GPU Acceleration for Simulating Massively Parallel Many-core Platforms

Shivani Raghav, Martino Ruggiero, Andrea Marongiu, Christian Pinto, David Atienza, Luca Benini

Abstract—Emerging massively parallel architectures such as a general-purpose processor plus many-core programmable accelerators are creating an increasing demand for novel methods to perform their architectural simulation. Most state-of-the-art simulation technologies are exceedingly slow and the need to model full system many-core architectures adds further to the complexity issues.
This paper presents a novel methodology to accelerate the simulation of many-core coprocessors using GPU platforms. We demonstrate the challenges, feasibility and benefits of our idea to use heterogeneous system (CPU and GPU) to simulate future architecture of many-core heterogeneous platforms. The target architecture selected to evaluate our methodology consists of an ARM general purpose CPU coupled with many-core coprocessor with thousands of simple in-order cores connected in a tile network. This work presents optimization techniques used to parallelize the simulation specifically for acceleration on GPUs. We partition the full system simulation between CPU and GPU, where the target general purpose CPU is simulated on the host CPU, whereas the many-core coprocessor is simulated on the NVIDIA Tesla 2070 GPU platform. Our experiments show performance of up to 50 MIPS when simulating the entire heterogeneous chip, and high scalability with increasing cores on coprocessor.

Index Terms—Parallel Simulation, Heterogeneous Architectures, Many-core Processors, Accelerators, GPGPU, CUDA, QEMU

1 Introduction

With increasing complexity and performance demands of emerging applications, heterogeneous platforms are becoming a popular trend in computer design. Increased use of embarrassingly parallel algorithms and fine-grained parallelism is creating a market for general-purpose hardware accelerators (coprocessors) to manipulate large amounts of data in parallel with high energy efficiency [1]. These future platforms consist of traditional multi-core CPUs in combination with a many-core coprocessor, which is composed of thousands of embedded cores. Examples of these heterogeneous architectures include on-chip specialized many-core coprocessors [4][5][6] and upcoming tile-based many-core architectures [7][8].

Simulating these heterogeneous architectures poses novel challenges, as current state-of-the art simulation technologies are not sufficiently well equipped to handle their complexity. Simulation platforms are needed to make meaningful predictions of design alternatives and early software development, as well as to be able to assess the performance of a system before the real hardware is available. Current state-of-the-art sequential simulators leverage SystemC [11], binary translation [17], smart sampling techniques [12] or tunable abstraction levels for hardware description. However, one of the major limiting factors in utilizing current simulation methodologies is simulation speed. Most of the existing simulation techniques are slow and/or have poor scalability, which leads to an unacceptable performance when simulating a large number of cores. Since next generation many-core coprocessors are expected to have thousands of cores, there is a great need to have simulation frameworks that can handle target workloads with large datasets, while is also suitable for parallel simulation of many-core architectures. In order to comprehensively and quickly evaluate the design, architecture and programming tradeoffs in such future heterogeneous platforms, a fast simulation method with scalability up to thousands of cores is a fundamental requirement. In addition, these new and scalable simulation solutions must be inexpensive, easily available, with a fast development cycle and able to provide good trade-offs between speed and accuracy.

It is easy to notice that simulating a parallel system is an inherently parallel task. This is because individual processor simulation may independently proceed until the point where communication or synchronization with other processors is required. This is the key idea behind parallel simulation technologies in which we distribute the simulation workload over parallel hardware resources. Parallel simulators have been proposed in the past [13][14][15], which leverage the availability of multiple physical processing nodes to increase the simulation rate. However, this requirement may turn out to be too costly if server clusters or computing farms are adopted as a target to run the many-core coprocessor simulations.

The development of computer technology has recently led to an unprecedented performance increase of General-Purpose Graphical Processing Units (GPGPU).

- *S. Raghav, M. Ruggiero and D. Atienza are with Embedded Systems Laboratory, EPFL, Lausanne, CH. (e-mail: shivani.raghav, martino.ruggiero, david.atienza@epfl.ch).*
- *A. Marongiu, C. Pinto and L. Benini are with DEI, University of Bologna, Italy. (email: a.marongiu, christian.pinto, luca.benini@unibo.it)*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems

Modern GPGPUs integrate hundreds of processors on the same device, communicating through low-latency and high bandwidth on-chip networks and memory hierarchies. This allows us to reduce inter-processor communication costs by orders of magnitude with respect to server clusters. Moreover, scalable computation power and flexibility is delivered at a rather low cost by commodity GPU hardware. Besides hardware performance improvement, the programmability of GPUs also has been significantly increased in the last five years [16][18]. This has led to the proliferation of computing clusters based on such many-cores, providing an inexpensive solutions in high performance computing domain for a wide community.

This scenario motivated our idea of developing a novel parallel simulation technology that leverages the computational power of widely-available and low-cost GPUs. In this simulation method we exploit the opportunity to parallelize the simulation of many-core coprocessor on top of GPGPU host platforms. The main novelty of our simulation methodology is to use heterogeneous system as a host platform to tackle the challenge of simulating the heterogeneous architectures of future heterogeneous platforms.

While exploring the idea of simulation acceleration using GPUs, we encountered several performance and implementation challenges. One of the main challenge is to identify the parallelization in simulator code and optimize it for the scalability on GPU. In this paper, we present the key challenges for simulation scalabilities and methods used to tackle these challenge. Specifically, we make following contributions:

- We present a comprehensive approach to build full system simulation frameworks for heterogeneous architectures. The architecture of coprocessor is inspired from the GPUs [6][9] and accelerator chips[7][8], which are most likely to scale to thousands of cores in near future. As a case study, we selected a target architecture which is composed of a general purpose CPU connected with a coprocessor with thousands of cores in a tile network. The proposed architecture is selected as an illustration for future coprocessor architectures to present the viability and benefits of using our approach.
- We present our main idea to partition heterogeneous system simulation workloads between CPU and GPU.
- We discuss the code optimization techniques that we utilize to maximize concurrency and gain maximum benefit from parallel GPU hardware.
- We provide a cycle-approximate model for fast performance prediction of the overall simulated platform. The simulation framework uses a *relaxed-synchronization* technique to minimizing synchronization overhead and achieving speedup.

For experimental purposes, we simulated an ARM CPU connected with ARM-based coprocessor composed of up to 4096 cores. The targeted architecture of RISC cores connected with a tile network is a popular candidate for future development in the area of many-core coprocessors. Although this work presents the experiences related to a particular target architecture accelerated on GPUs, the methods applied and lessons learned are more broadly applicable for future many-core designs.

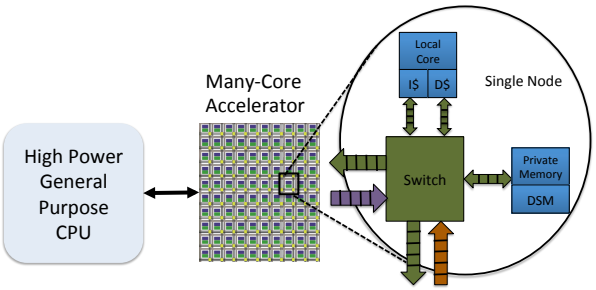


Fig. 1: System Target Architecture

Our experimental results demonstrate the benefits of using our proposed simulation method, with which we achieved up to 50 MIPS when simulating the complete CPU-Coprocessor system and high scalability compared to other state-of-the-art simulation approaches.

2 Overview of Full System Heterogeneous Simulation Approach

In this section, we provide architectural details of the heterogeneous platform targeted by our simulation technique. Next, we give an overview of our simulation flow for the full system simulator.

2.1 Target Architecture

Our target architecture is representative of future heterogeneous platforms. It consists of a a general purpose processor connected with a many-core coprocessor (accelerator), as shown in Figure 1. While currently available many-core coprocessors only integrate up to hundreds of cores interconnected via a network-on-chip (NoC)[8] [7] [10] , in the near future the number of cores is likely to increase to the thousands [21]. The simulator presented in this work is targeted to model such future embodiments of many-core paradigm. To simulate general-purpose CPU, we selected QEMU[17] which is an ARM Versatile Platform Baseboard featuring an ARM v7-based processor and input-output devices. QEMU is a popular, open-source, fast emulation platform based on dynamic binary translation which models a complete development board with a set of common devices (e.g. Ethernet interfaces, Disks, Audio controllers), enabling the execution of an un-modified operating system allowing applications compiled for an architecture to be run on many others.

