cortex A7 and Cortex A15 architecture, respectively.

Then, we add the SIMT co-processor to the in-order superscalar processor for the dual-mode implementation. In contrast to the in-order and the out-of-order cores exploiting instruction-level parallelism of a single thread, the proposed SIMT model leverages data-level parallelism for utilizing the execution units in parallel. To fairly compare the efficiency of the different ways achieving parallel execution, the issue and commit widths of the dual-mode processor are all set to three as that of the out-of-order processor. The branch prediction mechanisms do not work in the SIMT mode, since the branch divergence is handled by the SIMT co-processor. The other system configurations are the same as listed in Table 2 and the detailed evaluation result is given in Section 5.5.2.

We choose OpenCL framework as the data-parallel programing model. Since our work focuses on the thread or the work-item executions, the simulation platform only performs the kernel executions while the runtime APIs are executed by the host machine to achieve an acceptable simulation time [5]. The kernel source insertion mechanism for the ICS-First algorithm is based on the LLVM framework [19]. The instrumented kernel source is generated by the Clang front-end, libTooling, and is then translated into the assembly code and binary code via the LLVM static compiler and GCC, respectively.

To make a comparison with the software-based MIMD approach, we employ the SnuCL compiler [18] to build serialized OpenCL kernels and run them on the proposed platform as well. Since the SnuCL compiler supports work-item coalescing and also eliminates the unnecessary variable replications, it represents the performance of the state-of-the-art MIMD approaches for the OpenCL kernel execution on multi-core processors. The platform used to run the OpenCL kernels in MIMD mode has the same configurations as that of the SIMT mode, except without the SIMT architecture supports. With SnuCL support, each processor core runs an independent thread that executes a serialized kernel to serve a work-group at a time as there is no hyper-threading support.

We use OpenCL applications from the NVIDIA SDK [29], AMD SDK [1], and Rodinia benchmark suites [4] for system evaluations. The benchmarks are classified into barrier-intensive and sparse-barrier ones. The two types of the benchmarks show how the kernel features, including variable replication, data locality, and synchronization frequency, impact the execution mode efficiency. Table 4 lists the benchmarks along with static barrier count, dynamic barrier count, and barrier instructions per kilo kernel instructions (BPKI) that reveals the synchronization frequency.

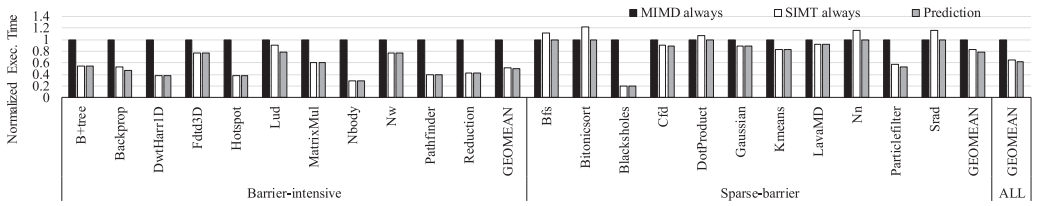
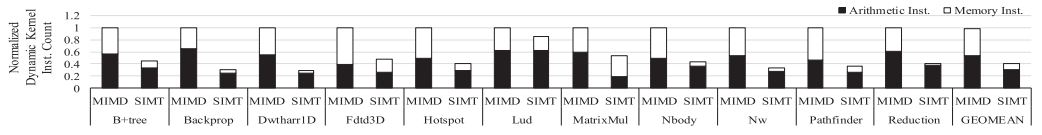
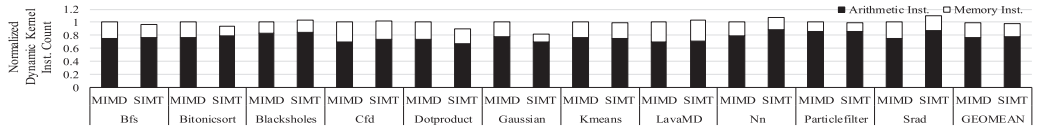


Fig. 10. Overall performance comparison.



(a) Barrier-intensive kernels.



(b) Sparse-barrier kernels.

Fig. 11. Dynamic kernel instruction count comparison.

Moreover, the ARM-v7 ISA supports the advanced SIMD extension. In general, explicit vectorization and implicit vectorization are the two methods to use the SIMD extensions for data-parallel kernel executions. The explicit vectorization requires the programmer to implement the vectorization kernel manually while the implicit one relies on compiler optimization to generate vectorization codes. However, for an ARM-based machine, the auto-vectorization mechanism of the contemporary compilers, e.g., GCC, hardly finds data-level parallelism in work-item loops [16], and this limits the efficiency of the implicit vectorization. To compare the kernel executions of using the SIMD extension, we select ten benchmarks including five barrier-intensive and five sparse-barrier applications shown in Figure 16, and modify their kernels in hand for explicit vectorization. The vector length is four, which means that the modified kernel packets four work-items of the original kernel into one for vectorization. The evaluation result of the vectorization kernel executions is discussed in Section 5.5.1.

