2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

GPGPU-Accelerated Parallel and Fast Simulation of Thousand-core Platforms

Christian Pinto†, Shivani Raghav‡, Andrea Marongiu†, Martino Ruggiero†‡, David Atienza ‡, Luca Benini†
DEIS - University of Bologna, Bologna, IT. (†)
ESL - EPFL, Lausanne, CH. (‡)

Abstract— **The multicore revolution and the ever-increasing complexity of computing systems is dramatically changing system design, analysis and programming of computing platforms. Future architectures will feature hundreds to thousands of simple processors and on-chip memories connected through a network-on-chip. Architectural simulators will remain primary tools for design space exploration, software development and performance evaluation of these massively parallel architectures. However, architectural simulation performance is a serious concern, as virtual platforms and simulation technology are not able to tackle the complexity of thousands of core future scenarios. The main contribution of this paper is the development of a new simulation approach and technology for many core processors which exploit the enormous parallel processing capability of low-cost and widely available General Purpose Graphic Processing Units (GPGPU). The simulation of many-core architectures exhibits indeed a high level of parallelism and is inherently parallelizable, but GPGPU acceleration of architectural simulation requires an in-depth revision of the data structures and functional partitioning traditionally used in parallel simulation. We demonstrate our GPGPU simulator on a target architecture composed by several cores (i.e. ARM ISA based), with instruction and data caches, connected through a Network-on-Chip (NoC). Our experiments confirm the feasibility of our approach.**

I. Introduction

Simulation is one of the primary techniques for application development in the high performance computing (HPC) domain. Virtual platforms and simulators are key tools both for the early exploration of new processor architectures and for advanced software development for upcoming machines. They are indeed extensively used for early software development (i.e. before the real hardware is available), and to optimize the hardware resources utilization of the application itself when the real hardware is already there. With simulators, the performance debugging cycle can be shortened considerably. However, simulation times are increasing further by the needs to simulate a still wider range of inputs, larger datasets, but, even more importantly, processors with an increasing number of cores.

During last decade the design of integrated architectures has indeed been characterized by a paradigm shift: boosting clock frequencies of monolithic processor cores has clearly reached its limits [18], and designers are turning to multicore architectures to satisfy the growing computational needs of applications within a reasonable power envelope [7]. This ever-increasing complexity of computing systems is dramatically changing their system design, analysis and programming [12].

New trends in chip design and the ever increasing amount of logic that can be placed onto a single silicon die are affecting the way of developing the software which will run on future parallel computing platforms. Hardware designers will be soon capable to create integrated circuits with thousands of cores and a huge amount of on-chip fast memory [21]. This evolution of the hardware architectural concept will bring to a revolution of the idea of how thinking and structuring the software for parallel computing systems [4]. The existing relation between computation and communication will deeply change: past and current architectures are equipped with few processors and small on-chip memory, which can interact via off-chip buses. Future architectures will expose a massive battery of parallel processors and large on-chip memories connected through a network-on-chip, which speed is more than hundred times faster than the off-chip one [10]. It is clear that current virtual platform technologies are not able to tackle the possible issues coming by the complexity derived by simulating this future scenario, because they suffer problems of either performance or accuracy. Cycle- and signal- accurate simulators are extensively used for architectural explorations, but they are not adequate for simulating large systems as they are sequential and slow. On the contrary, high level and hardware-abstracting simulation technologies can provide good performance for software development, but can not enable reliable design space explorations or system performance metrics because they are lacking low level architectural details. For example, they are not capable of modeling contention on memory hierarchy, system buses or network. Parallel simulators have been also proposed to address the problems of simulation duration and complexity [35][5], but they require multiple processing nodes to increase the simulation rate and suffer poor scalability due to the synchronization overhead when increasing the number of processing nodes.

None of the current simulators takes advantage of the computational power provided by modern manycores, like General Purpose Graphic Processing Units (GPGPU) [1]. The development of computer technology brought to an unprecedented performance increase with these new architectures. They provide both scalable computation power and flexibility, and they have already been adopted for many computation-intensive applications [2]. However, in order to obtain the highest performance on such a machine, the

978-0-7695-4395-6/11 $26.00 © 2011 IEEE
DOI 10.1109/CCGrid.2011.64

IEEE computer society

programmer has to write programs that best exploit the hardware architecture.

The main novelty of this paper is the development of fast and parallel simulation technology targeting extremely parallel embedded systems (i.e. composed of thousands of cores) by specifically taking advantage of the inherent parallel processing power available in modern GPGPUs. The simulation of manycore architectures indeed exhibits a high level of parallelism and is thus inherently parallelizable. The large number of threads that can be executed in parallel on a GPGPU can be employed to simulate as many target processors in parallel. Research projects such as Eurocloud[14] are building platforms to support thousands of ARM cores in a single server. To provide the simulation infrastructure for such large many core system we are developing a new technology to deploy parallel full system simulation on top of GPGPUs. The simulated architecture is composed by several cores (i.e. ARM ISA based), with instruction and data caches, connected through a Network-on-Chip (NoC). Our GPU-based simulator is not intended to be cycle-accurate, but instruction accurate. Its simulation engine and models provide accurate estimates of performance and various statistics. Our experiments confirm the feasibility and goodness of our idea and approach, as our simulator can model architectures composed of thousands of cores while providing fast simulation time and good scalability.

II. Related Work

In this section we give an overview about the state of the art in the context of architectural simulation of large computing systems. A considerable number of simulators has been developed by both scientific and industrial communities. We will try to present and extensively review the simulation environments that are most representative and widely used in the scientific community. We also highlight the potential of modern manycore architectures like GPGPUs when applied to the field of systems simulation, giving an overview of works and approaches proposed in the literature.

