Full System Simulation of Many-Core Heterogeneous SoCs using GPU and QEMU Semihosting

Blind Review

ABSTRACT

Modern system-on-chips are evolving towards complex and heterogeneous platforms with general purpose processors coupled with massively parallel manycore accelerator fabrics (e.g. embedded GPUs). Platform developers are looking for efficient full-system simulators capable of simulating complex applications, middleware and operating systems on these heterogeneous targets. Unfortunately current virtual platforms are not able to tackle the complexity and heterogeneity of state-of-the-art SoCs. Software emulators, such as the open-source QEMU project, cope quite well in terms of simulation speed and functional accuracy with homogeneous coarse-grained multi-cores. The main contribution of this paper is the introduction of a novel virtual prototyping technique which exploits the heterogeneous accelerators available in commodity PCs to tackle the heterogeneity challenge in full-SoC system simulation. In a nutshell, our approach makes it possible to partition simulation between the host CPU and GPU. More specifically, QEMU runs on the host CPU and the simulation of manycore accelerators is offloaded, through semi-hosting, to the host GPU. Our experimental results confirm the flexibility and efficiency of our enhanced QEMU environment.

Keywords

Parallel Simulation, GPGPU, Heterogeneous Platforms, Many Cores, GPU, CUDA

1. INTRODUCTION

Fueled by the continued shrinking of feature sizes in silicon manufacturing, an increasing number of functions can be implemented on a Systems-On-Chip (SoC). Contemporary silicon technology makes it possible to integrate complex SoC offering high architectural heterogeneity which provides tremendous amount of computational power, low cost, low power consumption and high flexibility.

Due to different functional characteristics and application requirements, modern SoC architectures consist of a combination of different processor types and dedicated hardware accelerators. Since chip manufacturers are focusing on reducing the power consumption and on packing of an ever-increasing processing unit number per chip, the trend towards simplifying the micro-architecture design of cores will be increasingly strong: manycore accelerators will be embedding thousands of simple cores in the future. The typical platform is composed of general purpose processors coupled with massively parallel manycore accelerator fabrics (e.g. embedded GPUs [1][6][7]). Examples of similar architectures may include on-chip many-core accelerators such as the Hypercore Architecture Line (HAL) from Plurality [10], ST Microelectronics Platform 2012 [24], or future evolutions of Intel prototypes Larrabee [3] and Single-Chip Cloud Computer[16].

The main challenge in the design of such complex heterogeneous SoCs is the reliable and efficient system programming to allow a safe integration of the different system parts. Platform architects and developers need full-system simulators capable of executing complex applications, middleware and operating systems on these heterogeneous targets. It is clear that current virtual platform technologies are not able to tackle the possible issues incurred due to high complexity of simulating this future scenario, mainly because they suffer from either poor performance or low accuracy. Therefore, in order to comprehensively evaluate the design, architecture and programming tradeoffs in such future heterogeneous SoC, there is a clear need for having a parallel, fast and accurate full system simulator.

The development of computer technology has recently led to an unprecedented performance increase of General-Purpose Graphics Processing Units (GPGPU). Modern GPGPUs integrate hundreds of processors on the same device, communicating through low-latency and high bandwidth on-chip networks and memory hierarchies. Even more important, such a scalable computation power and flexibility is delivered at a rather low cost by commodity GPU hardware.

In this paper, we address the problem of accelerating simulation of heterogeneous SoC architectures (i.e. composed by a host general purpose processor plus a many-core accelerator) by utilizing the high computation power provided by off-the-shelf graphic accelerator cards (GPGPU). Specifically, our idea is to partition SoC simulation workload between CPU and GPGPU. The CPU is in charge of simulating the host

processor of the target SoC. Fast simulation of this subsystem is achieved by using QEMU [11], a well-known processor emulator based on dynamic binary translation. The GPGPU runs a simulator of an embedded many-core accelerator, on top of which we execute parts of the simulated applications that are offloaded to the SoC accelerator.

The many-core simulator running on the GPGPU is based on our previous simulation technology, which we extended to support interaction with QEMU. Another extension that we present in this paper concerns better support for synchronization in shared memory programs, the most widely adopted paradigm for the targeted many-core accelerators. When the number of threads is larger than the number of physical cores on the GPU, inter-thread synchronization becomes an issue. Indeed, due to the lack of native support for inter-block synchronization, barrier primitives involving a large number of threads suffer from performance overheads due to the need for costly communication through global device memory. We introduce a simulation software infrastructure which enables efficient synchronization among the simulated cores.

The paper is structured as follows: the next section gives an overview of related work and highlights our contributions. Section 3 presents an overview of full simulation framework. Section 4 details the techniques used to effectively parallelize the simulation of manycore architecture on GPU platform. It also describes the approach used for interfacing with QEMU to achieve heterogeneous full simulation. Finally, Section 5 presents experiment results followed by a section that discuss conclusion and future work.

2. RELATED WORK

Scientific community and industry have long been using virtual prototyping tools and simulators to design hardware and software platforms. The need of simulating heterogeneous and large manycore systems has been recently detected and addressed but slow simulation speeds are a serious impediment to achieving full system simulation. Conventional simulation tools cannot adequately deal with the diverse modeling requirements and performance versus accuracy trade-offs, and must evolve to tackle the challenges inherent in simulating such complex and heterogeneous architectures.