5.2 Performance Overview

Figure 10 shows the normalized execution time for executing the kernel threads in the SIMT mode with the proposed ICS-First mechanism or in the MIMD mode with the work-item coalescing (WC) scheme proposed by SnuCL. Overall, the SIMT execution achieves geometric mean speedups of 1.52× over the work-item coalescing technique.

In general, the SIMT execution significantly speeds up barrier-intensive applications (their BPKIs are typically larger than 10) by on average 1.92× speedup over WC. For these barrier-intensive applications, the MIMD execution model relies on the software implementations to switch work-item executions for each of the code regions split by the barrier functions. As a result, more dynamic instructions are used in switching executions and then lead to more execution time. As shown in Figure 11(a), for the barrier-intensive kernels, their MIMD executions use extra 1.5 times

dynamic instructions compared to the SIMT executions, as expected. On the contrary, for the sparse-barrier kernels, their executions have similar dynamic kernel instruction counts in both SIMT and MIMD modes, which is shown in Figure 11(b).

Moreover, since the kernels use barriers to identify the parallel regions, the work-item coalescing technique performs selective variable replication that expands the kernel variables spanning multiple parallel regions to memory arrays. This produces memory-register transfers for the replicated variables. In contrast, when using the SIMT execution mode, the majority of these local variables, which are usually allocated to a register by the compiler, are directly mapped onto the L1 D-cache memory by the proposed addressing scheme, and as a consequence, the frequent data transfers for local variables are avoided. As a result, for the MIMD execution model, about 3.64 times more dynamic memory instructions are used for the transfers of the replicated variables.

These results show that the software approach overheads depends on not only the synchronization frequency, which can be represented by BPKI, but also the synchronization quantity related to the number of replicated variables. Given Nbody kernel for instance, although its BPKI is not relatively high (only 2.5), this kernel uses a large amount of variables that needs to be replicated. Consequently, this kernel requires about 7.13 times of dynamic memory instructions for the MIMD execution against to the SIMT mode.

On the other hand, the SIMT execution model only has little performance gains for some barrier-intensive kernels, *Fdtd3D*, *lud*, and *nw* for example. In *Fdtd3D* and *lud* kernels, there are few private variables used across multiple barrier regions. Hence, the compiler can optimize the two kernels by selective variable replication effectively when serializing the work-item executions for the MIMD execution. For *nw* application, since its kernel is simple and repeatedly used many times during the execution, the dual-mode processor frequently switches the execution mode and as a result the overheads increase.

For the sparse-barrier kernels, the SIMT mode performs poorly in the *Bfs*, *Bitonicsort*, *DotProduct*, *Nn*, and *Srad* due to thread initializations. In the SIMT mode, the active threads have to initialize their SPs, PCs, and thread descriptors before serving the work-items. This initialization overhead is unavoidable when using the SIMT approach. In contrast, the variable replication overhead coming from work-item coalescing in the MIMD operation can be thoroughly eliminated by selective replication when the kernel is synchronization free, e.g., *Bfs*, *Bitonicsort*, *DotProduct*, *Nn*, and *Srad*. As a result, the SIMT executions of these five introduce average 16% more execution time as compared to that required in the MIMD mode enhanced with selective replication.

Although the MIMD execution can use selective replication to eliminate the variable replication overhead, *Blacksholes* for instance, its SIMT executions can achieve 5× speedup over the MIMD mode. This is because the kernel has good spatial and temporal data locality among the work-items. Hence, the SIMT operation, which schedules the work-items in a warp-sized barrel method rather than in sequence as the work-item coalescing does, can leverage the data locality more effectively. In addition, the *ParticleFilter* kernel is classified as the sparse-barrier one due to the relatively low BPKI. Although the kernel is not barrier intensive, its SIMT execution also has 1.74× speedup compared to the MIMD execution. This notable speedup also comes from the efficient data locality for the SIMT execution.

According to the above results, for some particular kernels, the SIMT execution model is outperformed by the MIMD mode due to effective compiler optimizations. Even so, our dual-mode architecture can run in the MIMD mode with work-item coalescing or other approaches to obtain the best performance for these kernels. The favorable kernel execution mode (using SIMT or MIMD operation) can be estimated with a compiler-based analysis. At the compile time, deterministic kernel features, such as synchronization frequency, data locality, variable replications, and branch divergences, can be extracted and used in the performance-oriented prediction for the

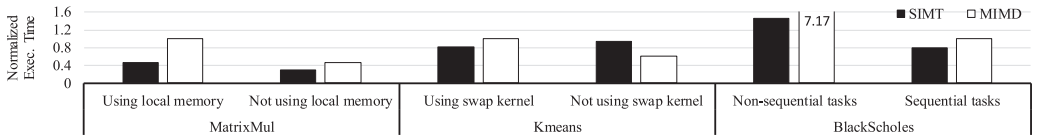


Fig. 12. Comparison of kernel optimizations.

operational modes. In our other work [6], we have proposed a prediction model using machine learning to achieve this goal and the prediction accuracy is 95%.

