Detailed and Highly Parallelizable Cycle-Accurate Network-on-Chip Simulation on GPGPU

Amir Charif, Alexandre Coelho, Nacer-Eddine Zergainoh, Michael Nicolaidis

Univ. Grenoble Alpes, TIMA, F-38000 Grenoble, France

CNRS, TIMA, F-38000 Grenoble, France

{Amir.Charif, Nacer-Eddine.Zergainoh, Alexandre.Coelho, Michael.Nicolaidis}@imag.fr

ASP-DAC, January 16-19, 2017

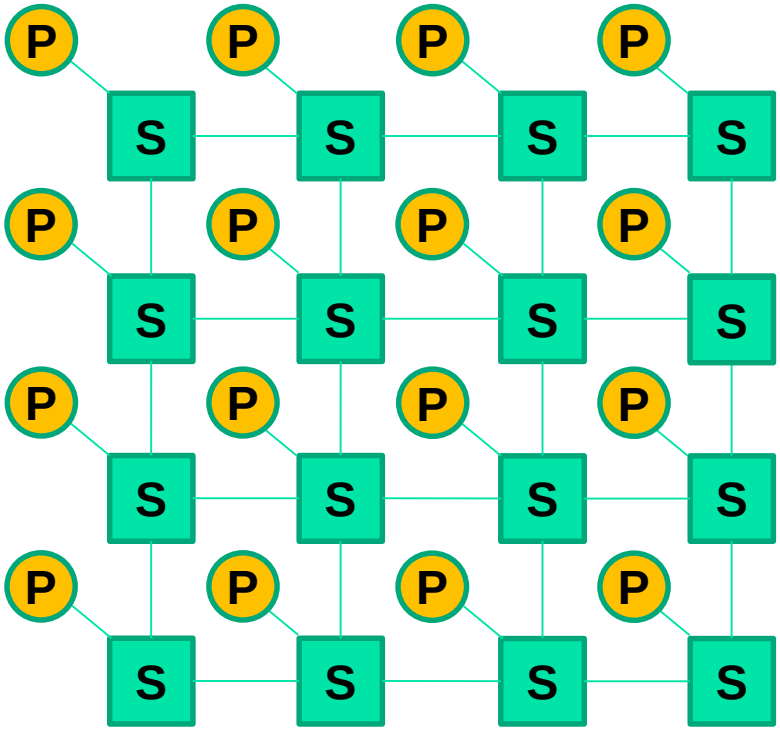
Chiba/Tokyo, Japan

Outline

- Introduction to NoCs and NoC simulation
- Challenges in NoC simulation
- Proposed approach:
  - Task decomposition
  - Implementation details
- Evaluation
- Conclusion and future improvements

Why NoCs ?

- The preferred communication medium for MPSoCs, CMPs, GPUs, …
- Scalable → support large number of nodes
- Actively researched:
  - Performance, Fault-tolerance, congestion, quality of service, …



Evaluating a new NoC Architectures

- Early validation is done by simulation
- Simulating at the RTL is extremely slow !
- Cycle-accurate simulators (booksim, garnet) are faster
  - But not fast enough !

Challenges in NoC Simulation

- A large number of scenarios must be tested
  - Fault scenarios, TSV placement in 3D NoCs, etc.
- NoCs are getting larger (thousands of nodes)
- The larger the NoC, the more iterations are required

→ **Need for fast, accurate and *scalable* simulation for larger NoCs**

Multicore-based Parallel Simulation

- Easy to program
- Portable code, but...
- Speedup is very hard to achieve because of slow synchronization
  - Cycle-accuracy requires one synchronization per cycle
  - Accuracy is often traded off for speed (e.g. HORNET)
- Number of hardware threads still very limited

Graphics Processors: The ideal Platform for Simulating NoCs ?