Virtual prototyping is normally used to explore different implementations and design parameters to achieve a cost efficient implementation. These needs are well recognized and a number of architectural level simulators have been developed for performance analysis of high performance computing systems. Some of them are SystemC based [15], like [26], others instead use different simulation technologies and engines [32], like binary translation, smart sampling techniques or tuneable abstraction levels for hardware description. These kinds of virtual platform provide a very good level of abstraction while modeling the target architecture with a high level of accuracy. Although this level of detail is critical for the simulator fidelity and accuracy, the associated trade-off is represented by a decreased simulation speed. These tools simulate the hardware in every detail, so it is possible to verify that the platform operates properly and also to measure how many clock cycles will be required to execute a given operation. But this interesting property from the hardware design point of view turns to be an inconvenient from the system point of view. Since they simulate very low level operations, simulation is slow. The slower simulation speed is especially limiting when exploring an enormous design space that is the product of a large number of processors and the huge number of possible system configurations.

Full-system virtual platforms, such as [23] [6] [25], are often used to facilitate the software development for parallel systems. However, they do not provide a good level of accuracy and can not enable reliable design space exploration or system performance profiling. They often lack low level architectural details, e.g. for modeling contention on memory hierarchy, system buses or network. Moreover, they do not provide good scalability as the system complexity increases. COTSon [3] uses functional emulators and timing models to improve the simulation accuracy, but it leverages existing simulators for individual sub-components, such as disks or networks. MANYSIM [34] is a trace-driven performance simulation framework built to address the performance analysis for CMP platforms.

Also companies showed interest in such field: Simics [19] and AMD SimNow [27] are just few representative examples. However, commercial virtual platforms often suffer from the limitations of not being open source products, and they also provide poor scalability when dealing with increasing complexity in the simulated architecture.

Complex models generally require significant execution times and may be beyond the capability of a sequential computer. Full-system simulators have been also implemented on parallel computers with significant compute power and memory capacity [24] [13]. In the parallel simulation, each simulated processor works on its own by selecting the earliest event available to it and processing it without knowing what happens on other simulated processors [22][35]. Thus, methods for synchronizing the execution of events across simulated processors are necessary for assuring the correctness of the simulation [11] [29] [28]. Parallel simulators [35][5] require multiple processing nodes to increase the simulation rate and suffer of poor scalability due to the synchronization overhead when increasing the number of processing nodes.

From this brief overview in the literature of system simulation, it can be noticed that achieving high performance with reasonable accuracy is a challenging task, even if the simulation of large-scale systems exposes a high level of parallelism. Moreover, none of the aforementioned simulation environments exploits the powerful computational capabilities of modern GPGPUs. In the last decade, GPU performance has been increasing very fast. Besides performance improvement of the hardware, the programmability also has been significantly increased.

In the past, hardware special-purpose machines have been proposed for manycore system emulation and to assist in the application development process for multi-core processors [30][9]. Even if these 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. Recently, a few research solutions have

been proposed to run gate-level simulations on GPUs [8]. A first attempt by authors in [20] did not provide performance benefits due to lack of general purpose programming primitives for their platform and the high communication overhead generated by their solution. Another recent approach [16] introduces a parallel fault simulation for integrated circuits and a cache simulator [17] on a CUDA GPU target. The main novelty of this paper is the development of a novel parallel simulation technology that leverages the computational power of widely-available and low-cost GPUs.

III. Target Architecture

The objective of this work is to enable the simulation of massively parallel embedded systems made up of thousands of cores. 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 processors will be embedding thousands of simple cores [4]. Future architectures will expose a massive battery of very-simple parallel processors and on-chip memories connected through a network-on-chip.

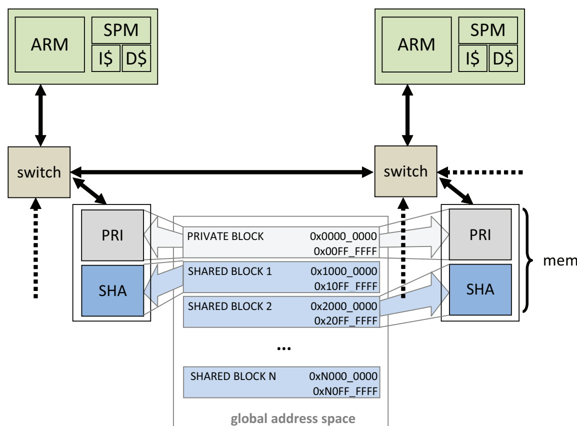


Fig. 1: Target simulated architecture

The platform template targeted by this work and our simulator is the manycore depicted in Fig.1. It is a generic template for a massively parallel manycore architecture [10][31][33]. The platform consists of a scalable number of homogeneous processing cores, a shared communication infrastructure and a shared memory for inter-tile communication. The main architecture is made by several computational tiles composed by a ARM-based CPU. Processing cores embed instruction and data caches and are directly connected to tightly coupled software controlled scratch-pad memories.

Each computational tile also features a bank of private memory, only accessible by the local processor, and a bank of shared memory. The collection of all the shared segments is organized as a globally addressable NUMA portion of the address space.

Interaction between CPUs and memories takes place through a Network-on-Chip communication network (NoC).

IV. The Fermi GPU Architecture and CUDA

The Fermi-based GPU used in this work is a Nvidia GeForce GTX 480, a two-level shared memory parallel machine comprising 480 SPs organized in 16 SMs (Streaming Multiprocessors). Streaming multiprocessors manage the execution of programs using so called “warps”, groups of 32 threads. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. All instructions are executed in a SIMD fashion, where one instruction is applied to all threads in warp. This execution method is called SIMT (Single Instruction Multiple Threads). All threads in a warp execute the same instruction or remain idle (different threads can perform branching and other forms of independent work). Warps are scheduled by special units in SMs in such a way that, without any overhead, several warps execute concurrently by interleaving their instructions. One of the key architectural innovations that greatly improved both the programmability and performance of GPU applications is on-chip shared memory. In the Fermi architecture, each SM has 64 KB of on-chip memory that can be configured as 48 KB of shared memory with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache. Fermi features also a 768 KB unified L2 cache which provides efficient data sharing across the GPU.

CUDA (Compute Unified Device Architecture) is the software architecture for issuing and managing computations on the GPU. CUDA programming involves running code on two different platforms: a host system that relies on one or more CPUs to perform calculations, and a CUDA-enabled NVIDIA GPU (the device). The device works as a coprocessor to the host, so a part of the application is executed on the host and the rest, typically calculation intensive, on the device.