Observing this, our dual-mode SIMT/MIMD architecture uses a coarse-grained mode selection policy. That is, the OpenCL runtime system decides the execution mode to run for a kernel and uses the selected mode throughout the execution. We will show that a fine-grained execution mode policy such as the alternate execution model, which will be discussed later in Section 5.3, generally makes no benefits for most of the kernels. As shown in Figure 10, this prediction execution model can outperform the SIMT-always and the MIMD-always models by 4.57% and 59.74%, respectively. Hence, the dual-mode architecture can always offer the best performance by using the suitable execution mode for data-parallel kernels.

The above experiment results are based on the original benchmarks without any modification. However, the performance of the kernel execution is significantly affected by the optimizations on the computing devices. To further discuss this issue, we use *MatrixMul*, *Kmeans*, and *BlackScholes* as the examples, and modify their kernels by removing the GPU-specific optimizations. Figure 12 shows the normalized execution time for executing the modified and the unmodified kernels in both the SIMT and the MIMD modes. The normalized base is the execution time of running the unmodified kernel in the MIMD mode.

The original *MatrixMul* kernel employs the local memory to reduce the accesses to the global memory. The GPUs can benefit from this, since an SM can use its high-speed internal memory to store the OpenCL memory objects in the local memory. However, the processor cores of CPUs generally use the internal memory for cache system. Thus, the OpenCL memory objects allocated in either the global memory or the local memory are all stored in the main memory and cached by the hardware, which makes the optimization a downside to CPUs. Therefore, we disable the use of local memory in the *MatrixMul* kernel. As shown in Figure 12, this improves the SIMT and MIMD executions by 1.57× and 2.16× speedups, respectively. The reason for the significant speedup is because the redundant memory transfers between the local and the global memories are eliminated. Moreover, the MIMD execution has more notable enhancement, because the barriers for the local memory synchronizations are removed.

The second application, *Kmeans*, uses a swap kernel to remap the data array from row-major order to column-major order for the better memory access pattern on the SIMT machine. However, this column-major layout is inefficient on the CPU platform [35]. Thus, we disable the use of the swap kernel in the modified version, which improves the MIMD execution by 1.61× speedup. And as expected, for the SIMT execution, about 12% extra execution time are required, since the data layout of the modified *Kmeans* kernel are inefficient for SIMT operation.

Finally, the application *BlackScholes* calculates the price of European put and call options. Assuming m work-items are used to evaluate $optN$ options, each of the work-items will calculate $optN/m$ options. In the unmodified version, the indexes of the options assigned to a work-item for evaluation are stridden by m. As a result, the load/store data for the option evaluations are also stridden in the main memory, which dramatically decreases the cache efficiency. By assigning the sequential tasks to each work-item, the MIMD execution is enhanced significantly by 7.17×

Table 5. Comparison of Divergence Optimization Techniques

	Min-PC	ICS-First (This work)	Compaction	Warp sub-division	Alternate execution model (ALT)
Target	early re-convergence	early re-convergence	divergence mitigation	divergence mitigation	divergence mitigation
Hardware support	N-level PC comparators	Bit-vector PC arbiter	Active mask stack with compaction logic	Active mask stack or table for each sub-warp	Synchronous MIMD support
Overhead	High (operation)	Low (operation)	Medium (storage)	High (storage)	High (Multiple instruction streams)

speedup. As this way improves the cache efficiency, the SIMT execution is also enhanced by 1.84× speedup, which makes the SIMT mode still outperform the MIMD mode by 21% less execution time. As we can see, the used program optimizations directly affect the kernel features. Thus, by the proposed dual-mode design, the processor will have potential to effectively execute the kernels in a proper execution mode without tuning the kernels case by case.

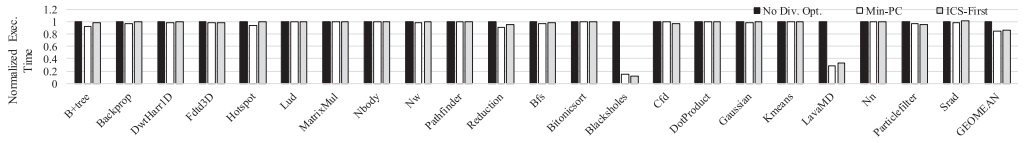
5.3 Comparison of Divergence Optimizations

Typically, a divergence problem can be optimized in early re-convergence and divergence mitigation. The ICS-First and Min-PC [7, 8] focus on re-converging the divergent threads as soon as possible while other previous works mitigate the divergent execution before warp threads reconverge either by the compaction mechanisms [9, 27, 32, 42] or by the warp sub-division [26, 31, 33] techniques. Table 5 shows the comparisons of these divergence optimization methods. As we have mentioned in Section 2, our ICS-First mechanism can be integrated with other divergence mitigation approaches. To explore the integration benefits, we implement the ICS-First mechanism with SIMD Lane Permutation (SLP) [32] and Dual-Path Execution (DPE) [31] models to represent the thread compaction and warp sub-division approaches, respectively.