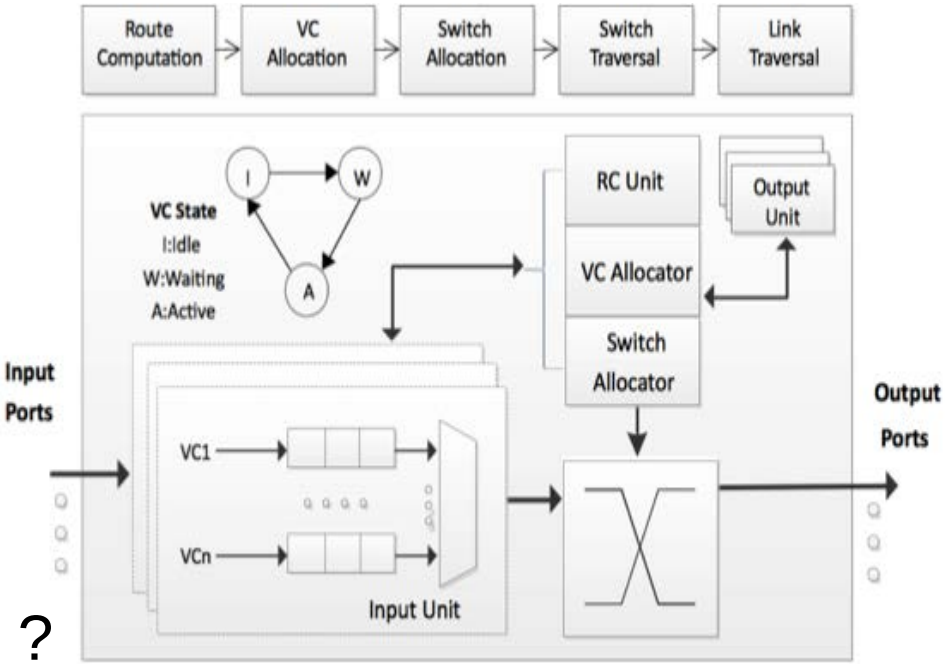
- Widely available
- Good programmability (CUDA, OpenCL)
- Very high compute capability
- Huge number of threads to compensate for slow synchronization
- But…
- Many architectural challenges: Memory access, thread divergence, etc.

Parallel simulation - Key challenges: Task decomposition

- How to split the simulation into independent tasks ?

- 1 Router = 1 task ? [1]
  - OK for CPUs, too coarse-grained for GPUs

- 1 IO port = 1 task ? [2]
  - How about centralized modules ? (e.g. VC Arbiter)
  - How about user extensions ?



[1] M. Eggenberger and M. Radetzki, NoCS 2013.
[2] M. Zolghadr, K. Mirhosseini, S. Gorgin and A. Nayebi, MEMOCODE 2011.

Parallel Simulation – Key Challenges: Thread Divergence and Memory Usage

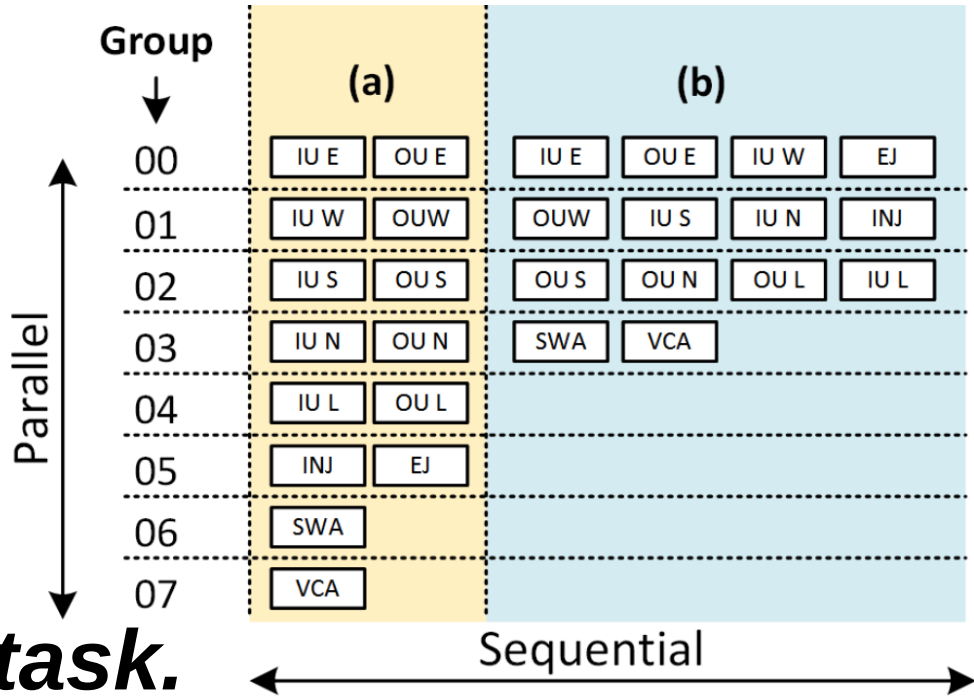
- How to map the tasks to GPU threads ?
  - Different tasks may execute different code
  - Every W (warp size) threads are executed following SIMT (Single Instruction Multiple Threads)
  - Divergent code is serialized within a warp
- How to optimize memory usage ?
  - Accessing global memory (VRAM) is relatively slow
  - A limited amount of fast on-chip memory (Shared memory) is available