In the area of full-system simulation platforms, there are several sequential simulators/emulators such as RSIM [17], Simics [19], Proteus [14] and QEMU [11]. Some of these are capable of simulating parallel target architectures but all of them execute sequentially on the host machine and their manycore simulation capability is limited to only hundreds of cores. OVPSim, a more refined tool [9], provides the infrastructure for describing platforms with one or more processors containing shared memory and buses in arbitrary topologies and peripheral models. Unfortunately these emulation tools are not able to tackle efficiently the complexity and heterogeneity of state-of-the-art SoCs [9, 19]. Some work [22, 12, 30, 29] has been focused on enhancing QEMU with the capability of a more accurate hardware simulation infrastructure by interfacing QEMU with SystemC[8]. However, QEMU performance is also strongly affected by the poor performance of the SystemC simulation when it is targeted for manycore simulation.

Parallel simulators of parallel target architectures include: SimFlex [28], GEMS [20], COTSon [13], BigSim [31], FastMP [18], Wisconsin Wind Tunnel II (WWT II) [23], and Graphite [21]. SimFlex and GEMS both use an off-the-shelf sequential emulator (Simics) for functional modeling plus their own models for memory systems and core interactions. Because Simics is a closed-source commercial product it is difficult to experiment with different core architectures. COTSon uses AMD's SimNow! for functional modeling and therefore suffers from some of the same problems as SimFlex and GEMS. BigSim and FastMP attain poor scalability when dealing with increasing complexity in the simulated architecture. Graphite is a distributed, parallel simulator which uses expensive multi-machine distribution for simulation of multi-cores system.

In the past, hardware emulation using special-purpose machines have been proposed for manycore system emulation to assist the application development process for multi-core processors [27]. 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, a software GPGPU-based solution provides better flexibility and scalability. Moreover it is cheaper and more accessible to a wider community. The main novelty of this paper is the development of a novel full system parallel simulation technology that leverages the computational power of widely-available and low-cost GPUs. The main idea of this paper is the exploitation of the inherent parallel processing power available in modern GPUs for the simulation of manycore coprocessors. In a nutshell, every core composing the manycore accelerator is directly mapped on a software thread running on the GPU.

3. OVERVIEW OF FULL SYSTEM SIMULATION OF HETEROGENEOUS SOCS

In this section we provide the architectural details of the heterogeneous SoC targeted by our simulation technique. We discuss an overview of our simulation framework, explaining how we interface QEMU with GPU simulation using semihosting. Finally, we give details on our GPU manycore simulation technology.

3.1 Simulation Framework

The target heterogeneous SoC consists of a *host processor* connected with a *many-core accelerator* (co-processor), as shown in Figure 1. This accelerator features many (hundreds of) simple cores, each equipped with data and instructions scratchpad memories (SPM) and all allowed to access a common shared memory region. This architecture is representative of most parallel embedded accelerators.

The programming model assumed for this kind of SoCs is based on the host-accelerator paradigm dictated by standards such as OpenCL. The execution of a parallel application is divided between the two entities: the host is normally in charge of executing (beside an operating system) the sequential part of an application, while the accelator is specialized in executing highly-parallel and computation-intensive

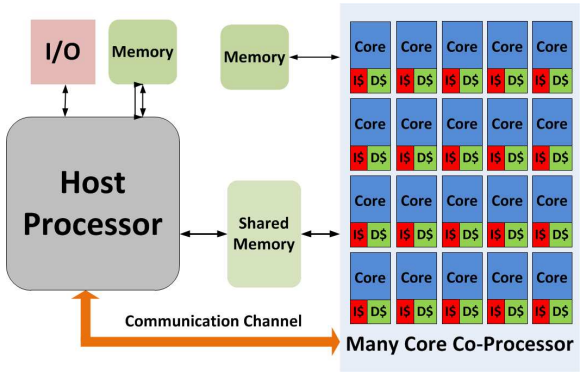

Figure 1: Target Architecture of Heterogeneous SoC

programs. The execution flow starts from the host and upon a parallel program region is encountered it is offloaded to the accelerator to benefit of its high performance. The accelerator from itself is not able to take care of all initialization procedures needed before starting the execution, it is the host that before offloading a parallel kernel is in charge of preparing a kind of execution environment (e.g. I/O buffers allocation and initialization, parallel function pointer) . The offloading procedure is normally an asynchronous call in order to give the host the possibility to continue its execution in parallel with the coprocessor, anyway all parallel programming models give the way to synchronize the execution of the two entities.

Figure 2 depicts the full-system simulation methodology adopted to model heterogeneous SoC architectures. Our simulator consists of two main blocks. QEMU emulates the host processor, capable of executing a full-fledged linux OS and file system. The simulation of the many-core accelerator leverages a fast and functionally accurate simulator (*GPUSim*, from now on) written in CUDA and running on top of the GPGPU. The interface between QEMU and *GPUSim* is provided by using QEMU's *semihosting* feature.

3.2 QEMU overview

QEMU [11] is a fast open source processor emulator able to model various target CPU architectures (e.g. ARM, Sparc, X86). It is based on dynamic binary translation, an emulation technique relying on on-line code translation to speedup the entire applications execution process. QEMU allows two different simulation modes: **User mode** and **System mode**. In the first one only the CPU architecture is modeled and it is used to run bare-metal applications on the target processor. In the second, beyond the CPU architecture, it is also possible to model complete development boards with a set of common use devices (e.g. Ethernet interfaces, Disks, Audio controllers, GPUs), enabling the execution of an un-modified operating system. In this work we will focus on the **System mode** emulation. We choose an ARM Versatile PB baseboard as a *host* system (featuring an ARMv7-based core), exploiting **semihosting** as the interface towards *GPUSim*.