A. Key Implementative Issues for Performance

When writing applications it is important to take into account the organization of the work, i.e. to use 32 threads simultaneously. The code that does not break into 32 thread units can have lower performance. Hardware chooses which warp to execute at each cycle, and it switches between them without penalties. Compared with CPUs, it is similar to simultaneously executing 32 programs and switching between them at each cycle without penalties. CPU cores can actually execute only one program at a time, and switching to other programs has a cost of hundreds of cycles.

Another key aspect to achieving performance in CUDA application is an efficient management of accesses to the global memory. These are performed without an intervening caching mechanism, and thus are subject to high latencies.

To maximally exploit the memory bandwidth is necessary to leverage some GPU peculiarities:

- All active threads in a half-warp execute the same instruction;
- Global memory is seen as a set of 32, 64 or 128 byte segments. This implies that a single memory transaction involves at least a 32 byte transfer.

By properly allocating data to memory, accesses from a halfwarp are translated into a single memory transaction (access coalescing). More specifically, if all threads in a half-warp are accessing 32-bit data in global memory it is possible to satisfy the entire team's requests with a single 64-Byte (32 bit x 16 threads) transfer.All the above mentioned aspects were taken into account to optimize the performance of code running on GPGPUs.

V. Full Simulation Flow

The entire simulation flow is structured as a single CUDA kernel, whose simplified structure is depicted in Fig. 2. One physical GPU thread is used to simulate one single target machine processor, its cache subsystem and the NoC switch to which it is connected. The program is composed by a main loop – also depicted in the code snippet in Fig. 2 – which we refer to as a *simulation step*.

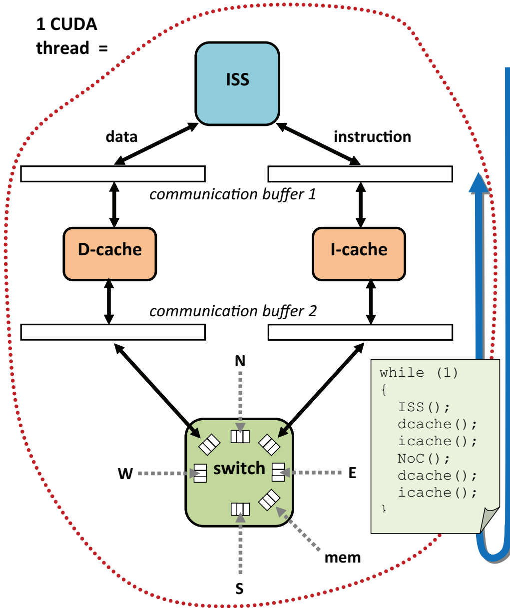


Fig. 2: Main simulation loop

The `ISS` module is executed first. During the *fetch* phase and while executing `LOAD/STORE` instructions the core issues memory requests to the `Cache` module, which is executed immediately after. *Communication buffer 1* is used to exchange information such as target address and data.

The `Cache` module is in charge of managing data/instructions stored in the private memory of each core. The shared segment of each core's memory is globally visible through the entire system. Shared regions are not cacheable. The cache simulator is also responsible for forwarding access requests to shared memory segments to the `NoC` simulator. Upon cache miss there is also the necessity to communicate with the NoC. This is done through *communication buffer 2*. For a LOAD operation (that does not hit in cache) to complete there is the need to wait for the request to be propagated through the NoC and for the response to travel back. Hence the Cache module is split in two parts.

After the requested address has been signaled on *communication buffer 2*, the `NoC` module is invoked, which routes the request to the proper node. This may be a neighboring switch, or the memory itself if the final destination has been reached. In the latter case the wanted datum is fetched and routed back to the requesting node. Since the operation may take several *simulation steps* (depending on the physical path it traverses on the network) `ISS` and `Cache` modules are stalled until the `NoC` module writes back the requested datum in *communication buffer 2*.

The second part of the `Cache` module is then executed, where the datum is made available to the ISS through *communication buffer 1*.

A. Instruction Set Simulator

The ARM ISS is currently capable of executing a representative subset of the ARM ISA. The Thumb mode is currently not supported. The simulation is decomposed into three main functional blocks: fetch, decode and execute. One of the most performance-critical issues in CUDA programming is the presence of divergent branches, which force all paths in a conditional control flow to be serialized. It is therefore important that this effect of serialization is reduced to a minimum. To achieve this goal we try to implement the fetch and decoding steps without conditional instructions.

The ARM ISA leverages fixed length 32-bit instructions, thus making it straightforward to identify a set of 10 bits which allows decoding an instruction within a single step. These bits are used to index a 1024-entry Look-Up Table (LUT), thus immediately retrieving the opcode which univocally identifies the instruction to be executed (see Fig. 3).

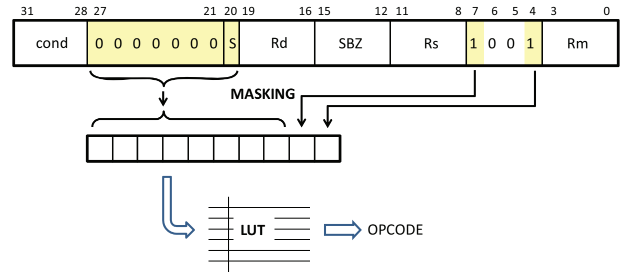


Fig. 3: Instruction decoding

Sparse accesses to the LUT are hardly avoidable, due to processors fetching different program instructions. This implies that even the most careful design can not guarantee the aligned access pattern which allows efficient (coalesced) transfers from the `global memory`. However, since the LUT is statically declared and contains read-only data, we can take advantage of the `texture memory` to reduce the access cost.