The target coprocessor features many (thousands of) simple ARM cores each equipped with data and instructions scratchpad memories (SPM), private caches, private

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems

and distributed shared memory of target (TDSM). The architecture of a core in the coprocessor is based on a simple single issue, in-order pipeline. The cores are interconnected via an on-chip network organized as a rectangular mesh. As shown in Figure 1, a single node includes a core, its cache subsystem, NoC switch, private memory per core and a bank of physically distributed shared memory (TDSM). Caches are private to each core, which means that they only deal with data or instructions allocated in the private memory. The distributed shared memory (TDSM) is non-cacheable; therefore the simulation of the cache coherence protocol is not required.

The applications and program binaries targeted for this system are launched from within Linux OS running on the general-purpose CPU. The execution of a parallel application is divided between the two entities, the host and the coprocessor. General purpose processor runs the sequential part of target application up to a point where a computation intensive and highly-parallel program region is encountered. When a parallel program region is encountered, this particular part of the program is offloaded to the coprocessor to gain benefit from its high performance.

The considered memory model of coprocessor adheres to the Partitioned Global Address Space paradigm (PGAS)[19]. Each thread (mapped to a single node of target coprocessor) has private memory for local data items and shared memory for globally shared data values. Both private and shared memory is mapped in a common single global address space, which is logically partitioned among a number of threads. Each thread has a private space as well as affinity with globally shared address space. Greater performance is achieved when a thread accesses data, which is held locally (whether in its private memory or a partition of the global address space). Non-local access to the shared memory space generate communication traffic across on-chip interconnect, therefore incur performance overhead. Programming model assumed for this target architecture is similar to Unified Parallel C [20]. It distributes the independent iterations across threads typically to boost locality exploitation. Interaction between threads is managed by synchronization primitives on shared data items. Data qualified as shared resides in shared memory space while rest of the data is considered thread private data.

2.2 Simulation Flow

Figure 2 depicts the full-system simulation methodology we propose to model heterogeneous architectures. Our simulator consists of two main blocks. First, QEMU emulates the target general purpose processor, capable of executing a Linux OS and file system. Next, our coprocessor simulator uses GPUs for accelerating simulation of its thousands of cores.

Our coprocessor simulator is entirely written using C for CUDA [16] and, in order to model thousands

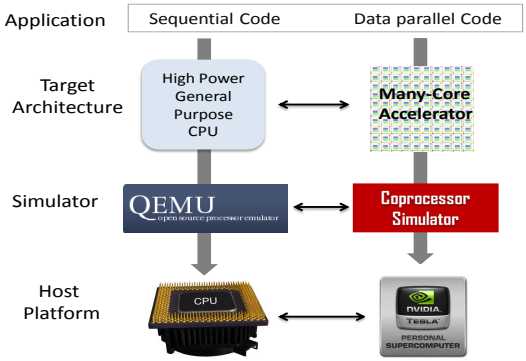


Fig. 2: Overview of Simulation Flow

of nodes, we map each instance of a simulated node to a single CUDA thread. These CUDA threads are then mapped to streaming processors of GPU by its hardware scheduler and run concurrently in parallel. Each target core model is written using an interpretation-based method to simulate the ARM pipeline. Thus, each simulated core is assigned its own context structure, which represents register file, status flags, program counter, etc. The necessary support for data structures are initially allocated from the main (CPU) memory for all the simulated cores. The host program (running on the CPU) initializes these structures, and then copies them to the GPU global device memory, along with the program binary. Once the main simulation kernel is offloaded to the GPU, each simulated core repeatedly fetches, decodes and executes instructions from the program binary. Similar to the operation on the hardware, the instruction byte is fetched, decoded and executed at run time. Each core updates its simulated registers file and program counter until program completion.

3 Interfacing QEMU with Coprocessor Simulator

As mentioned in Section 2, full system simulation is partitioned between CPU (using QEMU) and GPU (using coprocessor simulator), therefore it is essential to find an efficient way to offload the data parallel part of the target application from QEMU on to coprocessor simulator.

As shown in Figure 3, the target application running on guest kernel space of the target platform (simulated by QEMU) is named QEMU target process. QEMU process running on host CPU is named QEMU-Hprocess. Our coprocessor simulator program written in C and CUDA is designed to execute partly on the host CPU and host GPU. The part of coprocessor simulator program executed on host CPU is named CP-Hprocess. The other part that runs on host GPU platform is named CP-Gprocess. Target application needs to forward requests (data structure and parameters) between QEMU and the coprocessor simulator almost instantly in parallel. Therefore an interface is needed so that the QEMU target process can communicate to the QEMU-Hprocess and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems

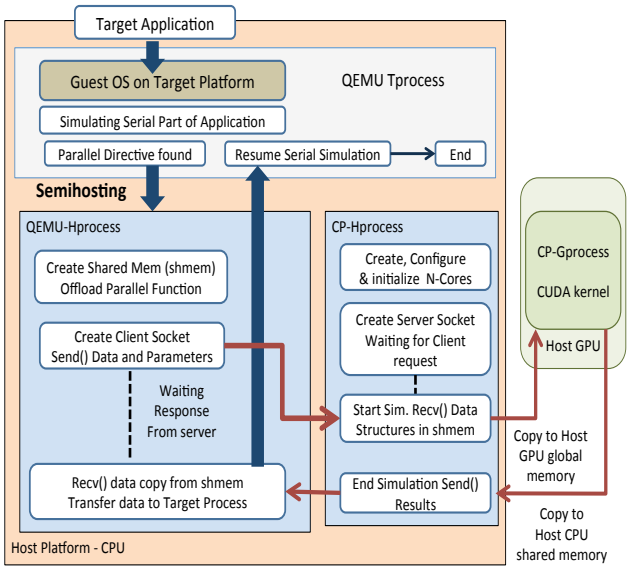


Fig. 3: Overview of our technique to interface QEMU and the Coprocessor-Simulator

finally to the CP-Gprocess. To implement this communication between the processes, we use *semihosting* [24](see Figure3).

Semihosting is a technique developed for ARM targets allowing the communication between an application running on the target and a host computer running a debugger. It enables the redirection of all the application's IO system calls to a debugger interface. We leverage this method to provide a direct communication channel between QEMU target process (QEMU-Tprocess) and QEMU-Hprocess. Next, we used Unix Domain Sockets to transfer data between the QEMU-Hprocess and the CP-Gprocess. The CP-Hprocess initially boots as a server and waits for an offloading request from the client. When the client (QEMU-Hprocess) encounters a data parallel function from the QEMU-Tprocess, it transfers the structure of parameters pointed by semihosting call to the server socket. When the CP-Hprocess receives all the necessary data structures (data and code segment), it launches the CUDA kernel as a CP-Gprocess for simulation on GPU platform. The QEMU-Hprocess waits for computation to end from GPU side and releases it in the end. To avoid overheads due to moving big amounts of data using the socket, larger data structures (e.g., I/O buffers) are accessed through host processor shared memory (HSM) segments defined and allocated by the QEMU-Hprocess. For more details on the implementation of this method, please refer to our previous work [25].

4 Functional Simulation of Coprocessor on GPU

The coprocessor simulator comprises many modules that simulate the various components of the target architecture. In particular, the core model is responsible for modeling the computational pipeline; the memory model includes scratchpad memory models, cache models for private memory and distributed shared memory of the target (TDSM) models. The network model handles TDSM operations and allows routing of network packets over the on-chip network.

The entire simulation flow is structured as a single CUDA kernel, whose simplified structure is depicted in Figure 1. One physical GPU thread is used to simulate one single node. The program is run inside a main loop until all simulated nodes in turn finish their simulation. The core model is executed first. During the instruction fetch phase and while executing memory instructions, the core issues memory requests to the cache model. The cache model is in charge of managing data/instructions stored in the private memory of each core. Caches are private to each core and therefore they only deal with data/instructions allocated in the private memory. Communication between the cache model, the core and the NoC model takes place using communication buffers allocated in shared memory region of GPU device (GSM). Our proposed information exchange mechanism exploits the producer/consumer paradigm without the need for synchronization because core, cache and NoC models are executed sequentially and communication buffers are used to exchange information between core, cache and network model.

Since the TDSM in our target architecture is distributed across the nodes and is non-cacheable, DSM regions are only accessible by sending packets through on-chip network. When an operation towards the shared address space is detected, the request is forwarded to the corresponding tile using mesh based network. For details on network-on-chip simulation model, please refer to the supplemental material.