Semihosting [2] is a technique developed for ARM targets allowing the communication between an application running

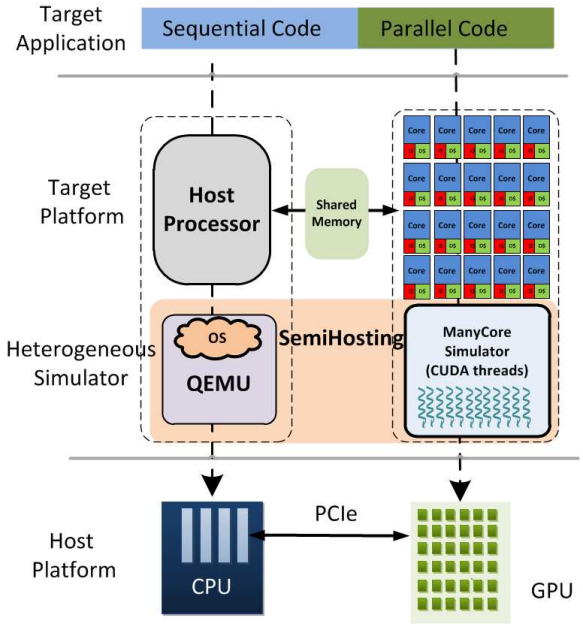

Figure 2: Full Simulation Framework

on the target and a host computer running a debugger. In other words, it enables the redirection of all the application's IO system calls (e.g. `printf()`) to a debugger interface. Details on how we exploit this feature to implement communication with *GPUSim* are provided in section 4.1.

3.3 Overview of Many-core Simulator

GPUSim provides scalable and fast simulation of thousand simple cores executing their instruction streams and maintaining the functional correctness of the program visible state. The key idea behind our approach is to identify the inherent parallelism in many-core architecture simulation and efficiently execute it on top of highly parallel GPU hardware. *GPUSim* is entirely written using C for CUDA [5] and in order to model thousands of cores we map each instance of a simulated core to a single CUDA thread. Many-cores are thus naturally modeled by using the numerous available GPU threads. Each simulated core is assigned its own context structure, which represents register file, status flags, program counter, etc. The necessary support data structures are initially allocated from the main (CPU) memory for all the simulated cores. The host program (i.e. the part of *GPUSim* running on the CPU) initializes these structures, then copies them to the GPU global device memory, along with the program binary. Once the main ISS simulation kernel is offloaded to the GPU, each simulated core repeatedly fetches, decodes and executes instructions from the program binary. Each core updates its simulated registers file and program counter until program completion.

4. IMPLEMENTATION DETAILS

In this section we describe implementation details for the various components of our simulation framework. We first explain the technique used to interface QEMU with *GPUSim*, then we provide details of *GPUSim* itself, along with a de-

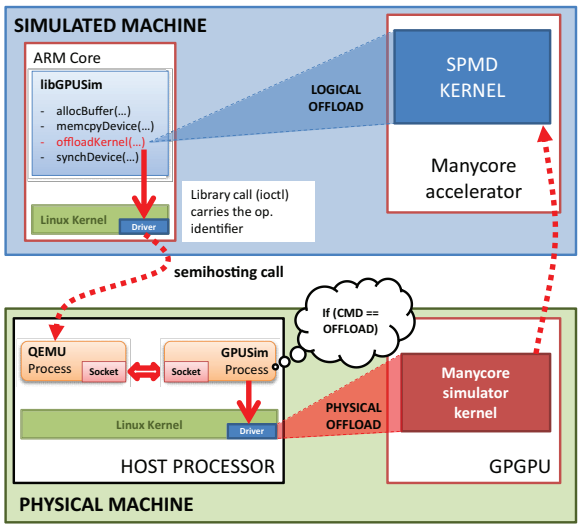

Figure 3: Accelerated simulation approach through semihosting

scription of the parallel programming model used.

4.1 QEMU semihosting

The essence of semihosting is a particular software interrupt (`svc 0x123456`) that, when caught from a debugger interface, redirects on the host machine the operation otherwise performed on the target. This interrupt takes two parameters in registers *r0* and *r1*: the first carries the semihosting operation identifier, a set of them has been pre-defined by ARM and we defined our custom `GPU_SEMI_ID`, while the second is a parameter to be passed to the host (e.g. a string pointer for a *printf()* call). When the *host processor* in the simulated SoC encounters a request for offloading a region of code to the (simulated) *accelerator*, it should trigger the execution of *GPUSim* on the CPU of the physical machine hosting QEMU itself.

The simulated *host processor* runs a Linux OS, for which we have developed a device driver (which registers a character device, `/dev/gpusim0`) that is invoked when there is the need to communicate to the simulated *accelerator* (i.e., upon simulated code offloading). This driver in fact acts as a bridge between the *simulated machine* and the *physical machine* (QEMU), performing semihosting calls instead than actually executing operations on the simulated hardware. Figure 3 shows the interaction between the two worlds.

Currently, to write programs capable of offloading code portions to the many-core accelerator we use a custom parallel programming API, *libGPUSim*, which performs all the necessary calls to the driver (e.g. `open`, `ioctl`). With a state-of-the-art programming model such as OpenCL in mind, we identified a generic subset of functions enabling the programmer to allocate buffers, move data between buffers and to offloading a certain portion of code to the accelerator requesting its execution. This API also provides a way to synchronize the execution of the host with the accelerator. The set of functions provided by our library is summarized in the following list:

- Allocate/Free buffer on the device memory.
- Transfer data to/from device memory area.
- Offload the execution of a parallel function.
- Wait for computation end, useful to define synchronization points.
- Release the co-processor, computation terminated.