During the *execute* step the previously extracted opcode and operands are used to simulate the target instruction semantics. Prior to instruction execution processor status flags are checked to determine whether to actually execute the instruction or not (e.g. after a compare instruction). In

case the test is not passed a NOP instruction is executed. Finally, the actual instruction execution is modeled within a switch/case construct. This is translated from the CUDA compiler into a series of conditional branches, which are taken depending on the decoded instruction. This point is the most critical to performance. In SPMD[1]-like parallel computation where each processor executes the same instructions on different data sets CUDA threads are allowed to execute concurrently. In the worst case, however, on MIMD[2] task-based parallel applications each processor may take a different branch, thus resulting in complete serialization of the entire switch construct execution.

The *execution contexts* of simulated cores are represented with 16 general-purpose registers, a status register plus an auxiliary register used for exception handling, or to signal the end of execution. Due to the frequent accesses performed by every program to its execution context, the data structure was placed in the low latency shared memory rather than accessing it from much slower global memory.

B. Cache Simulator

The main architectural features of the simulated cache are summarized in Table I. Our implementation is based on a *set-associative* design, which is fully re-configurable in terms of number of ways thus also allowing the exploration of *fully-associative* and *direct-mapped* devices. The size of the whole cache and of a single line is also parameterized.

Type	*set-associative (default 8 ways)*
Write policy	*write back*
Allocation	*write allocate, write no allocate*
Replacement policy	*FIFO*
Data format	*word, half word, byte*

TABLE I: Cache design parameters

Currently we only allow a common setup for cache parameters (i.e. we simulate identical devices). No coherence protocols or commands (i.e. explicit invalidation, flush) are available at the moment. We prevent data consistency issues by designing the memory subsystem as follows:

1) Caches are *private* to each core, meaning that they only deal with data/instructions allocated in the private memory. This implies that cache lines need not be invalidated upon memory updates performed by other processors.
2) *Shared* memory regions are directly accessible from every processor, and the corresponding address range is disregarded by the caching policy.

We show the functional behavior of a single simulated cache in the block diagram in Fig. 4 for the *write-allocate* policy. The blocks which represent a wait condition (dependency) on the NoC operation logically split the activity of the `Cache` module in two phases, executed before and after the `NoC` module, as discussed in Sec. V-A. The input points (ISS, NoC) for the `Cache` module are displayed

[1]Single Program Multiple Data
[2]Multiple Instruction Multiple Data

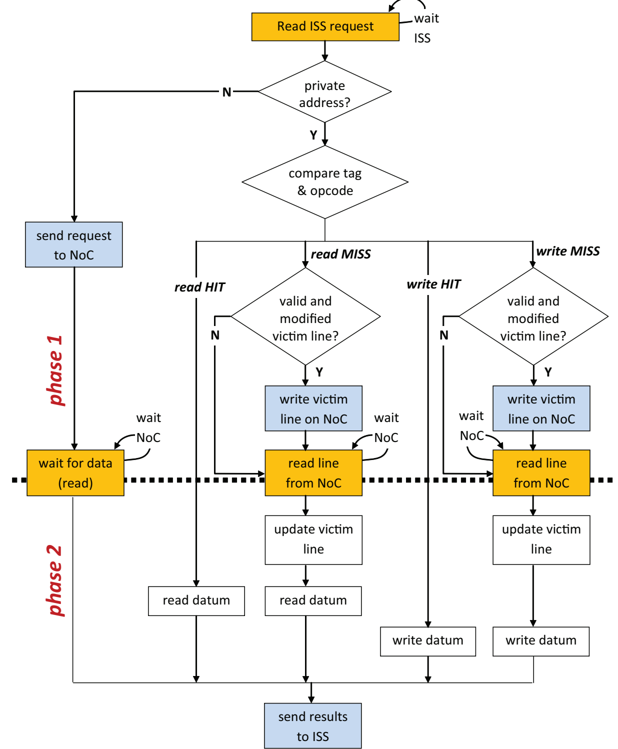


Fig. 4: Functional block diagram of a simulated cache (write-allocate)

within orange blocks. Upon execution of these blocks the presence of a message in the pertinent *communication buffer* is checked. Output operations – displayed within blue blocks – do not imply any wait activity. The code was structured so as to minimize the number of distinct control flows, which at runtime may lead to divergent branches, which greatly degrade the performance of CUDA codes.

1) Communication buffers: Communication between the `Cache` module and the `ISS` and `NoC` modules takes place through shared memory regions acting as shared buffers. Information exchange exploits the producer/consumer paradigm, but without the need for synchronization since ISS, cache and NoC modules are executed sequentially.

Buffers amidst `ISS` and `Cache` modules (*communication buffer 1*) host messages structured as follows:

1) a single-bit flag (`full`) indicating that a valid message is present in the buffer
2) an `opcode` which specifies the operation type (`LOAD`, `STORE`) and the size of the datum (word, byte)
3) a 32-bit `address` field
4) a 32-bit `data` field

The `full` and `opcode` fields are duplicated to properly handle bi-directional messages (i.e. traveling from/to the ISS). The `address` field is only meaningful for ISS-to-Cache communication, whereas the `data` field is exploited on both directions. In case of a `STORE` operation it carries the datum to be written in memory. In case of a `LOAD` operation it is used only when the cache responds to the ISS.

Messages exchanged between `Cache` and `NoC` modules (stored in *communication buffer 2*) have a slightly different structure. First, the `data` field must accommodate an entire cache line in case of a burst read/write to private addresses. If the requested address belongs to the shared range a single-word read/write operation is issued, and only 32 bits of the `data` field are used. Second, the `opcode` field should still discriminate between LOAD/STORE operations and data sizes. The latter have however a different meaning. For a cache miss (private reference) the only allowed type is a cache line. For shared references it is still necessary to distinguish between word, half-word and byte types. Third, in case of a cache miss which also requires the eviction of a (valid and modified) line it is also necessary to instruct the NoC about the replacement of the victim line. To handle this particular situation we add the following fields to the communication buffer:

1) a single-bit `evict` field, which notifies the NoC about the necessity for line replacement
2) an additional `address` field which holds the pointer to the destination of the evicted cache line

The `data` field can be exploited to host both the evicted line and the substitute.