Our Approach: Preliminaries

- Modular design to allow easy extension
- Modules can be executed in parallel within one cycle
- Each module is uniquely identified by its router ID and its module ID
  - E.G. if 3 = Switch Allocator, then (12, 3) is the switch allocator of router 12.
- Global function **`execModule(r,m,c)`** executes module (r, m) at cycle c.

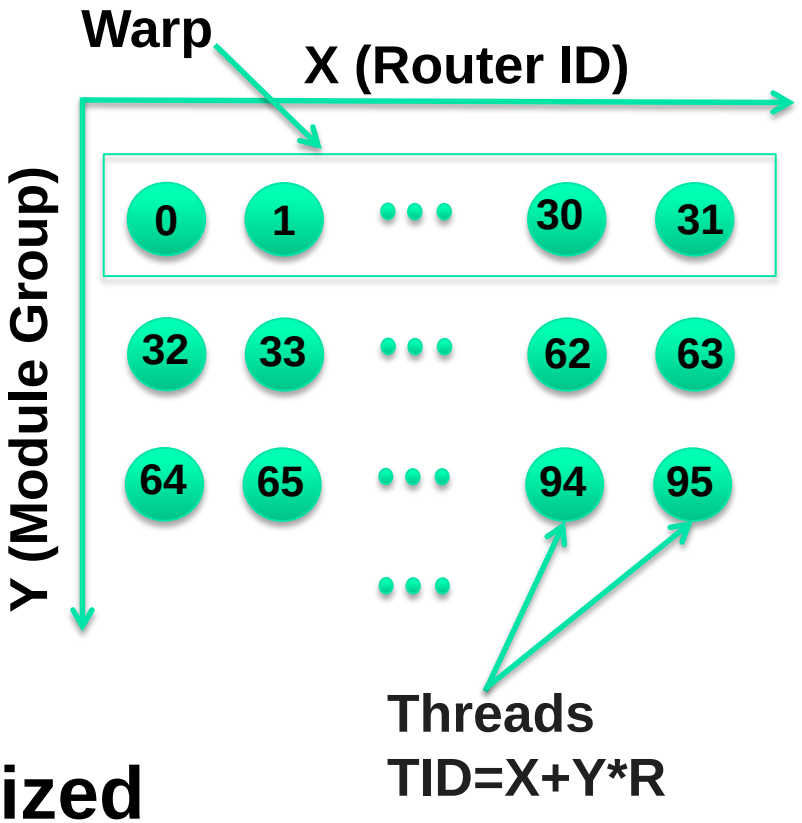
Our Approach: Task Definition

- Modules are grouped into « module groups »
- The modules within each module group are executed sequentially
- Makes it possible to control the level of parallelism (number of threads)
- ***1 module group = 1 task.***



Our Approach: Task Mapping

- Identify each thread using two coordinates **`(x,y)`** such that **`TID=x+y*R`** where **`R`** is the number of routers.
- Map task **`(r,g)`** to thread **`(r,g)`**
- The module group (code to execute) only changes every **`R`** threads
- If **`R`** is a multiple of the warp size **`W=32`**, every warp executes a single module group