We have highlighted in red in Figure 3 the steps undertaken when encountering an offload request in the simulated application. When the corresponding library call is performed, a structure is created to provide *GPUSim* with all parameters needed to perform the requested operation. This structure contains a pointer to the parallel function entry, the size of the function code itself plus a pointer to each possible parameter. A pointer to this structure is passed to the driver through an `ioctl` call, which also specifies an unique identifier for the requested operation. Consistent with the functions provided by our library, we defined a set of possible identifiers representing the operation requests that can be forwarded to the accelerator. When the Linux driver catches the `ioctl` command, it forwards the request made from the application towards *GPUSim* by raising the semihosting interrupt and setting all the parameters according to the operation identifier received. QEMU intercepts the particular software interrupt and sets-up the execution of the requested operation on *GPUSim*. In order to allow the **real parallel execution of QEMU and *GPUSim*** we implemented a solution based on a daemon process and Unix Domain sockets. When the first request from a target user-space process arrives, QEMU forks a daemon process that waits on a socket for incoming commands and that is responsible for managing execution of *GPUSim*. Henceforth all requests coming from the same process on the target environment are passed to the daemon using the appropriate socket. The daemon stays alive until all received commands are completed. To mimic the real hardware behavior, only one process from the target environment is allowed to use the accelerator at a certain time. Thus, once a process obtains the "ownership" of the co-processor it has exclusive access to it until the computation is completed.

The structure of parameters pointed to through a semihosting call is moved between host and target environment using two QEMU functions (lock_user, unlock_user) able to access the simulated memory for reading and writing. This method is used for all data (parallel code, I/O data) that needs to be moved from an environment to the other. Once copied, data is passed to the daemon using sockets; this happens only for small-sized data like pointers or operation identifiers. To avoid overheads due to moving big amounts of data using the socket, larger data structures (e.g. I/O buffers) are accessed by the daemon through shared memory segments defined and allocated by the QEMU process. In that case the socket is used only to pass the pointer to the specific shared memory segment.

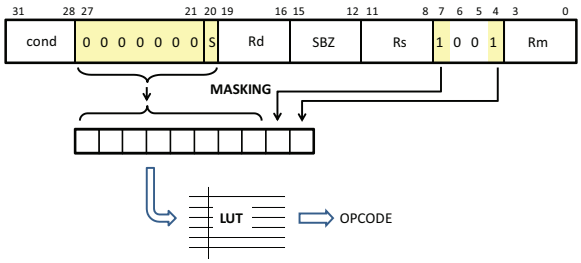

Figure 4: ARM instructions decoding

4.2 Accelerating Manycore Simulator using GPU

GPUSim is a parallel many-core simulator designed for fast execution on GPGPUs. The simulation infrastructure here presented is based on our previous work [26] [25]. While *GPUSim* can support different target ISAs, in this work we will consider an ARM-v5 architecture. Besides integration with QEMU, another main enhancement to this simulation technology that we introduce in this paper is full support for collective and point-to-point synchronization between simulated cores. In the remainder of this section, we outline the key implementation issues we had to face to achieve effective GPU-based simulation and summarize the details, coming from our previous works, needed by the reader to better understand our approach. Finally, we explain our novel techniques to efficiently support shared memory-based synchronization in *GPUSim*.

4.2.1 Control Flow Divergence

Due to the SIMT nature of GPU architectures, divergent control flow in CUDA programs force all paths in a conditional control flow to be serialized, thus being detrimental to GPU performance. It is paramount to reduce this serialization effect to a minimum. To achieve this goal, we design our simulation framework so as to minimize frequent divergent execution flows in all stages of the target ARM ISA pipeline. We ensure that cores simulated by a group of threads in a CUDA warp fetch their instructions from their instruction caches in parallel. Similarly, fixed-length 32-bit instructions from the ARM ISA can be decoded in parallel through the use of bitmasks to extract significant bits and a look-up table (see Fig. 4). This avoids the performance-critical divergence of threads in a warp.

During the execution step, previously extracted opcode and operands are used to simulate the target instruction semantics. The actual instruction execution is modeled within a C `switch/case` construct. This is translated from the CUDA compiler into a series of conditional branches, which are taken depending on the decoded instruction. The number of divergent instructions during this step is clearly highly dependent on the target application and the form of parallelism it exploits.

When running Single Program Multiple Data (SPMD) applications all cores simulate the same instruction flow. This matches the native GPU SIMT model of execution, and results in high parallelism of the simulation. On the contrary, when simulating a Multiple Program Multiple Data (MPMD) parallel application, simulated cores execute distinct instruction streams, which is highly likely to result in control flow divergence during the execution stage of the pipeline. This implies performance loss due to serialization of threads execution within a warp.

4.2.2 Global Memory Access Cost

GPU global memory has a very high latency of 400-800 cycles. Given the criticality of memory accesses and their impact on simulation performance, it is of the utmost importance to carefully design the layout of ISS execution contexts in memory. Each core contexts can be represented with an aggregated data structure (i.e., an array of registers and flags). Since each simulated core is mapped to a CUDA thread, parallel ISS execution implies concurrent accesses to core contexts, which we want to service in the most efficient manner. Every ISS frequently accesses its execution context during program execution, so it is beneficial to place the corresponding data structure in the low-latency shared memory rather than accessing it from the global memory. This requires explicit copy operations, whose cost can be minimized if we utilize *memory coalescing*, a well known optimization for effective memory bandwidth exploitation in CUDA programming. Therefore, we design the matrix layout of the *context* data structure in accordance with its access pattern. Padding is employed whenever the number of simulated cores is not an integer multiple of the size of a CUDA warp (i.e., a block of 32 parallel threads), Fig. 5.