C. Network-on-Chip Simulator

The central element of the NoC simulation is a *switch*. Each switch in the network for the considered target architecture (cfr. Sec. III) is physically connected to (up to) four neighbors, the local memory bank (private + shared) and the instruction and data caches. We thus consider each switch as having seven ports, modeled with as many packet *queues*. For each switch the simulation loop continuously executes the following tasks:

1) check the input queues for available packets
2) in case the packet is addressed to the local node, insert packet in the memory queue
3) otherwise, route the packet to the next hop of the path

Packet queues are stored in global memory. Hosting them on the local (shared) memory would have allowed faster access time, but is subject to several practical limitations. First, local memory is only shared among threads hosted on the same multiprocessor, thus complicating communication between notes simulated by threads residing on different devices (multiprocessors). Second, the shared memory has a limited size, which in turn limits the maximum number of switches that could be simulated (i.e. the size of the system).

Packet queues are implemented as circular buffers of configurable size. Their structure consists of a packet array (of the specified size) plus two pointers to the next read and write site, respectively.

The `NoC` module first copies requests coming from data and instruction caches (stored in *communication buffers 2*) into associated queues. This step is accomplished in parallel among threads and is thus very performance-efficient. Besides information included in the source buffer (see Sec. V-B.1), the queues also contain an index which identifies the node which generated the packet (i.e. the source node). This information is required to properly send back a response packet (e.g. to a `LOAD` operation). Then, the main loop is entered, which scans each queue consecutively. Within the loop, i.e. for each queue, several operations are performed, as shown in the simplified block diagram in Fig. 5.

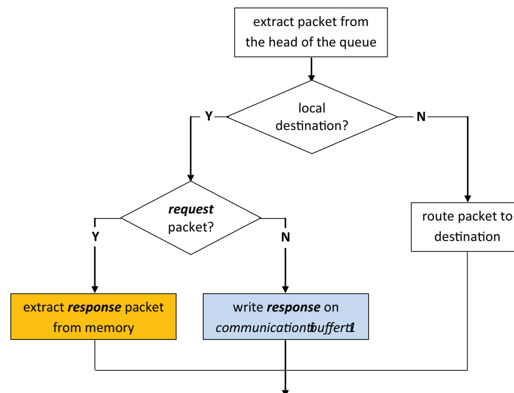
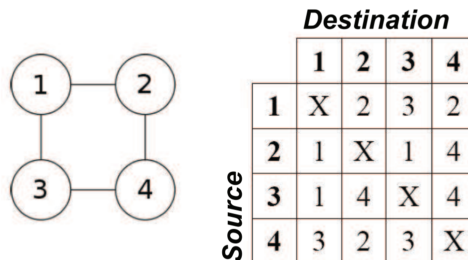


Fig. 5: Functional block diagram of the operations executed for every queue in a simulated NoC switch

First, the status of the queue is inspected to check whether there are pending packets. If this is the case, the first packet in the queue is extracted and processed. Second, we distinguish between two types of packets: *request* and *response*. Intuitively, the first type indicates transactions traveling toward memory, while the second indicates replies (e.g. the result of a `LOAD`). If the packet being processed is a *response* packet, the ID of the source node is already available as explained above. When dealing with *request* packets the destination address is evaluated to determine which node contains the wanted memory location. Third, we determine if the packet was addressed to the local node (memory, for *request* packets, or core, for *response* packets) or not. In the former case, if the packet contains a *request* the appropriate memory operation is executed and in case of a `LOAD` a *response* packet is generated and stored in the queue associated to the memory, ready to be processed in a successive step. If the packet is not addressed to the current node, it is routed toward the correct destination.



Source \ Destination	1	2	3	4
1	X	2	3	2
2	1	X	1	4
3	1	4	X	4
4	3	2	3	X

Fig. 6: 2×2 mesh and routing table (dimension-order)

Routing is implemented through a lookup table (LUT), generated before simulation starts. For every source-destination pair the next node to be traversed in the selected path is pre-computed and stored in the table, as shown in

Fig. 6. The routing table is accessed as a read-only datum from the CUDA kernel, and is thus an ideal candidate for allocation on the *texture cache*.

VI. Experimental Results

In this section we evaluate the performance of our simulator. The experiment results are obtained using a Nvidia GeForce GTX 480 CUDA-compliant video card mounted on a workstation with an Intel i7 CPU at 2.67 GHz running Ubuntu Linux OS. We carried out three different kind of experiments with our simulator. The first set of experiments is aimed at measuring the simulation time breakdown among system components, i.e. the percentage of the simulation time spent over cores, caches, NoC for different instruction types in the ISA. As a second set of experiments we evaluate the performance of our simulator – in terms of simulated MIPS – using real-world benchmarks and considering different target architectural design variants. Finally, we provide a comparison between the performance of our simulator and OVPSim (Open Virtual platform Simulator)

A. Simulation time breakdown

Since we can model different system components in our simulator (i.e. cores, I-cache, D-caches and on-chip network), it is important to understand the amount of time spent in simulating each of them.

For these evaluations, we considered a single-tile architecture composed of just one core equipped with both instructions and data caches, and a network switch connected to the main memory. We measured the cost of simulating each type of three main instructions classes, namely arithmetic/logic, control flow and memory instructions.

We considered two different granularities for our breakdown analysis. The first experiment has been conducted at the system level, and was meant to estimate the cost of modeling each component of the simulator. This allows to better understand where most of the simulation time is spent (i.e. which component is heaviest to simulate). The second analysis takes a closer look inside the core model to estimate the cost due to the simulation of each stage of the pipeline (i.e. fetch, decode and execute). In all experiments we measured the amount of host clock cycles spent to simulate each component or each stage of the pipeline.