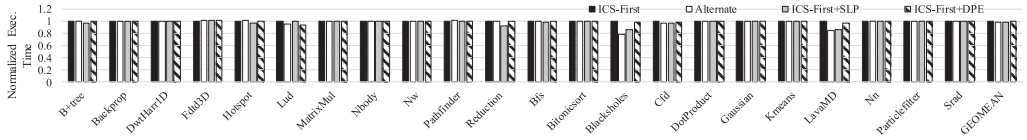
Another optimization is to switch the SIMT execution to the MIMD as soon as a divergence occurs in the SIMT mode, i.e., the alternate execution model (ALT). To do this efficiently, we assume running the divergent warps in a synchronous MIMD execution fashion, where the SIMT co-processor prepares the respective warp PCs to the processor cores that execute the different instructions synchronously. Note in this way, the non-divergent warps still run in the SIMT mode. In the ALT model, alternatively, each processor runs their respective instructions asynchronously, i.e., the asynchronous MIMD execution. However, in this asynchronous MIMD mode, scheduling the concurrent MIMD and SIMT threads is a complicated issue. Thus, for the ALT model, the synchronous MIMD model is used.

Table 5 illustrates the comparison of these divergence optimization techniques. For early reconvergence, the proposed ICS-First mechanism only requires a bit-vector PC arbiter, which can be performed in bitwise operations, while the Min-PC arbitration needs a multiple-layer PC comparison. To implement the divergence mitigation methods based on the compaction or the warp sub-division, a warp is given an active mask stack, which is used to check for compaction opportunities or to control the possible divergent paths.

Figure 13 shows the normalized execution time of using ICS-First, or its variants over the Min-PC mechanism for thread re-convergence. Figure 13(a) compares the Min-PC approach and our



(a) Comparison of re-convergence techniques (execution time is normalized to No Div. Opt.).



(b) Comparison of divergence mitigation techniques (execution time is normalized to the ICS-First approach).

Fig. 13. Performance comparison of divergence optimizations.

ICS-First mechanism. To demonstrate the importance of thread re-convergence, the normalized execution time of excluding any optimizations is also given in Figure 13(a) (the "No Div. Opt." bars). Overall, for the thread re-convergence techniques, the Min-PC approach outperforms the ICS-First by only 0.8% in terms of execution time. This result shows that the ICS-First approach can achieve the similar performance to Min-PC, but requires much simpler implementation.

When using the ICS-First approach, the proposed SIMT machine can save an average of 17% execution time compared to that without any divergence optimization. Especially, the divergent optimization significantly improves performance for divergence-intensive kernels. For example, ICS-First achieves 8.3× and 3.1× speedups compared with not using any divergent optimizations for *Blackscholes* and *LavaMD*, respectively. This is because *Blackscholes* and *LavaMD* use *log* and *exp* functions, which are implemented in a math library and cause various divergent executions. As math library is widely used in general-purpose computing, an effective re-convergence mechanism is necessary when using the SIMT execution model.

On the other hand, the ICS-First approach working with the divergence mitigation mechanisms, SLP and DPE, and ALT are evaluated and shown in Figure 13(b). Figure 13(b) shows that the ICS-First mechanism can achieve good performance without the divergence mitigation supports on the target homogeneous multi-core processor. Except the divergence-intensive kernels, for instance, *Blackscholes*, the ALT model in general does not improve the performance. The major reason is the use of the predicated instructions that greatly reduce the divergences. On the average, only about 8% of the dynamic instructions are executed in the MIMD mode in the ALT execution model for the benchmarks in Table 4.

SLP also improves divergence-intensive kernel, Blackscholes, significantly, and has a few positive performance gains for the other kernels. Moreover, our proposed spatiotemporal SIMT execution model has provided enough temporal threads to amortize the execution latency; as a result, DPE delivers only limited improvement when it focuses on warp sub-division intended to increase the schedulable warp threads. Overall, ALT and SLP only improve ICS-First by 2.6% and 2.5%, respectively, while DPE approach does not improve ICS-First generally.

5.4 Discussions

In this subsection, we first discuss the design issues of the proposed SIMT architecture, including thread privatization memory and SIMD-width selection. Then, the data-parallel executions using different programming models, i.e., OpenCL and OpenMP, are also discussed.