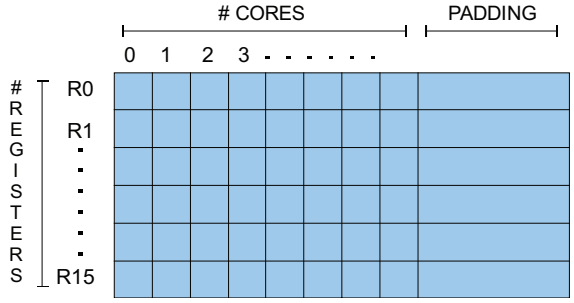

Figure 5: A matrix representing execution contexts of simulated cores in global memory.

With the proposed representation of execution contexts, concurrent simulated cores access memory in column-major order. In this way, separate threads in a warp access the same memory line, which is served with a single memory transaction, thus improving performance.

Another measure adopted to minimize global memory accesses is related to the instructions decoding look-up table. This data structure is accesses by simulated cores in a sparse read only way (processors can fetch different program instructions), and if allocated in global memory can lead to a significant performance loss. To obtain the maximum possible performance we allocate the decoding look-up table in texture cache.

4.2.3 Bank Conflicts in Shared Memory

Another important optimization is to ensure that there are no bank conflicts in the shared memory. Accesses by multi-

ple threads to a single data element within the same bank will cause conflicts and result in longer transactions. Hence, we place our CPU context data structure in shared memory in such a way that each access of simulated core to its context register file only results in linear addressing [4]. As a result, all threads in a half-warp access different banks, thus minimizing bank conflicts.

4.2.4 Synchronization

One of the main requirements of shared memory parallel programming is synchronization primitives such as *locks*. Many-core simulation should support effective inter-core synchronization mechanisms. To implement this feature, it is important that we have means to provide synchronization among the CUDA threads which are used to map the virtual cores of co-processor simulator. CUDA threads within a block can synchronize without any difficulty, however synchronization among threads blocks is not natively supported by CUDA and the GPU hardware, and can only be achieved through costly communication through the global memory, or through interaction with the CPU. This poses a serious performance bottleneck, because both solutions impose extremely costly operations. Therefore, we design techniques to minimize the performance overhead and achieve synchronization in the most efficient manner.

The support for shared memory synchronization is implemented with a combination of software primitives relying on specific instructions from target ISAs and dedicated hardware modules implemented in *GPUSim* to handle the atomicity control. Considering the ARM ISA, the programming API provides application-level primitives which translate into target operations such as *test-and-set*, *spin locks*, *wait-for-event* and *signal-event*. On the simulator side, the implementation of these operations is done using CUDA atomic instructions [5]. This ensures the correct execution of synchronization operations within all cores simulated by active thread blocks on GPU hardware. However, due to the lack of support for inter-block synchronization in CUDA, supporting synchronization among all simulated cores (mapped on both active and inactive CUDA thread-blocks) is a challenging task.

In a naive implementation where locks are implemented with busy waiting, we may easily experience deadlocks when simulating a higher numbers of cores than the available physical GPU processors. The GPU hardware scheduler selects thread-blocks for execution based on available computational resources. If the number of threads in all blocks is higher than the number of processors, only a subset of all the blocks can execute, while the remaining blocks wait (inactive) until the first set finishes its execution. This generates a deadlock. To address this issue we consider three different approaches:

1 **Preemption at Instruction Level** In this case, the ISS simulation is preempted and a synchronization point with the host CPU is inserted after every single simulated instruction. We save the ISS context state in GPU global memory, then we perform a global (barrier) synchronization by terminating the CUDA kernel and returning execution to the host side. Since the amount of computation (i.e. simulating a single instruction) is extremely small in comparison with communication between host CPU and GPU device, this approach shows high performance overhead.

2 **Preemption at Synchronization Level** In this approach, synchronization between different CUDA blocks is done only when inter-block communication is necessary. Only synchronization instructions between different cores and *wait-on-response* instructions from a simulated core residing on a currently inactive block may cause deadlock problems. Thus, by identifying such potentially "dangerous" instructions and preempting ISS simulation only upon their occurrence, we achieve considerable performance gain.

3 **Preemption at Timeout** Here, we further optimize the performance by preempting the core simulation only when the synchronization instruction occurs within a simulated core residing on an inactive block. Since it is difficult to determine in advance if the inter-core communication occurs between cores simulated on an active or inactive block, we adopt a timeout mechanism. At any point, if the simulator discovers that a particular simulated core has been waiting for an event for longer than a certain time period, it sends a trap which yields the simulation of the waiting core. This allows the simulation of previously inactive blocks to proceed, thus removing the deadlock.

We provide an evaluation of these approaches in Section 5.4.

5. EXPERIMENTS AND RESULTS

This section presents a set of experiments to evaluate the performance of our simulation infrastructure. We begin with providing the details of our experimental setup, and then we present the overhead of using QEMU semihosting technique in our simulation method. Then, we present the scalability of our many-core accelerator simulator and the speedup achieved by running parallel benchmarks on our many-core heterogeneous simulator. Finally, we provide an evaluation of our proposed synchronization techniques.

5.1 Experimental Setup

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. As a host platform (the physical machine in Figure 3) we used an Intel Xeon 2.67 GHz multi-core system running Linux 2.6.32, equipped with 6GB DDR3 DRAM. We used arm-linux-gcc for generation of target binaries for ARM.