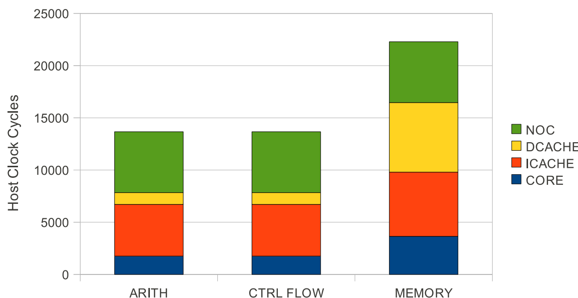


Fig. 7: Components Breakdown

Fig.7 shows the cost of each component for arithmetic, control flow and memory instructions in case of hits in instruction and data caches. Control flow and arithmetic instructions have almost the same overall cost values. Instructions involving memory operations consume instead more simulation time: intuitively they generate hit in both caches, while control flow and arithmetic ones trigger only the instruction cache. Even if packets are not propagated through the NoC, a certain amount of simulation time is required to check the status of communication buffers.

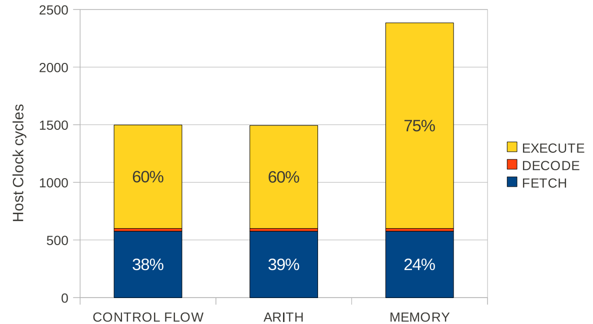


Fig. 8: Pipeline Breakdown

Fig.8 presents a deeper analysis inside the core model. As expected, fetch and decode phases take a constant number of cycles, since their duration is not influenced by the executed instruction. They respectively consume an average of 33% and 1% of the total host cycles. On the other side, the execution phase is the most time consuming and its duration varies depending on the instruction performed. This phase is also the most important source of thread divergence, exposing a different execution path for each supported instruction.

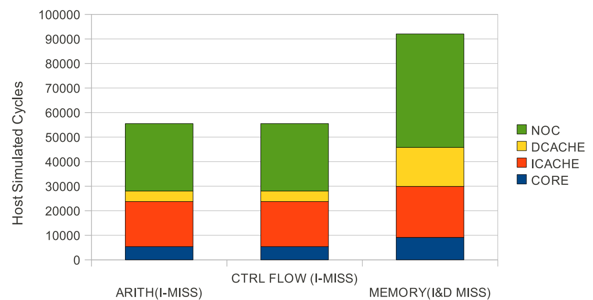
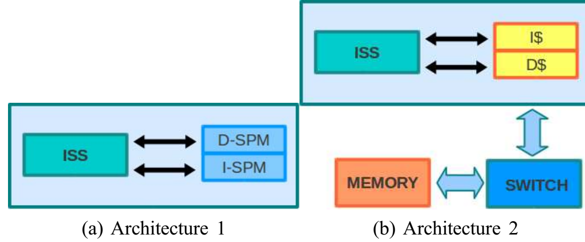


Fig. 9: Cache Miss Breakdown

Fig.9 shows the simulation time spent for arithmetic, control flow and memory instructions in presence of cache misses. Compared with Fig.7, it can be noticed that a miss in cache generates a 4x slowdown in performance. A cache miss produces indeed a trigger to all modules (namely cache and NoC), while the core is stalled until data is available.

B. Simulator Performance Evaluation

In this section we present the performance of our simulator using representative computational kernels found at the heart of many real applications. We considered two architectural templates as best and worst case, namely **Architecture 1** and **Architecture 2**.



(a) Architecture 1 (b) Architecture 2

Fig. 10: Two different instances of a simulation node representing as many architectural templates

In **Architecture 1** (Fig.10a), each core has associated instruction and data scratchpads (SPM). In this case, all memory operations are handled from within this SPM. From the point of view of the simulation engine we only instantiate ISSs. Memory references are handled by a dedicated code portion which models the behavior of a scratchpad.

In **Architecture 2** (Fig.10b), we instantiated all the simulation models including NoC, instruction and data caches. With **Architecture 1**, we configured each node with SPM size of 200K. With this memory configuration we can simulate up to 8192 cores system. Beyond that we reach the maximum limit on available global memory on Fermi card. Since **Architecture 2** has a higher memory requirement, due to large NoC and cache components data structures, we can simulate up to 4096 cores with 64K of private memory.

We investigated the performance of the presented architecture templates with five real-world program kernels which are widely adopted in several HPC applications.

- NCC (Normalized Cut Clustering)
- IDCT (Inverse Discrete Cosine Transform) from JPEG decoding
- DQ (Luminance De Quantization) from JPEG decoding
- MM (Matrix Multiplication)
- FFT (Fast Fourier Transform)

The performance of our simulator is highly dependent on the parallel execution model adopted for the application being executed so we adopt an Open MP-like parallelization scheme to distribute work among available cores. An identical number of iterations are assigned to parallel threads. The dataset touched by each thread is differentiated based on the processor ID. While selecting the benchmarks we considered the fact the application itself should scale well to large number of cores. Since in this case, our target is an 8192 core system for **Architecture 1** and 4096 cores for **Architecture 2**, we scaled up dataset to provide large enough data structure for all cores.

Table.II shows the benchmarks we used and the datasets which has been partitioned for parallel execution, as well as the total number of dynamically executed ARM instructions. The metrics we adopted to test simulation speed is Simulated Million-Instructions per Second (S-MIPS) which is calculated as total simulated instructions divided by wall clock time of the host.

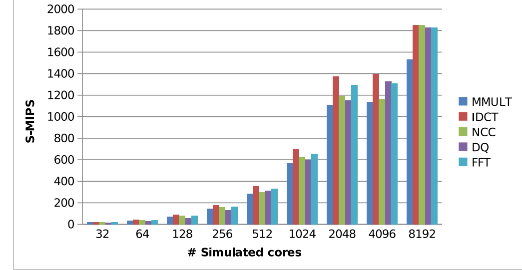


Fig. 11: Benchmarks performance - Architecture 1

Fig.11 shows the S-MIPS for **Architecture 1**. It is possible to notice that our simulation engine scales well for all the simulated programs. IDCT, Matrix Multiplication, NCC and Luminance De-Quantization exhibit a high degree of data parallelism which results in a favorable case for our simulator since a very low percentage of divergent branches takes place. FFT, on the other hand, features data-dependent conditional execution which significantly increases control flow divergence. The parallelization scheme for FFT assigns different computation to a thread depending on which iteration of a loop is being processed. Overall, we obtain an average of 1800 S-MIPS with the case when the benchmarks are executed on 8192 cores system. It is possible to notice that the performance scalability is reduced for more than 2048 cores. This happens due to the physical limit of active blocks per multiprocessor on the GPU. Given that Block Size (number of threads per block) we selected is 32 and total number of Multiprocessor in GTX 480 card is 15, we reach the limit of full concurrency when launching a total of 3840 threads (i.e. simulating as many cores).