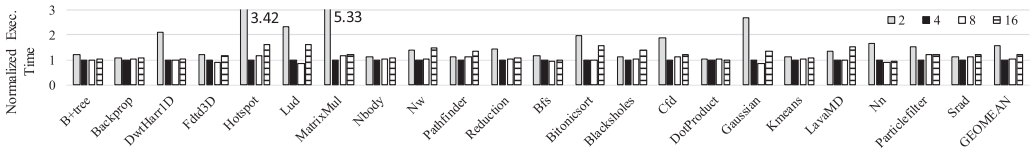


Fig. 14. Comparison of different SIMD widths (Total 32 CPU cores).

5.4.1 Impacts on Memory System. When the processor runs in the SIMT mode, its L1 data cache is used to store the thread-privatization objects or thread contexts rather than the main memory data, which will be cached by the L2 cache. Consequently, the increased memory latency and the data consistency between mode switches are the two raised problems. For the problem of increased memory latency, the number of active threads must be enough to amortize the memory latency for the efficient SIMT execution. We will discuss that how many active threads are enough to amortize the latency for the homogeneous multicore in Section 5.4.2.

For the data consistency problem, since we use the coarse-grained mode selection mentioned in Section 5.2, the mode-switching only occurs at the beginning and the finishing of a kernel execution. That is, there is no mode switching during the kernel execution. When switching to the SIMT mode, the processor cores have to flush their L1 data caches to ensure that the return states of the MIMD mode will not be destroyed by the SIMT operation. On the contrary, when the processor cores switch back to the MIMD mode after finishing the kernel execution, they only need to invalidate the L1 data caches without flushing, since the stored thread contexts will not be used again. Thus, the processor cores can return from the SIMT mode to the MIMD mode seamlessly.

5.4.2 Comparison of Difference SIMD Widths. For a particular work-group size, the number of active threads in a processor core depends on the SIMD width, i.e., the number of the cluster cores, referring to Table 1. Given a total of 32 processors in a system, we configure the SIMD width, to 2, 4, 8, or 16 to construct 16, 8, 4, or 2 SIMT clusters, respectively. The processor configurations are the same as those listed in Table 2. To focus on the SIMD width impact, for all of the cluster configurations, the intra-cluster cache coherence is handled in a snooping-based method, although the 8-core or the 16-core clusters may have unaffordable cost for the snooping-based implementation. The inter-cluster coherence is achieved by a directory-based manner.

Figure 14 shows the execution time normalized to the time for SIMD width of four. Overall, the result shows that an SIMD width of four performs better than other configurations, on average. Since a cluster is assigned a work-group once and the work-items in a work-group are evenly distributed to the cluster processors, a wider SIMD width will lead to fewer active threads assigned to a processor core. However, the SIMT architecture needs a large number of active threads to amortize the execution latency (especially memory operations). As a consequence, SIMT widths of 8 or 16 tend to suffer more easily from long latency instructions. In addition, the cluster cores rely on cache coherence for the work-group share memory consistency. Thus, too many processor cores in a cluster increase the frequency of cache coherency and then lengthen the memory operation latency. This is also the reason why an SIMD width of 32 is not considered here.

The SIMD width of 2 has the most active threads in a processor core to amortize operational latency, and thus, each active thread is allocated with the least number of TPOs. As a result, it is easier for the stack objects of an active thread to exhaust the TPOs, and resort to the storage in the main memory. For example, since the *Hotspot* and *MatrixMul* employ a lot of stack objects for their local variables, a two-lane machine takes 2.42× and 4.33× more execution time, respectively, compared to a four-lane machine. Note that these benchmarks often specify 128-sized or 256-sized

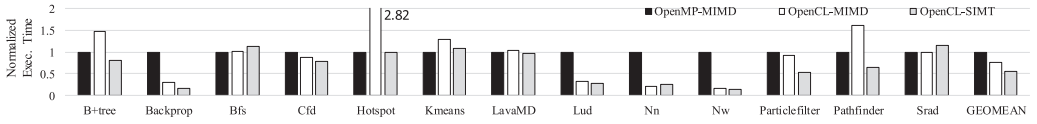


Fig. 15. Comparison of data-parallel executions based on OpenCL and OpenMP programming models (Normalization base is the execution time of OpenMP applications).

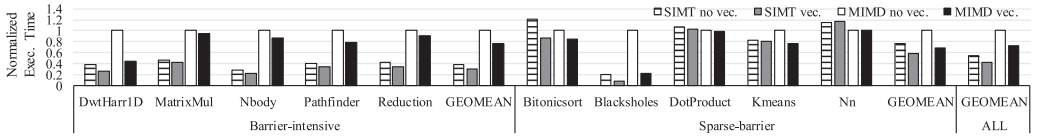


Fig. 16. Comparison of whether using the vectorization optimization or not.

work-groups that result in 32 or 64 active threads per processor for a four-lane machine. The kernel execution of these two active thread sizes amortizes the instruction execution latency well and effectively places the thread contexts in the L1 data caches. Consequently, better performance can be achieved in the given system configuration.