4.1 Key Challenges for Simulation Scalability

Although CUDA provides a powerful API with a swift learning curve, designing a parallel simulator for many-core running on GPUs is not a straightforward task and implementing a simulator for such a platform imposes several challenges as listed below:

- One of the main challenge handled in simulation of target nodes is to identify the parallelization in various stages of target node pipeline and map the target nodes on CUDA threads such that there is minimum control flow divergence.
- The layout of data structure that represent the context of target node is carefully organized in GPU memory to utilize high bandwidth GPU global memory and low-latency shared memory (GSM).
- We optimize the simulation code to ensure GPU device shared memory (GSM) is free of bank conflicts.
- Interaction between CPU and GPU is a costly process. Therefore, we minimize the amount of data transfer required between host CPU and GPU platform by using *relaxed-synchronization* method as presented in the following section.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems

The methods implemented to overcome abovementioned limitation are provided with supplemental material. For more details on implementation of each of these models, please refer to our previous publications [26][27].

5 Performance Prediction Model of Target Coprocessor

A performance model gives an insight about the application primitives that are particularly costly in a certain heterogeneous platform and allows us to predict the runtime behavior of a program in that target platform. Moreover, thread level parallelism creates timing dependent outcomes; therefore in addition to having functional correctness, it is important to have timing fidelity as well. In this paper, we present a cycle-approximate method to predict the performance of our many core coprocessor. This allows designers to make decisions about configurations of architectural parameters and to predict the scalability of their applications. The simulator only calculates the performance prediction for the part of the program running on the coprocessor. Accurate performance prediction for the whole heterogeneous simulator is beyond the scope of this work.

Cycle-Approximate Method - In this section, we perform fast performance estimation of the simulated platforms at runtime by annotating the events generated by a functionally accurate model with a fixed estimated delay. Ideally, system-level simulation should provide sufficient timing details for performance evaluation with cycle-accuracy. However due to extremely slow simulation of cycle-accurate models, it is realistic to say that for large scale many-core platforms, timing accuracy at the micro-architecture level is not a prime requirement. Predictions about features such as application scalability, costs due to memory locality and high synchronization rates are sufficient for users to perform early design space exploration. Our main goal is to achieve significant simulation speedup useful for architecture performance estimation at early design stages [28]. Therefore, we develop a simplified cycle-approximate model to estimate the operation latency of devices simulated by the functional simulator.

In order to quantitatively estimate perfomance of an application running on target many-core coprocessor, we apply simple fixed, approximated delays to represent the timing behaviors of simulation models. Since the functional model of cores is based on interpretation scheme, it is easier to tightly couple the timing information with each generated event. Every core has a local clock variable and this variable is updated after completion of each instruction (event). Considering a unit-delay model all computational instructions have fixed latency of a single clock cycle. Caches have 1-cycle hit latency and 10-cycle cache miss penalty. Each simulated node thread simulates a local clock which provides a timestamp to every locally generated event and clock cycles are updated for each event with round trip latency. As the simulator executes an application code on a core, it increases its local clock in accordance with the event executed on the code block. Each memory access or remote request is initially stamped with the initiator core's local clock time and is increased by a specific delay as it traverses the architecture's communication model. When the initiator core finally starts processing the reply, its own local clock is updated to that of the reply. To summarize, the sum of all delays induced by all the device models traversed is added to a core's local time in case of interaction. Similarly, when the response of a memory load instruction returns to its originating node thread, the local clock of this thread simulating the local clock of the node is updated by the round trip latency gathered by the memory packet. Communication packets through on-chip network update their latency as they traverse through the system and thus collect the delay due to congestion and network contention. Single-hop traversal cost on on-chip network is assumed to be one. Finally, the simulator output consists of global clock cycles of many-core processor as well as total instructions per cycles calculated for running the parallel application allowing the designers to forecast scalability and performance variations from routing and network contention as well as coarse-grain architecture changes.

6 Synchronization in Many Core Simulation

When simulating a many core coprocessor, synchronization requirements can add significant overhead to the performance efficiency. This section focuses on our efforts to increase the efficiency of required synchronization operations.

6.1 Synchronization Requirements in Parallel Many Nodes Simulation

Application programs offloaded to the coprocessor, may contain inter-thread interactions using various synchronization primitives such as barrier and locks. In this case, application threads running on many cores will generate accesses to the shared memory distributed across the simulated many nodes (TDSM), which in turn will result in traffic (remote packets) over the NoC. We call these remote packets *s-events*. To simulate these *s-events*, it is important to have a synchronization mechanism to ensure the timing and functional fidelity of the many core simulation maintaining both simulation speed and accuracy.

Timing Fidelity - Cycle approximate timing simulation assesses the target performance by modeling the node latencies using local clock cycles. Since this is simulated using parallel host GPU hardware threads, node clocks are non-synchronized and run independent of each other at different simulation speeds. To accurately model the timing behavior of the entire system, these simulated local clocks should be synchronized at some

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems

point in time to keep the accounting of global target time of coprocessor. Additionally, during the occurrence of *s-events*, it is important that target nodes clock cycles proceed in lock-step manner at every clock cycle by creating synchronization points after each clock tick of all nodes. This is essential to determine the correct round trip latency of NoC packet communication between otherwise unsynchronized nodes. For example, when application threads are mapped on different cores of simulated coprocessor and during an *s-event* such as spin lock for a shared memory (TDSM) variable, the local clocks of each node needs to know the correct number of iterations that each thread should wait before acquiring the lock.

Functional fidelity From the functional accuracy point of view, synchronization between many nodes is essential to maintain functional simulation accuracy particularly during simulation of *s-events*. Since only *s-events* modify the state of the shared memories of the target system, they needs to be simulated in non-decreasing timestamp order so that shared memory (TDSM) accesses with data dependencies are executed in order. *S-events* with smaller timestamp have potential to modify the state of the system and thereby affect events that happen later.

To illustrate this, let us consider an example where the first *s-event* s1 of the system is detected by node n1, at its local target timestamp T1. If another node n2 has a local timestamp T2, such that T2 >T1 then the simulation of s1 will not create any data dependency violation with respect to n2 because n2 is most likely executing local memory instructions and it is safe for the simulation to proceed. However, if another node n3 is at an earlier timestamp T3 such that T3 <T1, then a potential dependency violation may occur if n3 generates an *s-event* s3 and s1 and s3 have data dependency. In this case node n1 should wait for until timestamp of both n1 and n3 are synchronized such that T3 is at least equal to T1 and has finished simulating all previous s-events.

Therefore, a remote *s-event* arriving at a node should not have timestamp lower than local events and a node should only handle an *s-event* when it can be sure that no remote *s-event* with earlier timestamp will arrive in the future.

6.2 Challenges of Synchronization on GPU

Due to lack of support for inter-block synchronization in CUDA, supporting simulation of *s-events* is a challenging task. From the point of view of implementing synchronization behavior on GPU, this requires periodically synchronizing hardware threads on a barrier. Barrier functionality can only be imposed on threads within the same block and synchronization among thread blocks is not natively supported by CUDA and GPU hardware. This is achieved by terminating the GPU kernel function call and using costly communication between CPU and GPU (Host-CPU-Barrier). Host-CPU-Barrier requires suspension of execution on GPU, saving the states of all simulated cores in GPU's global memory and transferring control to host CPU where barrier synchronization is performed. This poses a serious performance, bottleneck and slows down the entire simulation.

Xiao and Feng [48] recently proposed on-GPU synchronization, which facilitates inter- block synchronization without returning to the CPU. However, special care must be taken because in a naive implementation we may easily experience deadlocks when simulating a higher number of cores than the available physical GPU processors. Simulated nodes are mapped on both active and inactive thread-blocks and the GPU hardware scheduler selects thread-blocks for execution based on available computational resources [16]. If the number of threads in all blocks is higher than the number of processors on the GPU, only a subset of all the blocks can execute, while the remaining blocks wait (inactive) until the first set finishes its execution.