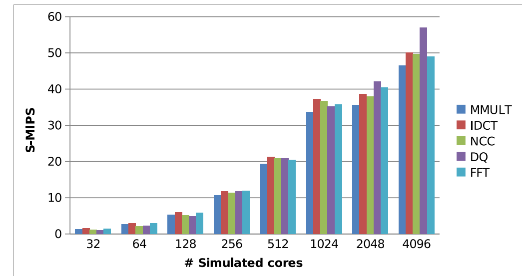


Fig. 12: Benchmarks performance - Architecture 2

In Fig.12 we show the performance of **Architecture2**. In this case, an average of 50 S-MIPS performance is achieved for 4096 cores simulation. The performance scalability is reduced after 1024 cores, due to the physical limit on available shared memory per Multiprocessor on Nvidia GTX 480 card. Due to higher shared memory required for the simulation we could only run 3 blocks per multiprocessor.

Kernel	Scaled up dataset (Arch 1)	#instr. (ARM, Arch1)	Scaled up dataset (Arch 2)	#instr. (ARM,Arch1)
IDCT	8192 DCT blocks(8*8 pixels)	17,813,586	4096 DCT blocks(8*8 pixels)	89069184
DQ	8192 DCT blocks(8*8 pixels)	1,294,903	4096 DCT blocks(8*8 pixels)	20719328
MM	(8192x100)*(100x100)	12,916,049,728	(4096x100)*(100x100)	6458025792
NCC	8192 parallel rows	12,954,417,184	4096 parallel rows	6405371744
FFT Cooley-Turkey	(Datasize =8192)	5,689,173,216	(Datasize = 4096)	2844638432

TABLE II: Benchmarks scaled-up datasets

Since the block size used is 32, on 15 multiprocessors we reach a maximum peak of concurrency for 1440 threads (or simulated cores).

C. Comparison with OVPSim

In this section we compare the performance of our simulator with OVPsim (Open Virtual Platforms simulator)[32], a popular, easy to use, and fast instruction-accurate simulator for single and multicore platforms. We used the OVP simulator model, similar to our **Architecture 1** which essentially has the ISS model but no cache or interconnect model.

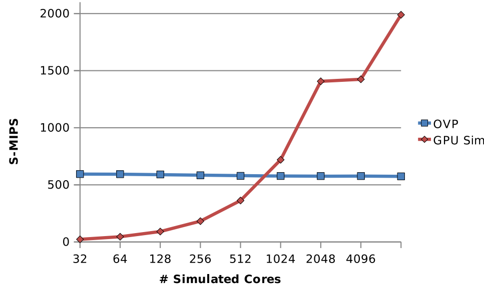


Fig. 13: OVP vs our simulation approach - Dhrystone

We ran two of the benchmarks provided by OVP suite, namely Dhrystone and Fibonacci. As we can see in Fig.13, the performance of OVP remains almost constant increasing the number of simulated cores, while the performance of our GPU-based simulator increases almost linearly. For the Dhrystone benchmark (see Fig.13), we modeled 64K of SPM per node and could simulate up to 4096 cores. Beyond that we reach the maximum limit on available global memory on the Fermi card.

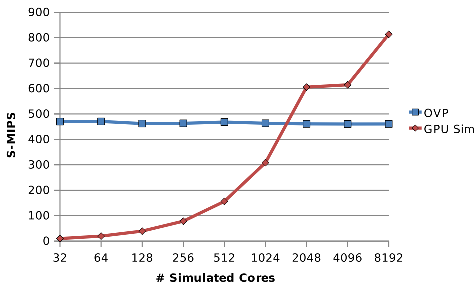


Fig. 14: OVP vs our simulation approach - Fibonacci

Regarding Fibonacci benchmark (see Fig.14) we could simulate up to 8K cores, since 32KB scratchpads are large enough for this benchmark. OVP leverages Just in Time Code Morphing. Target code is directly translated into machine code, which can also be kept in a dictionary acting as a cache. This provides both performance and fast simulation. Our GPU simulator is an Instruction Set Simulator (ISS) and has additional overhead for fetching and decoding each instruction. However, we gain significantly when increasing the number of simulated cores by leveraging the high HW parallelism of the GPU, thus confirming the goodness of our simulator design.

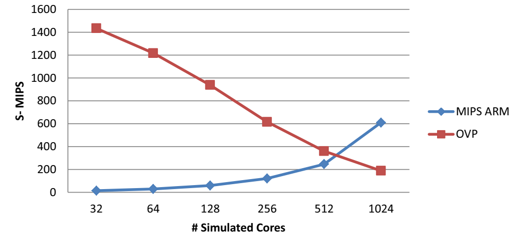


Fig. 15: OVP vs our simulation approach - MMULT

Next, we ran two of our data parallel benchmarks, namely Matrix Multiplication and NCC on OVP and compared them with numbers from our simulator in Fig.16 and Fig.15. As mentioned before these microkernels are equally distributed among the simulated cores in OpenMP style. The OVP performance scales down with increasing the number of simulated cores due to reduction in number of simulated instructions per core. When kernels are distributed among 1024 cores, the instruction dataset per core is very small and code morphing time of single core dominates the simulation run time. On the other hand, as the initialization time for our simulator is very small, we gain in performance when simulating an increasing number of cores in parallel. It is important to note that if the number of instructions performed, i.e. the amount of work undertaken on each core, remains constant the OVP simulation presents a steady performance when increasing the number of simulated cores (it also happens with our first two benchmarks Fig. 13 and 14).