→ **Thread divergence is minimized**

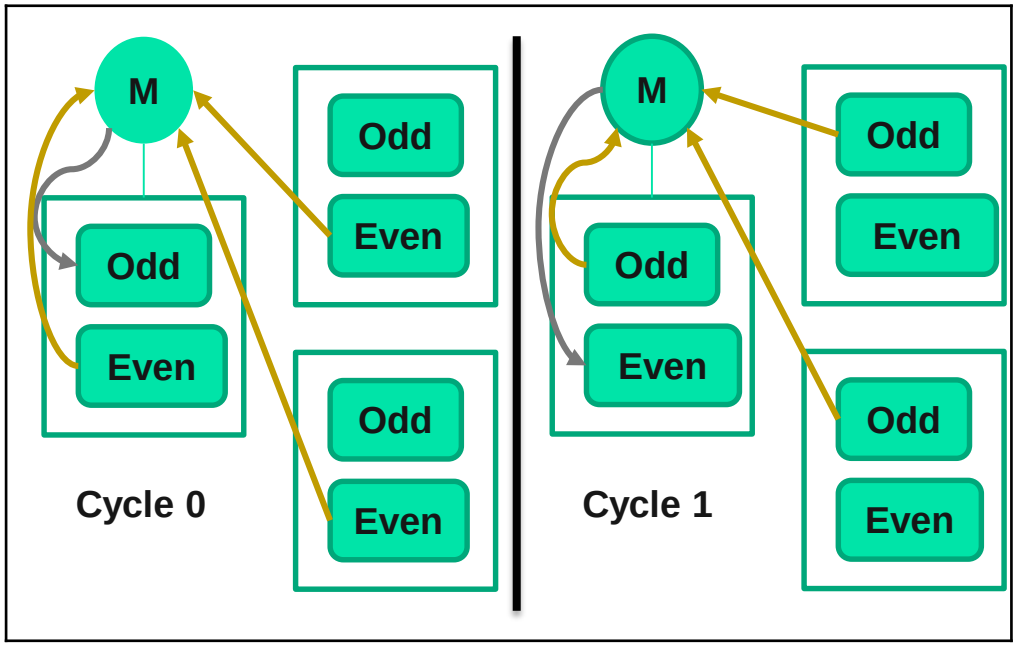
Implementation Details:
Achieving Hardware Fidelity

- Values written in cycle **T** must not be visible before **T+1**
- Modules must be executable in parallel without using atomic operations
- Solution: Register ownership
- Every module owns a set of Registers
- A register stores two values (odd/even) [1]
- On odd cycles, read from odd and write to even
- On even cycles, read from even and write to odd
- Modules only write to registers they own, but can read any module's registers

[1] M. Eggenberger and M. Radetzki, NoCS 2013.

Implementing Register Ownership

```
#define reg_read(reg,parity) (reg)[(parity)]
#define reg_write(reg,v, parity) (reg)[!(parity)]=v
struct MyModule {
    byte my_first_register[2];
    byte my_second_register[2];
    int  my_local_counter;
};
__global__ void kernel(…) {
    int64_t cycle = 0;
    bool parity = 0;
    while (continue_simulation){
        …
        execTask(threadIdx.x, threadIdx.y, parity,…);
        cycle++;
        parity = cycle & 1;
        synchronize(threadIdx, cycle * NumBlocks);
    }
}
```



Example Module: Credit Manager 1/2

- Owns one register per Virtual Channel representing the number of free buffers in the downstream input
- Reads two registers from other modules:
  - Switch allocation result (written by switch allocator in previous cycle)
  - Credit link (written by downstream router in previous cycle)
- If a flit is leaving the router, decrement the credit count for target VC
- If a credit was received, increment the credit count

Example Module: Credit Manager 2/2

```
#define reg_read(reg,parity) (reg)[(parity)]
#define reg_write(reg,v, parity) (reg)[!(parity)]=v
```

```
struct SwitchAllocator {
  …
  unsigned alloc_input[2]; //register
  unsigned alloc_output[2];
  …
};
```

```
struct CreditLink {
  bool valid[2];
  byte vc[2];
};
```

```
struct CreditManager {
    byte credit_count[NVC][2];
};
```

```
__device__ void doCreditManager(CreditManager* cm, SwitchAllocator* sa,
    CreditLink* cl, bool parity) {
        bool credit_valid = reg_read (cl->valid, parity);
        byte credit_vc = reg_read(cl->vc, parity);
        unsigned allocated_vcs = reg_read(sa->alloc_output, parity);
        for (int vc = 0; vc < NVC; vc++) {
                byte credits = reg_read(cm->credit_count[vc], parity);
                if (credit_valid && credit_vc == vc) credits++;
                if (allocated_vcs & (1<<vc)) credits--;
                reg_write(cm->credit_count[vc],credits,parity);
        }
}
```

Our Approach: Per-Cycle Synchronization

- Synchronization among thread blocks is done using a global lock variable similar to [1]
- Thread `0,0` of each block waits for its block to finish current cycle
- Thread `0,0` of all blocks participate to global lock-based synchronization
- Threads of each block wait for thread 0,0 of their block before starting next cycle

```
procedure synchronize(threadID,target){
    let x,y←threadID
    __syncthreads()
    if (x,y=0,0){
        atomically Lock←Lock+1
        while (Lock ≠ target){
        }
    }
    __syncthreads()
}
```