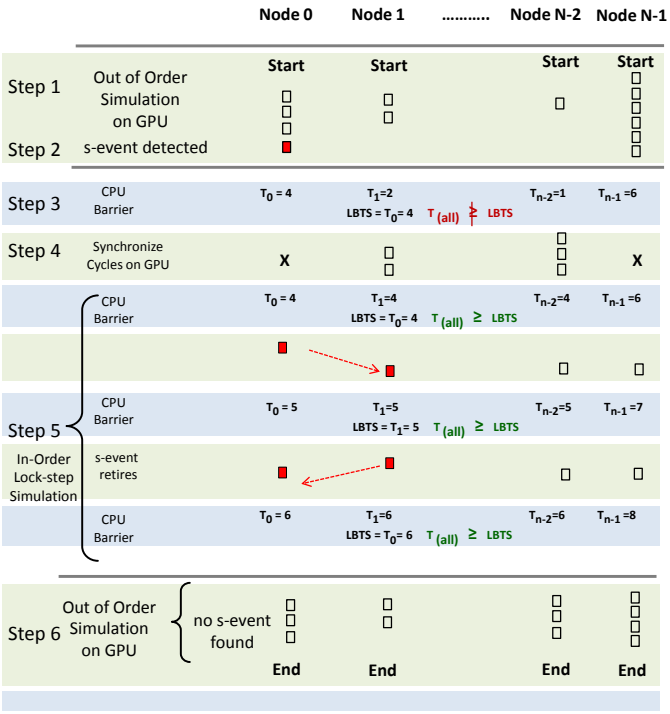


Fig. 4: Graphical Representation of using *Relaxed-Synchronization* to minimize performance overhead from Host-CPU-Barrier on GPU Platforms

6.3 *Relaxed-Synchronization*

A traditional lock-step approach or cycle accurate simulation would force synchronization of each node simulation at every clock cycle, so that the simulated program can execute in synchronized target time. The major drawback of this approach is immense synchronization overhead, as the simulation would stop and synchronize at each clock-tick.

To address the problem of high overhead due to Host-CPU-Barrier synchronization and gain significant

speedup, we use less frequent synchronization, introducing a technique which we call *relaxed-synchronization*. In *relaxed-synchronization* , instead of synchronizing at every few clock cycles, we choose to synchronize only upon *s-events* to minimize the number of synchronization points. Figure 4 shows the steps taken to achieve fast synchronization on GPU as described below:

- **Step 1** - Simulation of all nodes are allowed to run freely and cores simulate with high speed until an *s-event* is detected in coprocessor system. This is to ensure that we achieve fast simulation by allowing the hardware context thread to execute independently without the need to synchronize frequently. All nodes which are simulated independently, continuously checks for the presence of any *s-event* in the system after execution of each local instruction by polling an *s-event* flag in GPU device memory. When any of the nodes in the system detects the first occurrence of an *s-event*, it sets the *s-event* flag and all other nodes are notified of the presence of an *s-event* in the system. In Figure 4 node 0 is the first to detect the presence of an *s-event* at its local clock cycle T0 reads 4. As explained above, at this point it is necessary to synchronize all nodes to maintain the functional and timing fidelity of the simulation.
- **Step 2** - As explained in Subsection 6.1, to avoid data dependency violation we need to make sure that all the nodes should have local clock cycles at least equal to the timestamp of the node generating the *s-event*. Local clocks Tn of all nodes are collected and Lower Bound on Timestamps (LBTS) is calculated, where LBTS is equal to the clock cycle of the node that notifies the presence of an *s-event*.
- **Step 3** - Calculating LBTS requires inter-thread synchronization of GPU CUDA threads. Therefore all nodes suspend their independent simulation and control returns to the host CPU platform to perform Host-CPU-Barrier operation. As shown in Figure 4, the LBTS is calculated as 4 in this case.
- **Step 4** - When GPU kernel is launched again, if it was detected in the previous step that some of the nodes have Tn less than LBTS, then a synchronization cycle is called which ensures that the simulation of all nodes has a timestamp greater than or equal to LBTS. Simulation of nodes with local clock lower than LBTS proceed while nodes with timestamp greater than or equal to the lower bound wait until all local clocks are at least equal to LBTS =4 as shown in figure. If there are any previous *s-events* present in the system, Step 2,3 and 4 are repeated until all of them are detected and LBTS is set to the minimum value of their timestamp.
- **Step 5** - The simulation proceeds in lock-step fashion until all detected *s-events* in the system are retired as shown in Figure 4 . As explained in Step 1, due to the presence of *s-events* in the system, each node simulates a single event on GPU before returning the control to the CPU to perform the Host-CPU-Barrier. Lock-step simulation also helps to ensure that timing fidelity is maintained and we get the expected round trip latency of the packet traveling across NoC by continuously checking any modified value of LBTS after each GPU kernel launch.
- **Step 6** - Once the simulation of *s-events* in the system is complete, normal simulation resumes without barriers until the simulation ends or next *s-event* is encountered at which point Step 2 is invoked again. In Figure 4 *s-event* generated in Step 1 is serviced and fast GPU simulation continues.

Next, with the help of Figure 5 , we explain lock step simulation implemented in NoC simulation. We recall that a single CUDA thread simulates a single node of the NoC, which has a network switch connected to local and neighboring queues as shown in Figure 1. Local processor and memory queues are bidirectional and switch receives requests and inserts the forwarding packets from/to them independently. However neighboring nodes are connected using two queues - incoming and outgoing. The incoming queue for one switch of the neighbor acts as an outgoing queue for the second one and vice versa. Within one single kernel launch (step) of lock-step synchronization, every switch queries its incoming packet queues, then selects a single packet from one of its queues and forwards it to the destination outgoing queue. This implies that in a single step, two neighboring switches may be reading and writing from the same packet queue. Therefore additional synchronization is needed where one of the neighbors is trying to insert the packet in the outgoing queue while another one is trying to read the packet from the same queue location (which may result in write-after-read hazard).

This synchronization is done using a combination of CPU barrier synchronization and a lock-free fast barrier synchronization [48] (see Figure 5). Therefore, a single step between two CPU Barrier Synchronization points is further divided into a read and writes cycles. First, in a read cycle all switches poll their incoming queue for requests and read a packet to be serviced. This is followed by a lock-free fast barrier synchronization[48] to ensure that all threads have finished reading their incoming queues, before writing the selected packet into their outgoing queues. Finally, the following CPU barrier synchronization ensures that packets written to all queues in the write cycle are globally visible for reading with the next CUDA kernel launch.

The overall simulation performance loss due to synchronization is related to the number of *s-events* present in each simulated workload. Since our target coprocessor is aimed at running many-thread data parallel workloads, we expect very low synchronization requirements and consequently high simulation speedups for heterogeneous platforms.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems

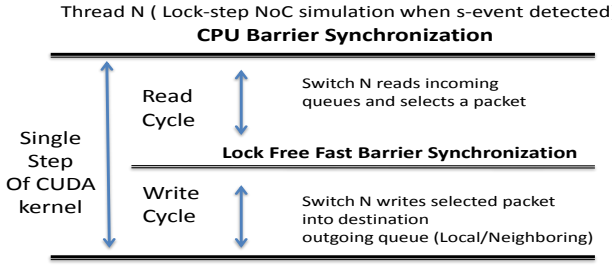


Fig. 5: Synchronization in a single step of NoC simulation under Lock-step Simulation

7 Experimental Results

In this section we first present our experimental set up and benchmarks. Next we show performance results for benchmarks running on top of our simulator. Finally, we show the evaluation of scalability of our simulator and a detailed comparison of its performance with respect to other state-of-the-art commercial platform simulation approaches.

7.1 Experimental Setup

For evaluating our simulation methodology, we carefully selected our target architecture and benchmarks for simulation. As a target architecture for heterogeneous platforms, we decided to simulate an ARM versatile baseboard with dual core ARM1136 CPU connected with a many-core programmable coprocessor. As described before in Section 2, the architecture we are targeting for target coprocessor has thousands of very simple in-order cores. We use simple RISC32 cores (based on ARM instruction set architecture [24]). Since we model various different system components in our coprocessor simulator (i.e. Cores, Caches, On-chip network, Memory), it is important to understand the cost of modeling each component on performance of simulator. Therefore we conducted our experiments with two different architectures.

- *Architecture I* - First we considered an architecture where each tile is composed of a RISC32 core with associated instruction (ISPM) and data scratchpad memory (DSPM). All private memory references are handled by dedicated code portion modeling the behavior of scratchpad memories. All synchronization instructions targeted towards shared memory (TDSM) are handled from a global space.
- *Architecture II* - This includes the entire set of components such as cores, caches, NoC, SPMs, distributed shared memory (TDSM) and performance models. Caches are private to each core, which means that they only deal with data or instructions allocated in the private memory. The distributed shared memory (TDSM) is non-cacheable; therefore simulation of cache coherence protocol is not required. The complete list of architectural features is provided in Table 1.