5.4.3 Comparison of OpenCL and OpenMP Programming Models. OpenCL and OpenMP are the two popular and primary data-parallel programming models used in multicore platforms. In general, OpenCL is a fine-grained parallelism model where a work-item is executed for each point inside the problem space. In contrast, OpenMP model employs a coarse-grained parallelism, which splits data-parallel workloads, such as for-loop iterations, into several thread executions [35]. The different parallelization mechanisms actually affect the execution order of the parallel workloads as well as the memory access sequences, and, consequently, reflect on execution performance.

Figure 15 shows the execution time comparisons of the OpenCL and OpenMP applications. The used applications are from Rodinia and the configuration of the multi-core system is shown in Table 2. Note that the applications in AMD and NVIDIA SDKs do not support the OpenMP version. For OpenMP application executions, a thread is used to execute the split data-parallel workloads and run on a processor core in the MIMD mode. On the other hand, for OpenCL, the processor uses either the SIMT or the MIMD executions.

In general, for the evaluated benchmarks, the applications implemented in OpenCL can achieve similar performance or even outperform the OpenMP. The reason is that these applications favor the fine-grained parallelism where their memory access sequences achieve more efficient cache utilization than the OpenMP implementations. As a result, running the OpenCL applications in MIMD mode and SIMT mode can respectively save 25% and 45% execution time compared to the OpenMP applications. Thus, the result shows that our dual-mode architecture is a high-performance design for data-parallel executions on the homogeneous multi-core system.

5.5 Comparison of Kernel Executions on Advanced Processor Architectures

In this subsection, we first discuss the impacts of the vectorization optimization. And then, we will show the comparison of running the data-parallel kernels on multiple superscalar architectures mentioned in Section 5.1.

5.5.1 Comparison of Vectorization Kernel Executions. Figure 16 shows the normalized execution time of running the vectorization and the non-vectorization kernels on either the SIMT mode or the MIMD mode. The normalized base is the execution time of running the non-vectorization

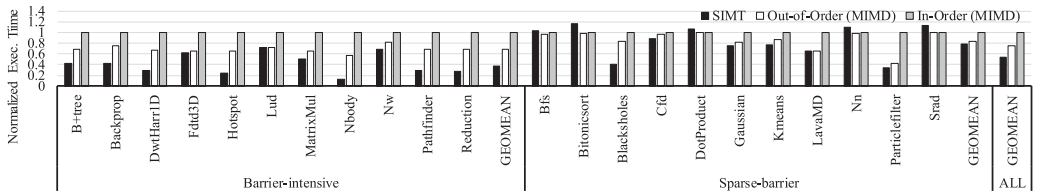


Fig. 17. Comparison of superscalar architectures.

kernels by the MIMD mode. Overall, by using explicit vectorization optimization, the SIMT and the MIMD executions are respectively improved by 21% and 39% over the non-vectorization kernels. The *DotProduct* and *Nn* kernels are not enhanced by vectorization, since they spend most of the execution time for memory accesses.

The vectorization *BlackScholes* kernel significantly improves the SIMT and MIMD executions by 2.5× and 4.4× speedups, respectively. This is because the kernel is compute-intensive and, hence, is efficient for using the vectorization optimization. Furthermore, its MIMD execution has remarkable enhancement, since the vectorization kernel merges sequential tasks of option calculations into a work-item and this significantly improves the cache efficiency for the MIMD operation as we have mentioned in Section 5.2. In summary, for the ten evaluated applications, the SIMT model gains further improvement from the SIMD extension and outperforms the MIMD execution by an average of 1.71× speedup. The result indicates that even the SIMD extension is helpful in running the data-parallel programs effectively, the SIMT model still has opportunity to exploit more data-level parallelism for the higher throughput.

5.5.2 Comparison of Superscalar Architectures. Figure 17 shows the normalized execution time of the three modeled superscalar architectures, including in-order superscalar processor, out-of-order superscalar processor, and the proposed dual-mode superscalar processor. The detailed system configurations are listed in Table 2 and Table 3. The in-order and the out-of-order processors run the kernels by the MIMD approach while the dual-mode processor executes the kernels in the SIMT way. The normalized base is the kernel execution time of using the in-order processor.

For the barrier-intensive applications, as the MIMD executions have software approach overheads, using the SIMT execution model on the superscalar processor can achieve 2.70× and 1.84× speedups over the in-order and out-of-order approaches, respectively. For *Fdtd3D* and *Lud*, the out-of-order processor achieves similar performance to the SIMT model. One reason is that these two kernels can be optimized by selective variable replication more effectively than other barrier-intensive kernels as we have mentioned in Section 5.2. Another reason is that the out-of-order processor can exploit ILP effectively for these two kernels.