In Tab 1, we list the benchmarks adopted for our experiments with associated dataset size. The considered benchmarks are representative computational kernels found at the heart of many embedded applications. They have been extracted from a complete JPEG decoder and from the OpenMP Source Code Repository [15] benchmark suite. The data parallel nature of these benchmarks makes them amenable to parallelization with a host-accelerator scheme. Specifically, an identical computation is replicated over parallel

Kernel	Acronym	Source	Dataset size
Inverse DCT	IDCT	JPEG Decoding	IMGSIZE (1024*512) Blocksize (8*8 pixels)
Luminance Dequantization	DQ	JPEG Decoding	IMGSIZE (1024*512) BlockSize (8*8 pixels)
Matrix Multiplication	MM	Fox Algorithm	(8192x50)*(50x50)
Background Subtraction	NCC	Normalized Cut Clustering	IMGSIZE(8198*16)
Fast Fourier Transform	FFT	OpenMP Source Code Repository	8192 parallel Rows

Table 1: Benchmarks

threads, which operate on disjoint chunks of the iteration space and dataset, according to their identification number. We parallelized these benchmarks using the primitives of our *libGPUSim* library (see Section 4.1). To study the effectiveness of our simulation methodology over a wide range of modern and possible future many-core implementations, we provide results for parallelized benchmarks running on number of simulated cores which varies from 128 to 4096 cores.

5.2 Overhead of Interfacing with QEMU

As mentioned in Section 3.2, all the benchmarks are launched from within a Linux OS running on QEMU. During the executions of these benchmarks, when a parallel kernel of the benchmark is encountered, it is offloaded for simulation on top of *GPUSim* using the QEMU semihosting technique. We want to evaluate this one-time overhead for kernel offloading. In our first experiment, we run all the benchmarks for increasing core count of the simulated accelerator, and measure the wall clock simulation time at the boundaries of these parallelized kernels. This measure accounts for the GPU execution time as well as the overhead for our QEMU semihosting extensions. We calculated the total time spent during the semihosting operation between QEMU and *GPUSim* for each of the mentioned benchmark. As shown in Table 2), this one time QEMU semihosting cost is almost constant for all the benchmarks and in absolute terms it is very contained (0.04 seconds on average).

MM	IDCT	NCC	FFT	DQ	Average
0.04	0.04	0.05	0.05	0.04	0.04

Table 2: Semihosting cost [sec]

MM	IDCT	NCC	FFT	DQ
1,26,657,376	70,771,680	11,534,336	2,433,024	3,746,816

Table 3: Total Number of Instructions simulated by each benchmarks on Many-core Simulator

5.3 Benchmark Execution Time

In this section we want to estimate the impact for our semihosting solution in overall simulation time for all the benchmarks. Our experiments aim at mimicking the typical offloading pattern of the OpenCL programming style. Thus, all the kernels presented in Table 1 are executed from within one of these offloading sequences. Specifically the main program executes on the *host processor*, but it does nothing more than a synchronous call to the offloading primitives. We measure wall clock time at the boundaries of this call, to account for the actual GPU simulation time, plus the overhead for our QEMU semihosting extensions.

The breakdown of these two contributions is shown in Fig. 6 for all the considered benchmarks. On the X-axis we report increasing core count for the simulated *accelerator*, while on the Y-axis we plot overall simulation time. The latter is normalized to the time taken by the execution of the target benchmark on a single QEMU simulated *host processor* (i.e., a scenario where instead of offloading parallel kernels to the *accelerator*, we execute them entirely on the *host processor*). In the plots we have highlighted the performance of QEMU (Y=1) with a purple line. This metric gives an idea of the benefits of the *GPUSim* approach as opposed to QEMU emulation alone. QEMU uses fast binary translation techniques, where *GPUSim* is based on a slower interpretation-based method for simulating the ARM core pipeline. From Figure 6 it can be seen that while on average the simulation time for a single QEMU processor is comparable to *GPUSim* for a small number of cores, parallel simulation provides much better results when the number of simulated cores increases (17x speedup when simulating 4096 cores). It has also to be noticed that for benchmarks like FFT and DQ semihosting is taking the major part of the entire simulation time, effect that is accentuated increasing the number of cores. This happens because for those benchmarks the number of instructions executed by the many-core simulator is low (see Tab. 3) and when simulating large machines the chunk of work assigned to each core is not big enough to hide semihosting. For benchmarks, like MM, with a higher number of simulated instructions this effect is less visible even for high number of cores.

In our next experiment, we aim at highlighting the advantages of using GPU based parallel solution for manycore simulator against any CPU based sequential version of such simulator. To the best of our knowledge, no such many-core simulator is available,therfore we have developed a sequential version of manycore simulator which instead of running on GPU runs sequentially on host CPU. Similar to our GPU based solution, this version of simulator also emulates the simple ARM pipeline for each core and it is interfaced with QEMU using the semihosting method where instead of triggering the execution of our GPU simulator, in this case we launch the sequential version running on CPU instead. As seen from the results obtained in Figure 7 the speedup increases linearly with increases number of cores and is as high as 64x when simulated for 4096 cores. As it can be noticed that benchmark such as MM and IDCT show somewhat better speedup than rest other. This is due to the fact that since MM and IDCT benchmarks are comparatively longer running benchmarks as shown in Table 3, therefore they gain better from parallelization on GPU.