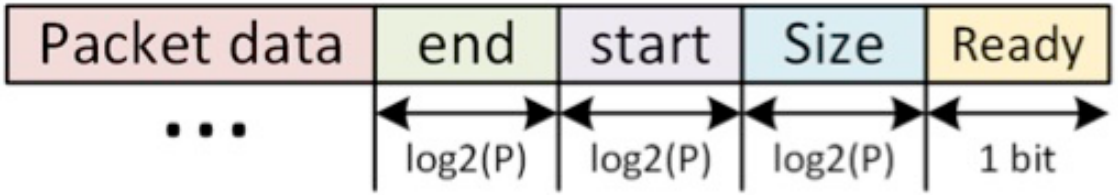
[1] S. Xiao and W. c. Feng, IPDPS 2010.

Memory Usage Optimization: Compact Flit Queues 1/2

- Memory usage usually varies during simulation
- E.G. injection of a new packet in Garnet:
  - `for (int i=0;i<size;i++) flit_t* flit = new flit_t(…);…`
- Usually only header information is useful for simulation
- An input VC FIFO usually stores flits of one unique packet (atomic VC allocation)

Memory Usage Optimization: Compact Flit Queues 2/2

- We propose…



- Size-independent flit queue structure
- Can represent any sequence of flits belonging to the same packet
- Push, Pop and other operations can be implemented by incrementing « end » and « start »
- No allocation required for inserting new flits

→ The amount of traffic does not affect the memory usage of the simulator !

Evaluation

Speed of GPU-based implementation is compared with CPU-base sequential version →

```
void seq_sim(…) {
   int64_t cycle = 0;
   while(continue_simulation) {
     for(int router=0; router<R;router++)
      for(int module=0;module<M;module++)
        execModule(router, module, cycle);
     cycle++;
   }
}
```

- Module code is identical
- CPU: AMD A8-6500 @3.5Ghz, 4.1Ghz Boost clock (good single core performance) -O2
- GPU: NVidia GeForce GTX 980Ti, 22 Stream Multiprocessors (SM), 6GB GDDR5, 96KB Shared Memory / SM.
- Network latency is compared with that obtained with an RTL router implementation (Netmaker)

Speedups and Accuracy

- Excellent speedups
- Higher for larger networks
- High accuracy (close to RTL)

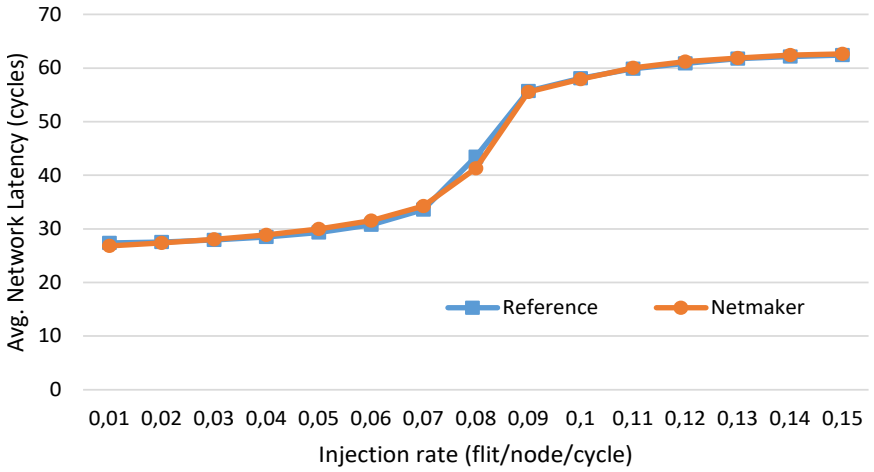


TABLE II
Simulation speedup (1)

16x16 2vc	16x16 4vc	24x24 2vc	24x24 4vc	32x32 2vc	32x32 4vc
26,53	28,60	55,07	59,65	95,53	102,59