We characterize our simulator's performance using a metric called S-MIPS. S-MIPS presents the rate at which the host platform simulates target instructions. We define S-MIPS as follows:

$$\text{S-MIPS} = \frac{\text{Millions of Simulated Instructions}}{\text{Host wall clock time in seconds}}$$

For all experiments, we used a NVIDIA C2070 Tesla graphic card (the *Device*), equipped with 6 GB memory and 448 CUDA cores. The QEMU ARM emulator runs a Linux kernel image compiled for the Versatile platform with EABI support. The many-core coprocessor simulator executes the data parallel section of the workload offloaded by the main program, simulated on QEMU. As a host CPU platform, we used an Intel Xeon 2.67 GHz multi-core system running Linux 2.6.32. To generate target binaries for ARM, we have used arm-linux-gcc. We vary the number of processors in the simulated many-core from 128 to 4096. This allows us to explore the performance of our simulator when modeling both current and future many-core designs.

7.2 Benchmarks

The benchmarks we have selected aim at evaluating i) the scalability of target workload on many-cores, ii) the design alternatives for target architecture with CPU-coprocessor scheme iii) the efficacy of simulator implementation on GPU. We measure the impact on four most important factors:

- Data level parallelism
- Dataset sizes
- Synchronization
- Task level parallelism

In Table 2, we list the benchmarks adopted for our experiments with their dataset size. As mentioned in Section 2, the coprocessor architecture enables fine-grained parallelism and is best suited for workloads with high level of data-level parallelism. Therefore, the first four benchmarks are extracted from a JPEG decoder and from the OpenMP Source Code Repository [29] benchmark suite and exhibits high degree of data parallelism. We also used the EP kernel from NAS parallel benchmark [31]. The EPCC benchmark is taken from the well-known OpenMP Microbenchmarks Suite [30], which contains large data parallel phases interspersed with several implicit synchronizations. Figure 6 presents the instruction profile showing percentage of different mix of instructions in each of these benchmarks when 4096 cores are simulated on the many-core coprocessor. The percentage of instructions is referred to the application portion that is running on the target coprocessor. We can see that for EPCC benchmark, the fraction of synchronization instructions represents a small percentage of the total. The dataset size changes between benchmarks. In particular, MM and NCC have larger datasets, which implies a longer duration of their overall execution time compared to the other benchmarks.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems

Processing Cores	128 to 4096 ARM ISA cores, 3-stage pipeline
Caches	Instruction and Data Caches per core, 32 KB per node, Reconfigurable set-associative (default 8 ways), FIFO replacement policy, write allocate, write no allocate, write back, 128-byte lines Cache miss latency: 10 cycles, Cache hit latency: 1 cycle
Interconnect	Network-on-chip, nxn 2D mesh, static routing, XY, Single cycle per hop when no network congestion
Private Memory	4GB per target coprocessor, Distributed equally for available number of cores, Cacheable
Shared Memory (TDSM)	40MB per target coprocessor, Distributed equally for available number of cores, Non-cacheable

TABLE 1: System parameters of target architecture of coprocessor

	Benchmark	Acronym	Source	Comments
1	*Inverse DCT*	IDCT	JPEG Decoding [29]	Datasize - 4K DCT blocks(8*8 pixels)
2	*Luminance Dequantization*	DQ	JPEG Decoding [29]	Datasize - 4K DCT blocks(8*8 pixels)
3	*Background Subtraction*	NCC	Normalized Cut Clustering [29]	Datasize - 4K parallel rows
4	*Matrix Multiplication*	MM	OpenMP Source Code Repository[29]	Datasize - (4096x100)*(100x100)
5	*Fast Fourier Transform*	FFT	OpenMP Source Code Repository[29]	(Datasize = 4K)
6	*EPCC Test Atomic*	EPCC	EPCC OpenMP Microbenchmark Suite [30]	Datasize = 4K (Atomic Operations)
7	*Embarrasingly Parallel*	EP	NAS Parallel Benchmarks [31]	Datasize= 2^{18}
8	*Barrier - Centralized*	BC	PARKBENCH [32], Marongiu et al.[33]	Datasize = 4096 End barrier (Spin Locks, Polling Loops)
9	*Barrier- Master Slave Distributed*	B-MSD	PARKBENCH [32], Marongiu et al.[33],	Datasize = 4096 End barrier (Spin Locks, Polling Loops)
10	*Barrier - Master Slave Distributed Tree*	BMSD-T	PARKBENCH [32], Marongiu et al.[33],	Datasize = 4096 End barrier (Spin Locks, Polling Loops)

TABLE 2: Benchmarks

FFT was chosen as a representative of task (MIMD) parallelism. Indeed, in this benchmark, threads with an odd ID perform different computation than threads with an even ID. These threads, when mapped to the simulation cores, create control flow divergence, as discussed in Section 4.

Finally, synchronization is an important feature for any shared memory programming model and it is important to measure the overhead of using synchronization primitives in a given workload. Therefore, we selected a worst case scenario and used a barrier synchronization benchmark, described in [32] and [33]. This benchmark consists of a sequence of data-parallel tasks (or algorithmic phases) and a final barrier synchronization, which makes threads wait for each other at the end of parallel region. We consider three different implementations of barrier algorithm to show that our simulator can precisely capture the performance impact of software and architectural design implementations, namely:

- *Centralized* - In the first implementation of the barrier benchmark (BC), a centralized shared barrier is used. It uses shared entry and exit counters atomically updated through lock-protected write operations. Implementation of synchronization primitives is done using spinlocks and polling a shared variable which is stored in one single segment of the distributed shared memory of the target (TDSM). Threads busy-waiting for barrier to complete are constantly sending memory packets with high latency towards a single node. Therefore the number of synchronization instructions increases with the increasing number of simulated cores, creating increasingly high traffic towards a single tile in the network.
- *Distributed Master-Slave* - In this barrier algorithm (B-MSD), we work around the contention problem by designating a master core, responsible for collecting notifications from other cores (the slaves). Each slave notifies its presence on the barrier to the master on a separate location of an array stored in DSM portion local to the master. The master core polls on this array until it receives the notification from all the slaves. Slave cores however poll on a separate DSM portions local to them. When the master core determines that all slaves have reached barrier, it releases the slaves by writing to their polling location [33]. Distributing the polling location for slaves local to their DSM segment, greatly reduces the traffic on network due to busy-waiting, as evident from Figure 6.
- *Tree Based Distributed Master-Slave* - This algorithm is similar to B-MSD, but further improves performance by using a tree based multi-stage synchronization mechanism (BMSD-T) where cores are organized in clusters. The Master-Slave approach is maintained as explained above, and each core in the cluster has dedicated notification and polling flags. In this case, the first core of each subcluster is master to all slave cores in that subcluster. When all

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems

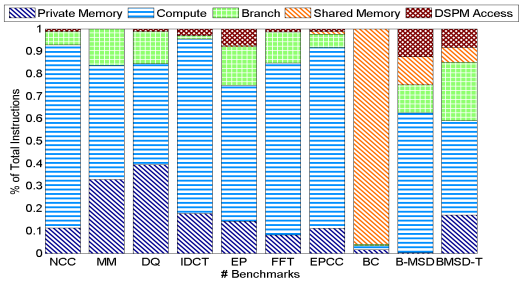


Fig. 6: Instruction Profile of Benchmarks (parallelized for 4096 cores)

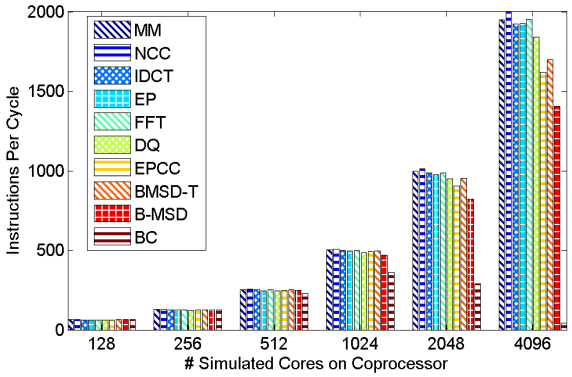


Fig. 7: Instructions Per Cycles

slaves in a subcluster reach the barrier, they trigger top-level synchronization betweeen local (cluster) masters [33]. Since the tree-based implementation (BMSD-T) is better suited for a large number of processors, it is expected to further mitigate the effect of barrier synchronization.