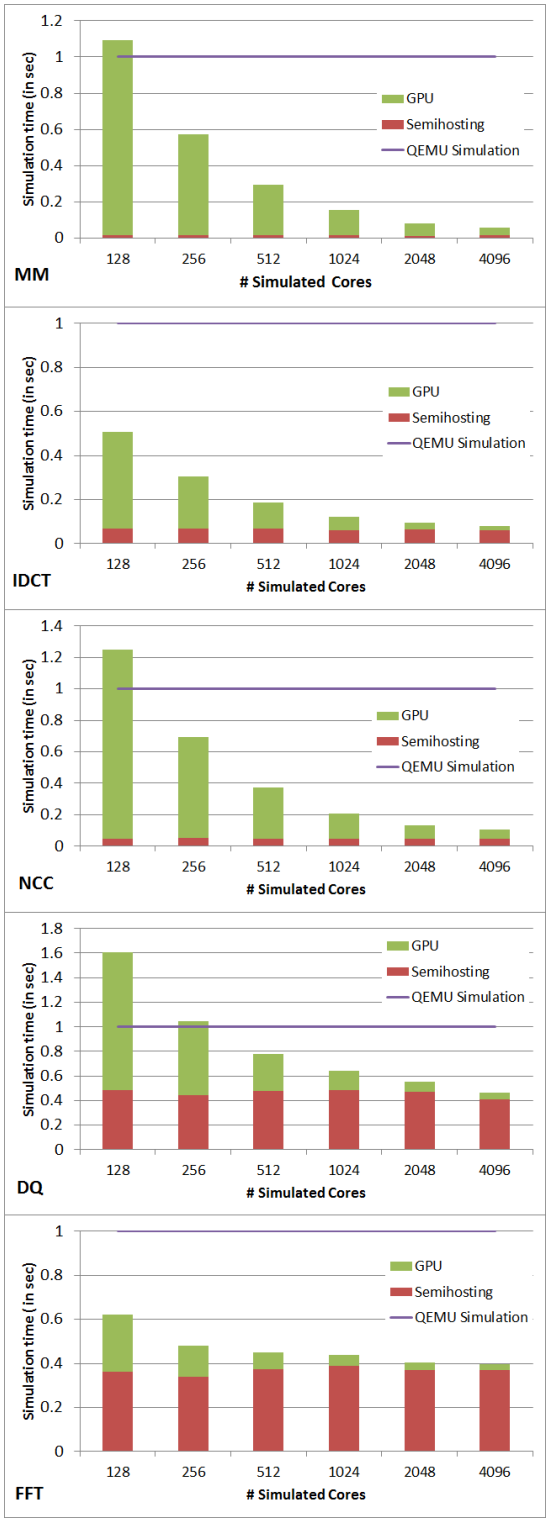

Figure 6: Breakdown of QEMU semihosting + GPU simulation time, normalized to serial benchmark execution on a single QEMU processor

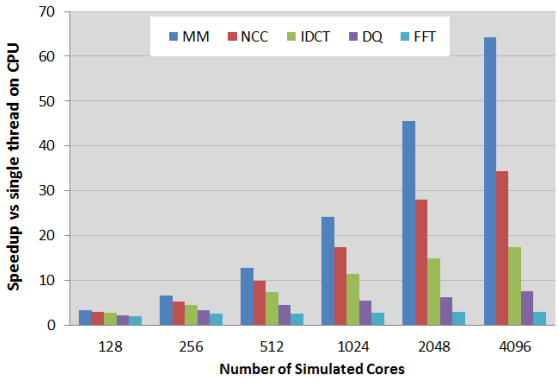

Figure 7: Speedup of parallel benchmark simulation time running on QEMU+GPU Simulator compared against sequential version of many-core simulator running on host CPU.

5.4 Evaluation of Synchronization Techniques

In this section we evaluate the three different approaches for supporting deadlock-free synchronization primitives described in Section 4.2.4. To perform this evaluation we consider a classical producer-consumer synchronization pattern. The experiment simulates a total of 4096 cores, with a CUDA thread-block size of 256. As mentioned in Section 4.2.4, the goal is to investigate the possible deadlock scenario where the simulated cores are mapped on both active and inactive thread-blocks. To do this, we maximize the usage of available GPU resources within each thread block. This ensures that the maximum number of thread blocks are active at one time and that each thread block is individually running on a single Streaming Multiprocessor (SM). One thread block remains inactive until it is granted its resources. Deadlock occurs when all cores acting as consumers are simulated on inactive thread blocks, because producer cores are stalling GPU resources while waiting for a response from consumers. Since it is randomly decided by the GPU scheduler if the consumer cores will be mapped onto active or inactive thread blocks, it is important to evaluate all possible cases. Therefore, we selectively control the cores that act as consumers and generate results for the three different cases described below:

- **Case 1** : Consumer Core simulated with an active CUDA thread block
- **Case 2** : Consumer Core simulated with an inactive CUDA thread block (triggers deadlock)
- **Case 3** : Consumer Core simulated with both active and inactive CUDA thread blocks

Only Case 2 is prone to deadlock, because consumers are mapped to threads forced into inactive blocks. The purpose of this experiment is to identify the optimal approach between those proposed in Section 4.2.4 to ensure deadlock-free and efficient synchronization. Table 4 shows the simulation time of this benchmark using the three different approaches. Although all the three techniques resolve the deadlock of Case 2 by yielding running blocks and allowing inactive ones

Cases	Preemption at Instruction Level	Preemption at Synchronization Level	Preemption at Time-out Level
Case 1	183.2 ms	3.3 ms	2.6 ms
Case 2	210.1 ms	4.2 ms	7.5 ms
Case 3	204.7 ms	3.5 ms	2.7 ms