For both Dhrystone and Fibonacci benchmarks, OVP is not able to get performance as high as that in Fig.16 and 15 for small number of simulated cores because these benchmarks contain a high number of function calls, meaning a high number of jump instructions. Each time the target address of a jump points to a not-cached code block, a code morphing

phase is executed. This introduces a high overhead resulting in a consequent loss of performance. Our approach instead, is not affected by the execution path because instructions are fetched and decoded each time they are executed.

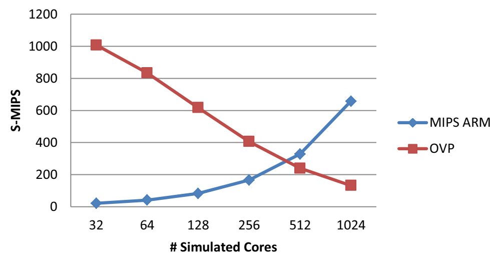


Fig. 16: OVP vs our simulation approach - NCC

VII. Conclusion

In this paper we presented a novel parallel simulation approach that represents an important first step towards the simulation of manycore chips with thousands of cores. Our simulation infrastructure exploits the high computational power and the high parallelism of modern GPGPUs. Our experiments indicate that our approach can scale up to thousand of cores and is capable of delivering fast simulation time and good accuracy. This work highlights important directions in building a comprehensive tool to simulate many-core architectures that might be very helpful for the future research in computer architecture.

Acknowledgement

This research has been partly supported by the Swiss National Science Foundation (SNF), grant number 200021-130048 and within the PRO3D project, and under EUs FP7-ICT grant number 248776.

References

[1] *NVIDIA CUDA Programming Guide*, 2007. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[2] *CUDA: Scalable parallel programming for high-performance scientific computing*, June 2008. [Online]. Available: http://dx.doi.org/10.1109/ISBI.2008.4541126

[3] E. Argollo, *et al.*, "Cotson: infrastructure for full system simulation," *Operating Systems Review*, vol. 43, no. 1, pp. 52–61, 2009.

[4] K. Asanovic, *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[5] N. Beckmann, *et al.*, "Graphite: A Distributed Parallel Simulator for Multicores," MIT, Tech. Rep., November 2009. [Online]. Available: http://dspace.mit.edu/handle/1721.1/49809

[6] N. L. Binkert, *et al.*, "The m5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.

[7] S. Borkar, "Thousand core chips: a technology perspective," in *DAC '07: Proceedings of the 44th annual conference on Design automation*. New York, NY, USA: ACM, 2007, pp. 746–749.

[8] D. Chatterjee, *et al.*, "Event-driven gate-level simulation with gpgpus," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, july 2009, pp. 557 –562.

[9] D. Chiou, *et al.*, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–261. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.16

[10] I. Corp., "Single-chip cloud computer," http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[11] S. Das, *et al.*, "Gtw: A time warp system for shared memory multiprocessors," in *in Proceedings of the 1994 Winter Simulation Conference*, 1994, pp. 1332–1339.

[12] W. R. Davis, *et al.*, "Demystifying 3d ics: The pros and cons of going vertical," *IEEE Design and Test of Computers*, vol. 22, pp. 498–510, 2005.

[13] P. M. Dickens, *et al.*, "A distributed memory lapse: Parallel simulation of message-passing programs," in *In Workshop on Parallel and Distributed Simulation*, 1993, pp. 32–38.

[14] EuroCloud, "http://www.eurocloudserver.com/."

[15] T. Grotker, *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[16] 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.

[17] W. Han, *et al.*, "Using gpu to accelerate cache simulation," in *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, august 2009, pp. 565 –570.

[18] M. Horowitz, "Scaling, power and the future of cmos," *VLSI Design, International Conference on*, vol. 0, p. 23, 2007.

[19] P. S. Magnusson, *et al.*, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[20] M. M. K. Martin, *et al.*, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Computer Architecture News*, 2005.

[21] R. Patti, "Three-dimensional integrated circuits and the future of system-on-chip designs," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1214 –1224, june 2006.

[22] S. Prakash *et al.*, "Mpi-sim: using parallel simulation to evaluate mpi programs," in *WSC '98: Proceedings of the 30th conference on Winter simulation*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 467–474.

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

[24] S. K. Reinhardt, *et al.*, "The wisconsin wind tunnel: Virtual prototyping of parallel computers," in *In Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1993, pp. 48–60.

[25] J. Renau, *et al.*, "SESC simulator," January 2005, http://sesc.sourceforge.net.

[26] M. Ruggiero, *et al.*, "Scalability analysis of evolving SoC interconnect protocols," in *In Int. Symp. on Systems-on-Chip*, 2004, pp. 169–172.

[27] SimNow, "http://developer.amd.com."

[28] J. Steinman, "Breathing time warp," in *PADS '93: Proceedings of the seventh workshop on Parallel and distributed simulation*. New York, NY, USA: ACM, 1993, pp. 109–118.

[29] J. S. Steinman, "Interactive speedes," in *ANSS '91: Proceedings of the 24th annual symposium on Simulation*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, pp. 149–158.

[30] Z. Tan, *et al.*, "Ramp gold: An fpga-based architecture simulator for multiprocessors."

[31] M. B. Taylor, *et al.*, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams," *SIGARCH Comput. Archit. News*, vol. 32, pp. 2–, March 2004. [Online]. Available: http://doi.acm.org/http://doi.acm.org/10.1145/1028176.1006733

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

[33] Tilera, "Tilera-gx processor family," http://www.tilera.com/products/processors/TILE-Gx_Family.

[34] L. Zhao, *et al.*, "Exploring large-scale cmp architectures using manysim," *IEEE Micro*, vol. 27, pp. 21–33, 2007.

[35] G. Zheng, *et al.*, "Bigsim: a parallel simulator for performance prediction of extremely large parallel machines," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, april 2004, p. 78.