Overall, each of these benchmarks is either representative of a category of applications widely used in the many-core domain, or contains specific computation or memory patterns frequently found in highly parallel applications. All benchmarks are launched from within Linux OS running on QEMU. During the execution of these benchmarks, when a parallel kernel is encountered, it is offloaded for simulation on our many-core simulator using the semihosting technique. The parallelization scheme we have developed for this purpose is similar to OpenMP static loop scheduling and focuses on evenly dividing total loop iterations among all participating processors. More specifically, an identical computation is replicated over parallel threads, which operate on disjoint chunks of the iteration space and dataset according to their identification number.

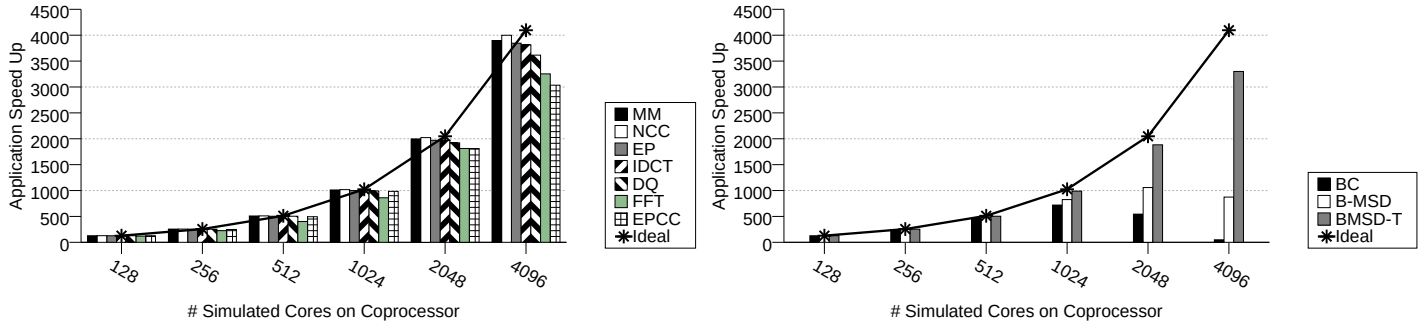
7.3 Application Performance Estimation

In this subsection, we present the results related to the performance prediction capabilities of our many-core coprocessor simulator. Application performance depends upon a large number of parameters. Performance overhead is related to the various software features such as parallel programming model, number of loop iterations per thread, chunk size and software implementation of synchronization directives. In addition to this, it also depends upon architectural features such as cache design, memory locality and communication cost due to network delay. Therefore, with this set of experiments we assess how each of these characteristics affects application performance. We report average instructions per cycles (IPC) for each benchmark in Figure 7. As mentioned in Section 7.2, the first five benchmarks have high data parallellism and therefore show increasing IPC of up to 2000 with increasing number of cores. EP and EPCC benchmark have a small percentage of synchronization instructions, so they still benefit from the rest of the available data parallel section. BC is our worst case scenario and maximum IPC it achieves is around 350, for the case when coprocessor is simulated for 1024 cores. This is due to heavy traffic coming from synchronization directives, which creates a bottleneck in the NoC and imposes sequential execution. The simulator therefore correctly demonstrates the cause of poor performance for BC. The poor performance is due to the ill-matched synchronization scheme of the BC barrier algorithm with an architecture with NUMA (distributed) memory. B-MSD and BMSD-T are more suitable implementations for the assumed memory model and hence show better results compared to BC. As expected, due to multi-stage implementation of Master-Slave algorithm, the simulation cycles are further reduced in BMSD-T as compared to BMSD, and therefore show higher instructions per cycles.

In Figure 9a and 9b, we show scalability results for up to 4096 cores. The application speedup is calculated by dividing the cycle counts of a parallelized benchmark running on N cores by the cycle count of the same benchmark running on a single core of the coprocessor. Results for data parallel kernels are shown in Figure 9a and barrier algorithms are shown in Figure 9b. All the benchmarks achieve good scalability in Figure 9a due to high parallellism. FFT features data-dependent conditional execution and EPCC includes synchronization primitives. Due to this, the inherent parallelism of EPCC and FFT is lower than that of the other benchmarks and consequently we see a slight decrease in application scalability when parallelizing for 4096 cores.

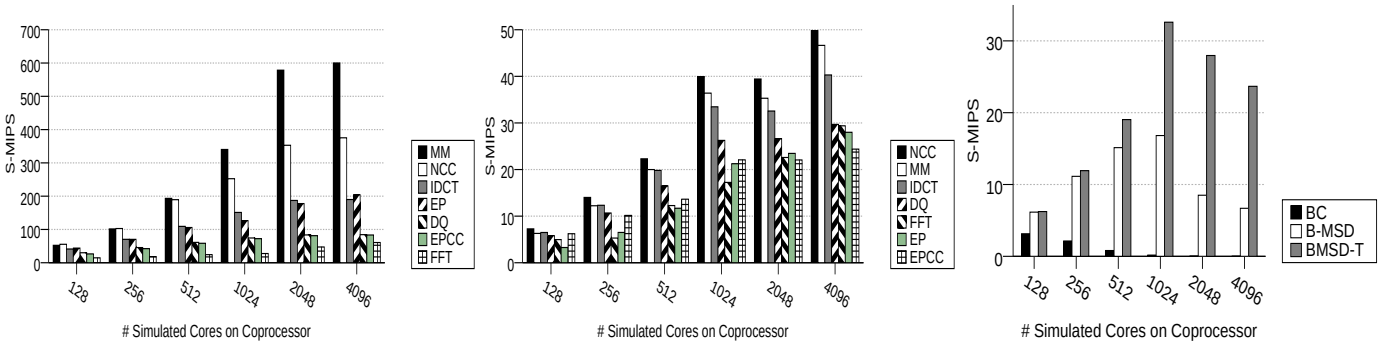
In Figure 9b, as we expect, BC does not scale, showing the effect of heavy network congestion due to contention for centralized barrier counters. B-MSD mitigates the effect of synchronization and shows significantly better results where the application speed up increases to 1000x. Employing a tree-based algorithm in BMSD-T further removes the traffic due to busy-waiting, and therefore shows better speedup (3300x) than BMSD with increasing number of cores.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems



(a) Data Parallel Benchmarks (b) Barrier Benchmarks

Fig. 9: Application Speed Up when target coprocessor simulated with up to 4096 cores



(a) Architecture I (b) Architecture II (c) Architecture II- Barrier Synchronization

Fig. 10: S-MIPS: Simulated Millions of Instructions Per Second with increasing core counts on coprocessor.

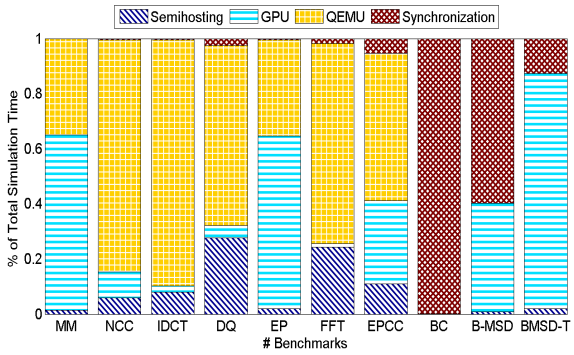


Fig. 8: Percentage of overall simulation time for benchmarks running on 1024 cores

7.4 Simulator Scalability Evaluation

In this section, we evaluate the scalability of our many-core simulator running on GPU. We present the simulator's performance using S-MIPS for both *ArchitectureI* and *Architecture II* as explained in Section 7.1.

Figure 10a shows S-MIPS for increasing core count of the simulated accelerator for *Architecture I*. We can notice that the performance is as high as 600 S-MIPS when simulated for 4K cores for the MM benchmark with biggest dataset sizes. The same experiment for Architecture II is shown in in Figure 10b. The performance is close to 50 S-MIPS. As explained in Section 4, the performance of our simulator is directly related to the level of parallelism available in the coprocessor workload. High level of data parallelism in application implies longer simulation of workload in parallel, thus benefiting from GPU hardware parallelism and therefore results in better scalability and performance. Therefore, MM and NCC benchmarks with the largest parallel datasets benefits the most and show highest performance compared to rest other benchmarks. For other benchmarks, due to the small workload size on coprocessor, the overhead associated with semihosting and parallelization directives wins over the benefits of parallelism on GPU. In addition to this, as explained in Section 4, task parallel workload has a performance impact on architecture simulation due to the serialization of execution when control flow divergence occurs (during the execution phase of core pipeline simulation), which is visible in the simulation performance of the FFT benchmark in both Figure 10a and Figure 10b. In Figure 10b, we see that the increase in S-MIPS stagnate between 1024 and 2048 cores. This happens because when simulating Architecture 2, due to a very large data structure of NoC, Caches etc, we exhaust the available shared memory resource on GPU device (GSM) and GPU scheduler can only launch

a limited number of total CUDA thread blocks per multiprocessor, which can simulate up to 1792 cores concurrently. Simulation of rest of the cores wait until the first batch of simulation finishes before launching the next batch of core simulation. Due to this serialization, we see no gain in performance between 1024 and 2048 cores, however when simulating 4096 cores, we again see increase in performance due to the fact now a very high number of cores are simulating in parallel, although in three batches of 1792 cores simultaneously.