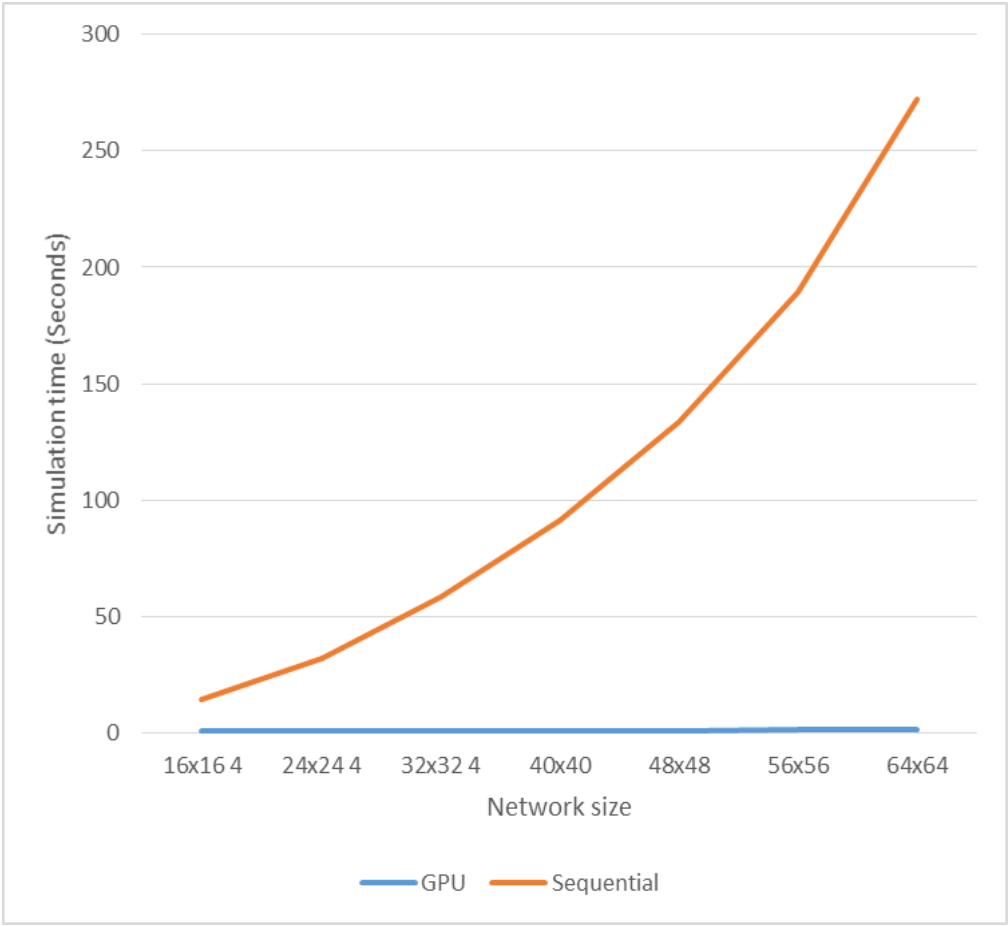
TABLE III
Simulation speedup (2)

# groups	40x40 2vc	40x40 4vc	48x48 2vc	48x48 4vc
7	139,92	146,75	190,55	193,37
# groups	**56x56 2vc**	**56x56 4vc**	**64x64 2vc**	**64x64 4vc**
4	210,33	182,03*	277,27	252,90*

(*) Injectors were evicted from shared memory.

Scalability

- Simulation time doubles (0,5 second to 1 second) but remains very small
- Use of more SMs vs. Longer synchronization
- Reflects high scalability



Currently Implemented Features

- VC router with separable allocators and credit-based flow control
- Topologies
  - 2D, 3D, and stacked Meshes with partial vertical connections
  - A variety of deterministic and adaptive routing algorithms
  - All the commonly used synthetic traffic patterns (Uniform, Hotspot, Transpose, Bit Complement, Shuffle, Bit Reversal)
- Trace-based simulation
  - Dependency-driven trace simulation with Netrace (PARSEC)
  - Support for checkpointing and fast-forwarding

Future Improvements

- Power/Area estimation with ORION3.0
- Temperature estimation using HotSpot
- Custom Fault model for realistic fault simulation
- Source code to be available online

Conclusion

- A parallel simulator design was presented
  - Modular, highly extensible, highly parallelizable
- GPU-specific implementation challenges were addressed
  - Task mapping, synchronization, memory usage
- Ultra-fast NoC simulation on GPU with high hardware fidelity was achieved

amir.charif @imag.fr
http://tima.imag.fr/tima/en/index.html

ご清聴ありがとうございました

THANK YOU FOR YOUR ATTENTION !