On the other hand, for the sparse-barrier kernels, *Bfs*, *Bitonicsort*, *DotProduct*, *Nn* and *Srad*, due to the thread initialization overheads, their SIMT executions are outperformed by the MIMD executions, either on the in-order processor or on the out-of-order one. For the kernels, *BlackScholes* and *ParticleFilter*, we have discussed that they favor the SIMT executions due to the inefficient data locality for the MIMD operation. Compared to the in-order processor, the out-of-order processor shows relatively low performance losses against the SIMT execution model for these two kernels. The reason is that the out-of-order processor can hide the long memory latency as much as possible by executing the later independent instructions in advance.

Overall, by using the out-of-order approach, the superscalar processor can exploit ILP effectively for the data-parallel programs and then outperforms that using the in-order design by 1.32× speedup on the average. On the other hand, enabling the SIMT model on the superscalar processor

can not only exploit data-level parallelism for parallel executions but also eliminate the software approach overheads and thus can achieve on average 1.85× and 1.40× speedups over the in-order and the out-of-order designs, respectively.

6 CONCLUSIONS

This article has addressed architecture issues in enabling the SIMT execution model on a homogeneous multi-core processor. We present three effective approaches for an SIMT/MIMD dual-mode processor, including spatiotemporal SIMT execution, kernel thread context placement, and early thread re-convergence mechanism. We implement these dual mode processor features in an ARM-ISA multi-core platform based on the OpenCL and LLVM compiler frameworks.

To enable the SIMT operations, we develop an SIMT co-processor that schedules the kernel threads in terms of spatiotemporal warps. In the SIMT mode, the processor cores use their respective L1 data caches for the placement of the kernel thread contexts, including ISA registers and stack objects. We introduce an addressing scheme to access these kernel thread contexts. This SIMT execution model saves on average 36% dynamic instructions and boosts the data-parallel kernel executions by 1.52× over the MIMD approach.

For the thread divergence problem, we proposed an Inner Conditional Statement First (ICS-First) algorithm, which guarantees thread re-convergence at the outer statement as soon as the inner thread finishes. This ICS-First mechanism significantly improves the divergent kernel executions and saves 17% execution time over that without using any thread re-convergence mechanism on the average. In addition, we show that the SIMD width has a significant impact on SIMT efficiency. Given a 32KB L1 data cache, the performance favors 32–64 active threads per processor core.

Our study also shows that the execution mode efficiency is mainly affected by the kernel features in terms of variable replication, synchronization frequency, and data locality. Executing a kernel in the most favorable mode can be performed through kernel feature analysis at the compile time. This can achieve on average 1.60× and 1.05× speedups over the MIMD-always and SIMT-always executions, respectively. Moreover, for the processor supporting the SIMD extension, the SIMT model can further achieve an average of 1.71× speedup over the MIMD approach by using the explicit vectorization optimization on kernels.

Nonetheless, the SIMT model can be integrated into the superscalar processor and achieve on average 1.85× and 1.40× speedups compared to the in-order and the out-of-order designs, respectively. Therefore, this work has demonstrated that the proposed SIMT/MIMD dual-mode processor architecture is an attractive feature for future multicore systems.

REFERENCES

[1] Advanced Micro Devices Inc. 2017. AMD Accelerated Parallel Processing SDK. Retrieved from http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/.

[2] Cavium. 2017. Cavium ThunderX ARM Processors. Retrieved from http://www.cavium.com/ThunderX_ARM_Processors.html.

[3] En-Hao Chang, Chen-Chieh Wang, Chien-Te Liu, Kuan-Chung Chen, and Chung-Ho Chen. 2014. Virtualization technology for TCP/IP offload engine. *IEEE Trans. Cloud Comput.* 2, 2 (Apr. 2014), 117–129.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE Computer Society, Los Alamitos, CA, 44–54.

[5] Kuan-Chung Chen and Chung-Ho Chen. 2014. An openCL runtime system for a heterogeneous many-core virtual platform. In *Proceedings of the 2014 IEEE International Symposium on Circuits and Systems (ISCAS'14)*. 2197–2200.

[6] Yuan Chi. 2016. *OpenCL Kernel Attribute Prediction for Operation Mode Se-lection in SIMT/MIMD Dual-mode Architecture*. Master's thesis. National Cheng Kung University, Taiwan.

[7] Sylvain Collange. 2011. *Stack-less SIMT Reconvergence at Low Cost*. Technical Report.

[8] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. 2011. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 477–488.

[9] Wilson W. L. Fung and Tor M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE Computer Society, Los Alamitos, CA, 25–36.

[10] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2009. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Trans. Archit. Code Optim.* 6, 2, Article 7 (Jul. 2009), 37 pages.

[11] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. 2010. Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, NY, USA, 205–216.

[12] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th ed.). Elsevier.

[13] Yun-Chi Huang, Kuan-Chieh Hsu, Wan-Shan Hsieh, Chen-Chieh Wang, Chia-Han Lu, and Chung-Ho Chen. 2016. Dynamic SIMD re-convergence with paired-path comparison. In *Proceedings of the 2016 IEEE International Symposium on Circuits and Systems (ISCAS'16)*. 233–236.