Figure 10c shows our implementation of barrier benchmarks. BC being a worst-case scenario incurs a huge performance overhead due to slow CPU-GPU communication, as explained in Section 4. The impact of synchronization overhead is visible in *Architecture II*, where each application synchronization requires frequent CPU-GPU barrier synchronization to faithfully simulate on-chip network communication and synchronize the cycles counts of the performance model as explained in Section 5. B-MSD and BMSD-T show significant improvement in performance. The cost of synchronization is visible beyond the simulation of 1024 cores, but as we expect, BMSD-T shows the best performance among all three implementations.

Figure 8 gives further insight into our heterogeneous simulator scalability. It shows the breakdown of total simulation time for different benchmarks. We evaluate the amount of time spent on the QEMU, the time spent on many-core simulation, overhead of semihosting required for the communication between the two and time spent in CPU-GPU communication. These results refer to a 1024-core instance of many-core coprocessor. Since the MM benchmark has the largest dataset, the time spent on the GPU is highest, whereas for BC, B-MSD, BMSD-T, most of the time is spent in the CPU-GPU communication (due to synchronization). For most cases, we can see that semihosting time is a small fraction of total execution time.

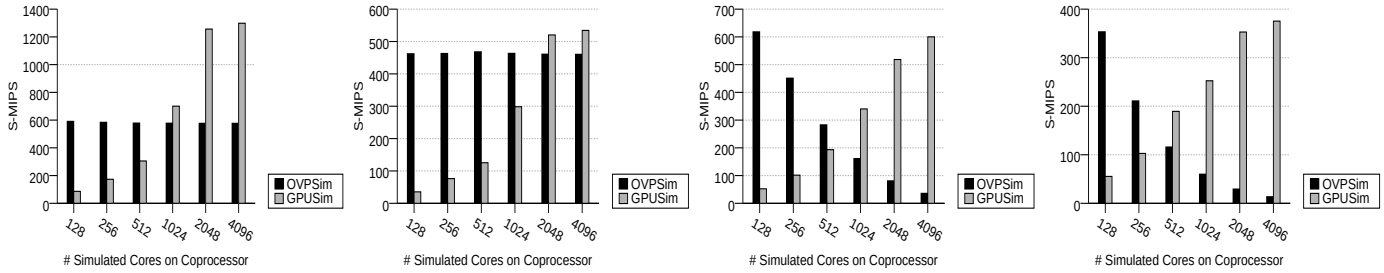
7.5 Simulator Performance Comparison

In this section we compare our simulation methodology with dynamic binary translation (DBT). Single core simulation on a powerful CPU using DBT is likely to outperform our interpretation-based simulation approach on the GPU. However, we can expect that the high number of streaming processors available on a single graphics card would allow our simulation method to deliver significant speedup benefits when simulating thousands of cores on coprocessor.

To the best of our knowledge, none of the currently available simulators can simulate thousands of ARM cores along with caches and interconnect similar to the full system architecture of our target coprocessor. As a term of comparison, we selected OVPSim [34], which is a famous commercial state-of-the-art simulation platform able to model architectures composed of thousands of ARM-based cores. OVPSim is a sequential simulator where each core takes turn after certain number of instructions, however it exploits the benefits of Just in Time Code Morphing and translation caching system to accelerate the simulation. OVPSim is a functionally accurate simulator without support for cache or interconnect modeling. The host platform used for running OVPSim is the same we use for our QEMU-based target CPU simulator; an Intel i7 quad-core x86-64 based machine, running Linux at 2.67 GHz We compare the performance of OVPSim against *Architecture I* (Section 7.1), which most closely matches what OVPSim is capable of modeling. We conducted two different experiments. First we consider two benchmarks from the OVPSim test suite, Dhrystone and Fibonacci. Unlike our other benchmarks, these two benchmarks are not parallelized and every core on the coprocessor simulator executes the benchmarks entirely. The main reason for the using this set of benchmarks is to highlight the reason behind the steady throughput (S-MIPS), exhibited by OVPSim as shown in Figure 11a, 11b. They also allow us to present the difference between the OVPSim technique that uses code morphing technology as compared to our interpretation based method. In both benchmarks, OVPSim shows a constant performance with increasing number of simulated core because of its code morphing technology. With these benchmarks, OVPSim needs to invoke its morphing phase just once and exploits the translation caching system to speed up the simulation. Our GPU based simulator, on the other hand, scales well up to 2048 simulated. Beyond 2048 cores the achievable throughput only increases very slightly. Due to per-block GPU device shared memory (GSM) requirements on the GPU, we are only able to run at most 3 blocks per multiprocessor at a time. When simulating 4096 cores we exceed this limit and extra blocks are dynamically scheduled thus impacting the final scalability. The breakeven performance point between our coprocessor simulator and OVPSim is 1024 cores.

In the second experiment, we consider our data-parallel benchmarks MM and NCC. We recall here that, with this parallelization approach, smaller chunks of data are processed by each core when the core count increases. Results for this test are shown in Figure 11c and 11d.These graphs show that performance of OVPSim decreases significantly, showing less than 50 S-MIPS when simulating 4096 cores. Indeed, OVPSim suffers from a high initial overhead, induced by its code morphing phase. This overhead is increasingly evident as the workload size diminishes, since morphing time tends to dominate. This initial overhead is clearly amortized as soon as workload increases. On the contrary, our simulator performs equally well in this context (600 MIPS), even for very small workloads. On-chip many-core coprocessors are often involved in data-parallel computation, which may contain even very small amounts of work (e.g., embedded accelerators for image processing, which may perform single-pixel computation). In these scenarios our simulation approach

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2014.2319092, IEEE Transactions on Parallel and Distributed Systems



(a) Dhrystone (b) Fibonacci (c) Matrix Multiplication (d) NCC

Fig. 11: Performance Comparison of GPU based Coprocessor Simulator with OVPSim when simulating up to 4096 cores.

performs better than OVP. The breakeven performance point between coprocessor simulator and OVPSim for data-parallel kernels is close to 512 cores.

Results presented in this section indicate that our simulation approach shows a very high performance and scalability capability for the target many-thread workloads and many-core architectures. Thousands of hardware thread contexts available in GPU host make a perfect match for simulation of the simple, single issue, in-order cores of our target many-core coprocessor. The performance of our simulator is however dependent upon the total number of cores simulated in coprocessor simulator as well as on the target workload. It is easy to notice that high performance gain is obtained when we simulate very high number of cores. We also proved that our simulator performance scales further with the increasing scalability level shown by the workload being simulated in the target platform. Although there is a small impact of synchronization and task parallelism on the performance of the simulator, but the probability of their presence in our target workloads is expected to be very low. With future development of GPU architectures, we aim to incorporate dynamic binary translation technique in our simulator to achieve further improvement in performance.

8 Related Work

Researchers and computer architects have proposed different approaches in the recent years to address the challenges of simulating large-scale SoC systems.

First, there are **functional-only simulators** or virtual platforms/emulators such as [17][35][34]. Although they show good performance for single core simulation, when simulating many-core they do not provide good scalability and due to sequential nature, their simulation time increases exponentially. In order to predict performance, there are several **performance simulators** that use fast emulators model in addition with their own timing models [12] [37]. Each estimation approach can be evaluated on the basis of speed, accuracy and tunable abstraction level. **Transaction level models** and examples of using high abstraction level include System C based software performance estimation techniques [38][39]. They provides significant speed up but their many-core simulation capability is still limited to only hundreds of cores.

Parallel Simulation solutions are proposed in the past include [13][14][15]. BigSim[14] is parallel multithreaded emulator that mimics low level hardware and message passing primitive to facilitate the execution of parallel applications. Graphite [15] is a multicore simulator that model thousands of cores by parallelizing the simulator on multiple networked computers as opposed to our inexpensive and easily available GPU based platform. None of these simulation tool mentioned above target heterogeneous architectures with a thousand-core coprocessor.