Table 4: Simulation of Producer-Consumer Benchmark and Comparison for various approaches of Implementation

to proceed, there are clear differences in performance. Preemption of core simulation at synchronization level achieves much better performance than the baseline (preemption at instruction level). This is because we reduce the number of required synchronization points. When preempting core simulation after a timeout, for Case 1 and Case 3 (where the deadlock does not take place), we do not see any overheads for CPU barrier synchronization. Even when the deadlock occurs (Case 2) the performance loss induced by wait-for-timeout seems acceptable. Thus, the best implementation for synchronization primitives can be selected depending on the nature of the simulated workload. If the target application seldom generates situations similar to Case 2, preemption at timeout is the most convenient implementation. Otherwise preempting core simulation at synchronization level can be more conservatively enabled.

6. CONCLUSION

This paper presents a QEMU and GPU based simulation infrastructure targeting the systems with thousand-core co-processor and address the limitation of our previous work by achieving high performance full system simulation. This novel simulation method utilizes QEMU to emulate master CPU running entire operating system and devices and it uses a GPU based highly scalable manycore simulator written in CUDA to emulate the manycore co-processor. It uses semi-hosting technique to interface between QEMU and manycore simulator by offloading the parallel section of the program to the co-processor simulator. Thus our proposed simulation method utilizes the heterogeneous accelerators to tackle the challenge of simulating heterogeneous architecture of future SoC system. Moreover we proposed a novel software GPU inter block synchronization mechanism going beyond hardware limits imposed by the hardware architecture. This work can be considered as an extension of QEMU towards the modeling of an ever vast and complex range of hardware platforms.

We plan to extend our work in several directions. We will integrate our full system simulation framework with the cache and network-on-chip simulation model which has been designed to work effectively on GPU platform [25]. We also intend to add the timing model for co-processor which will be able to utilize the event traces generated from functional model of manycore simulator to estimate the overall system performance.

7. REFERENCES

[1] Arm-gpu hybrid supercomputer. http://www.montblanc-project.eu/.

[2] Arm semihosting interface. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471c/Bgbjhiea.html.

[3] Larrabee. http://software.intel.com/en-us/articles/larrabee/.

[4] Nvidia cuda best practices guide from the cuda toolkit version 3.2. http://developer.download.nvidia.com.

[5] Nvidia cuda programming guide, 2007. NVIDIA CUDA Programming Guide, 2007. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[6] Nvidia tegra. http://www.nvidia.com/object/tegra-2.html.

[7] Omap processors. http://focus.ti.com.

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

[9] The open virtual platforms (ovp) portal. http://www.ovpworld.org/.

[10] Pluralitys website. http://www.plurality.com.

[11] Qemu. http://wiki.qemu.org.

[12] Qemu systemc project. http://greensocs.com/en/Projects/QEMUSystemC.

[13] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: infrastructure for full system simulation. SIGOPS Oper. Syst. Rev., 43:52–61, January 2009.

[14] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical report, Cambridge, MA, USA, 1991.

[15] A. Dorta, C. Rodriguez, and F. de Sande. The openmp source code repository. In Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on, pages 244 – 250, feb. 2005.

[16] J. Howard, S. Dighe, Y. Hoskote, and e. a. Vangal. 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.

[17] C. Hughes, V. Pai, P. Ranganathan, and S. Adve. Rsim: simulating shared-memory multiprocessors with ilp processors. Computer, 35(2):40 –49, feb 2002.

[18] S. Kanaujia, I. Papazian, J. Chamberlain, and J. Baxter. Fastmp: A multi-core simulation methodology. In Workshop on Modeling, Benchmarking and Simulation, 2006.

[19] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. Computer, 35(2):50 –58, feb 2002.

[20] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput. Archit. News, 33:92–99, November 2005.

[21] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal.

Graphite: A distributed parallel simulator for multicores. In High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on, pages 1 –12, jan. 2010.

[22] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina. Mixed sw/systemc soc emulation framework. In Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on, pages 2338 –2341, june 2007.

[23] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, M. Hill, D. Wood, S. Huss-Lederman, and J. Larus. Wisconsin wind tunnel ii: a fast, portable parallel architecture simulator. Concurrency, IEEE, 8(4):12 –20, oct-dec 2000.

[24] P. Paulin. Programming challenges & solutions for multi-processor socs: An industrial perspective. In Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE, pages 262 –267, june 2011.

[25] Omitted for blind review

[26] Omitted for blind review

[27] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. AsanovicÌĄ and. Ramp gold: An fpga-based architecture simulator for multiprocessors. In Design Automation Conference (DAC), 2010 47th ACM/IEEE, pages 463 –468, june 2010.

[28] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe. Simflex: Statistical sampling of computer system simulation. Micro, IEEE, 26(4):18 –31, july-aug. 2006.

[29] T.-C. Yeh, Z.-Y. Lin, and M.-C. Chiang. Optimizing the simulation speed of qemu and systemc-based virtual platform. In Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on, pages 1 –4, dec. 2010.

[30] T.-C. Yeh, Z.-Y. Lin, and M.-C. Chiang. Enabling tlm-2.0 interface on qemu and systemc-based virtual platform. In IC Design Technology (ICICDT), 2011 IEEE International Conference on, pages 1 –4, may 2011.

[31] G. Zheng, G. Kakulapati, and L. Kale. Bigsim: a parallel simulator for performance prediction of extremely large parallel machines. In Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, page 78, april 2004.