[14] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. pocl: A performance-portable openCL implementation. *Int. J. Parallel Program.* 43, 5 (Oct. 2015), 752–785.

[15] James Jeffers and James Reinders. 2013. *Intel Xeon Phi Coprocessor High Performance Programming* (1st ed.). Morgan Kaufmann, San Francisco, CA.

[16] Gangwon Jo, Won Jong Jeon, Wookeun Jung, Gordon Taft, and Jaejin Lee. 2014. OpenCL framework for ARM processors with NEON support. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing (WPMVP'14)*. ACM, New York, NY, 33–40.

[17] Khronos Group. 2011. The OpenCL Specification 1.2. Retrieved from https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

[18] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. 2012. SnuCL: An openCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, NY, 341–352.

[19] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, 2004 (CGO'04)*. 75–86.

[20] Jun Lee, Jungwon Kim, Junghyun Kim, Sangmin Seo, and Jaejin Lee. 2011. An openCL framework for homogeneous manycores with no hardware cache coherence. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE Computer Society, Washington, DC, USA, 56–67.

[21] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jungho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi. 2010. An openCL framework for heterogeneous multicores with local memory. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, NY, 193–204.

[22] Dong Li, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, and Stephen W. Redder. 2015. Priority-based cache allocation in throughput processors. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 89–100.

[23] Yuxi Liu, Zhibin Yu, Lieven Eeckhout, Vijay Janapa Reddi, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Chengzhong Xu. 2016. Barrier-aware warp scheduling for throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*. ACM, New York, NY, Article 42, 12 pages.

[24] Jan Lucas, Michael Andersch, Mauricio Alvarez-Mesa, and Ben Juurlink. 2015. Spatiotemporal SIMT and scalarization for improving GPU efficiency. *ACM Trans. Archit. Code Optim.* 12, 3, Article 32 (Sept. 2015), 26 pages.

[25] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. 2002. Hyper-threading technology architecture and microarchitecture. *Intel Technol. J.* 6, 1 (2002), 1–12.

[26] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 235–246.

[27] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, 308–317.

[28] John Nickolls and William J. Dally. 2010. The GPU computing era. *IEEE Micro* 30, 2 (Mar. 2010), 56–69.

[29] NVIDIA Corporation. 2012. NVIDIA CUDA Toolkit 4.2. Retrieved from https://developer.nvidia.com/cuda-toolkit-42-archive.

[30] Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. 2015. gem5-gpu: A heterogeneous CPU-GPU simulator. *IEEE Comput. Arch. Lett.* 14, 1 (Jan. 2015), 34–36.

[31] Minsoo Rhu and Mattan Erez. 2013. The dual-path execution model for efficient GPU control flow. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE Computer Society, Los Alamitos, CA, 591–602.

[32] Minsoo Rhu and Mattan Erez. 2013. Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 356–367.

[33] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. 2015. A variable warp size architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 489–501.

[34] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Los Alamitos, CA, 72–83.

[35] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. 2012. Performance gaps between openMP and openCL for multi-core CPUs. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*. 116–125.

[36] Milan Stanic, Oscar Palomar, Timothy Hayes, Ivan Ratkovic, Adrian Cristal, Osman Unsal, and Mateo Valero. 2017. An integrated vector-scalar design on an in-order ARM core. *ACM Trans. Archit. Code Optim.* 14, 2, Article 17 (May 2017), 26 pages.

[37] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. 2010. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. ACM, New York, NY, 111–119.

[38] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. 2008. *MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs*. Vol. 5335. Springer, Berlin, 16–30.

[39] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. ACM, New York, NY, 392–403.

[40] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 335–344.

[41] Ali Vahidsafa, Sebastian Turullols, David Smentek, Ram Sivaramakrishnan, Paul Loewenstein, Sumti Jairath, and John Feehrer. 2013. The oracle sparc T5 16-core processor scales to eight sockets. *IEEE Micro* 33, 2 (2013), 48–57.

[42] Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy, and Mani Azimi. 2013. SIMD divergence optimization through intra-warp compaction. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 368–379.

[43] Yaohua Wang, Shuming Chen, Jianghua Wan, Jiayuan Meng, Kai Zhang, Wei Liu, and Xi Ning. 2013. A multiple SIMD, multiple data (MSMD) architecture: Parallel execution of dynamic and static SIMD fragments. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE Computer Society, Los Alamitos, CA, 603–614.

[44] Y. Wen, Z. Wang, and M. F. P. O'Boyle. 2014. Smart multi-task scheduling for openCL programs on CPU/GPU heterogeneous platforms. In *Proceedings of the 2014 21st International Conference on High Performance Computing (HiPC'14)*. 1–10.

[45] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. 2007. On-chip interconnection architecture of the tile processor. *IEEE Micro* 27, 5 (Sept. 2007), 15–31.

Received June 2017; revised October 2017; accepted December 2017