In the past, **FPGA and GPU-accelerated simulation** have been proposed for many-core system emulation to assist the application development process for multicore processors ProtoFlex[41] RAMP[42] HASim[49] and BeeFarm[50]. Such techniques utilize the concurrency of hardware such as FPGA to directly imitate the internal design of the target system. Even though hardware emulation solutions provide good performance, they are still limited to protyping a few hunderds of cores and a software GPU-based solution provides better flexibility and scalability. Moreover the GPU based solution is cheaper and more accessible to a wider community. Recently, a few research solutions have been proposed to run gate-level simulations on GPUs [43] [44] and a cache simulator [45] on a CUDA GPU target.

In the domain of **Simulating Heterogeneous Architectures**, a few simulators targeting GPU cores have been proposed. Barra[46] and Ocelot [47] are functional simulators bound around specific architecture of NVIDIA. The major limitations faced by all of these solutions is a poor simulator performance. None of these simulator use the parallelization available in GPU host platform. To the best of our knowledge, our simulator is the first one to use GPU based platforms for parallel simulations of many-core heterogeneous architecture.

9 Conclusion

In this paper, we have presented a novel methodology to use GPU acceleration for architectural simulation of heterogeneous platforms in which a general purpose processor is coupled with a many-core coprocessor . The main motivation of this work is to present feasibility, optimization techniques, and performance benefits gained from accelerating simulation on GPUs. We have shown in this work how to effectively partition simulation workload between the host machine's CPU and GPU. Thousands of hardware thread contexts available in GPU hardware make a perfect match for simulation of the simple, single issue, in-order cores of our target many-core coprocessor.

In order to experimentally assess the effectiveness of our approach, we selected an illustrative target architecture of an ARM CPU connected with a coprocessor featuring up to 4096 simple cores. The methods and approach presented in this work are also applicable for similar future target architectures. In our future work, we aim to target architecture of commercially available GPUs and accelerator chips with thousands of cores. Compared to performance results from the OVPSim simulator, our solution demonstrates better scalability when simulating a target platform with increasing number of cores. More precisely, our proposed simulator achieved up to 50 MIPS when simulating full architecture for coprocessor including cache and NoC model while 600 MIPS when a simpler architecture is considered with thousands of cores using scratchpad memory. We also proved that our simulator performance scales further with the increasing scalability level shown by the workload being simulated in the target platform. Moreover, we presented application performance and profiling information of different benchmarks, which shows the utility of the proposed simulation approach.

References

[1] Bader, A. et al., Guest Editors Introduction: Special Issue on High-Performance Computing with Accelerators, IEEE Transactions on Parallel and Distributed Systems, Vol. 22, No. 1, January 2011.

[2] Yang, X. et al., The TianHe-1A Supercomputer: Its Hardware and Software in the Journal of Computer Sci. & Tech., Vol 26, 344-351

[3] ARM-GPU Hybrid Supercomputer.http ://www.montblanc-project.eu/.

[4] ClearSpeed Whitepaper: CSX Processor Architecture. http://www.clearspeed.com/.

[5] Plurality Software Emulator for Multi-Cores, http ://www.plurality.com.

[6] NVIDIA's Tegra http ://www.nvidia.com/object/tegra-2.html.

[7] Bell, S. et al. "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect",Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International , vol., no., pp.88-598, 3-7 Feb. 2008

[8] Howard, J. et al., A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International, pages 108 -109, feb. 2010.

[9] http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx

[10] http://www.kalray.eu/products/mppa-manycore/mppa-256

[11] The open SystemC initiative. http ://www.systemc.org.

[12] Argollo, E. et al., COTSon: Infrastructure for Full System Simulation. in Operating Systems Review, Vol 43, Num 1,2009

[13] Penry, D. et al., Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors,in HPCA, 2006.

[14] Zheng, G. et al., "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," IPDPS'04

[15] Miller, J. et al., Graphite : A distributed parallel simulator for multicores. In High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on, pages 1 -12, jan. 2010.

[16] / NVIDIA CUDA Programming Guide. http://developer.download.nvidia.com

[17] QEMU, "http://wiki.qemu.org."

[18] AMD. ATI Stream Computing OpenCL Programming Guide. http://developer.amd.com

[19] V. Saraswat, et al. The asynchronous partitioned global address space model. Technical report, June 2010.

[20] W. W. Carlson, et al. Introduction to UPC and Language Specification, 1999

[21] Borker S. Thousand core chips: a technology perspective In DAC '07: Proceedings of the 44th DAC (2007), pp. 746-749

[22] The open standard for parallel programming of heterogeneous systems http://www.khronos.org/opencl/

[23] NVIDIA CUDA Best Practices Guide, version 3.2. http ://developer.download.nvidia.com.

[24] ARM architecture - http://infocenter.arm.com

[25] Raghav S. et al. Full System Simulation of Many-Core Heterogeneous SoCs using GPU and QEMU Semihosting. Fifth Workshop on GGPGU , Held with ASPLOS XVII, London, 2012.

[26] C. Pinto. et al.,, "GPGPU-Accelerated Parallel and Fast Simulation of Thousand-Core Platforms," Cluster, Cloud and Grid Computing (CCGrid), 2011, vol., no., pp.53-62, 23-26 May 2011

[27] S. Raghav. et al.,, Scalable Instruction Set Simulator for Thousand-core Architectures Running on GPGPUs, HPCS 2010, Caen, France.

[28] J. Bammi et al., Software performance estimation strategies in a system-level design tool. CODES, 2000

[29] Dorta, A. et al., The OpenMP source code repository. In Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pages 244250, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Bull,J., Measuring Synchronisation and Scheduling Overheads in OpenMP, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, Sept.1999, pp. 199-105

[31] H. Jin et al., The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Advanced Supercomputing Division, Oct. 1999

[32] R. Hockney et al. P A R K B E N C H Report - 1: Public International Benchmarks for Parallel Computers , Scientific Programming, 3 (2) , 1994, pp. 101-146.

[33] Marongiu A. et al., Supporting OpenMP on a multi-cluster embedded MPSoC, Microprocessors and Microsystems, Volume 35, Issue 8, November 2011, Pages 668-682.

[34] The Open Virtual Platforms (OVP) portal. http ://www.ovpworld.org/.

[35] P. S. Magnusson et al., "Simics: A full system simulation platform, Computer, vol. 35, no. 2, pp. 5058, 2002.

[36] SimpleScalar D. Burger et al, The SimpleScalar Tool Set, Tech. report CSTR-97-1342, Computer Sciences Dept., Univ. of Wisconsin, 1997

[37] Yourst, M.T. "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," ISPASS 2007.

[38] Ruggiero M. et al., Scalability Analysis of Evolving SoC Interconnect Protocols, Proceedings of The 2004 International Symposium on System-on-Chip, Finland, Nov 16-18, 2004, pp. 169-172.

[39] L. Formaggio et al. A Timing-Accurate HW/SW Co-simulation of an ISS with SystemC. In the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, Sep. 2004.

[40] Trevor E. et al., Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations, International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Nov, 2011.

[41] Eric S. Chung et al., ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs, ACM Transactions on Reconfigurable Technology and Systems (TRETS), v.2 n.2, p.1-32.

[42] J. Wawrzynek et al. RAMP: Research accelerator for multiple processors. IEEE Micro, 27(2):4657, Mar. 2007

[43] D. Chatterjee et al., Event-driven gate-level simulation with gp-gpus, in Design Automation Conference, 2009. DAC 09. 46th ACM/IEEE, july 2009, pp. 557 562.

[44] K. Gulati et al., Towards acceleration of fault simulation using graphics processing units, in Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE, june 2008, pp. 822 827.

[45] W. Han, et al., Using gpu to accelerate cache simulation, in Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on, aug. 2009, pp. 565-570.

[46] Collange, S. et al. Barra: A Parallel Functional Simulator for GPGPU, Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on , vol., no., pp.351-360, Aug. 2010

[47] Diamos G. et al., Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10). ACM, New York.

[48] Shucai X. et al., Inter-block GPU communication via fast barrier synchronization. IPDPS, 19-23 April 2010

[49] Pellauer M. et al. Hasim: Fpga-based highdetail multicore simulation using time-division multiplexing. In HPCA (February 2011)

[50] Sonmez, N., et al. From Plasma to BeeFarm: Design Experience of an FPGA-Based Multicore Prototype. In 7th Symposium on ARC, March 2011.