Architectures for Packet Processing Applications: the GPU case

Fulvio Risso
Politecnico di Torino
Our reference scenario: packet processing apps

- Compelling demand for more processing capabilities
  - Much more compelling than in other fields of general computing
  - Always increasing network speed
    - 10Gbps and more
  - More intelligence in network devices
    - Necessity to provide added-value in the network
    - E.g., Application-layer processing (e.g. deep packet inspection)
    - E.g., Content-centric networks
    - ...

- Power consumption is a big issue
  - Not covered in this presentation
General computing applications ...

- Computational intensive
- Fairly limited amount of memory accesses
  - More ALU-based instructions than LOAD/STORE
  - Often, program requires a small amount of data (e.g., variables)
  - Furthermore, data typically exploits some locality pattern
    - E.g., Access to consecutive members of the same structure, access to consecutive cells of an array, ...
- Do we really need more speed here?
... and Packet processing apps (1)

- Memory intensive (data-driven applications)
  - Frequent data load from packet
  - Frequent data load/store from other memories, where we store structures for the required algorithms
    - E.g., trees for address lookup, finite state automata for regex, ...
  - Huge amount of data involved in the processing
    - Tables can be as large as several GB

- Traditional apps
  - Few accesses to the memory
  - Usually CPU bounded, not memory bounded
... and Packet processing apps (2)

- No data locality
  - Unpredictable loads from different memory areas
  - Algorithm may have “random” access patterns over large memory structures (e.g. Finite State Automata)
  - We may consume a huge amount of memory bandwidth in case of caches
    - A cache loads the entire line (64B on x86) even if only a few bytes are requested, therefore consuming a lot of memory bandwidth

- Traditional apps
  - High data locality
    - If we load member $X$ of struct $S$, we will probably load also member $Y$ soon
... and Packet processing apps (3)

- **Small tasks, over a large number of packets**
  - E.g., IP address lookup
  - Billions of packets (most of them independent)

- **Traditional apps**
  - Very complex tasks, to be executed once on a given data set
    - Usually CPU bounded, not memory bounded
And what about the hardware?

- How do we support the new requirements coming from the application world?

- Do we have to engineer different platforms for different application domains?
  - Let’s analyze processing and data requirements for different application domains
General computing CPUs...

• Single memory, with caching
  – Can accommodate the small amount of data required by the program
  – Hides memory latency by presenting most requested data to the CPU
  – Long cache lines (e.g., 64 bytes) are appropriate in case of data locality

• CPU optimized for few threads, high performance per thread
  – Large amount of transistors dedicate to ALUs
  – Higher CPU frequencies
  – Maximizes instruction-level parallelism
    • Pipeline
    • Superscalar
    • Out-of-order execution
    • Branch prediction
    • Speculative loads
  – Are many cores useful here?
... and packet processing CPUs

- Multiple memories, with limited amount of caches
  - Caches not very effective
  - “Narrow” data buses (sparse accesses of a few bytes each time)
  - Multiple data buses (multiple access paths toward the memory)
  - Memory hierarchies (local to the core, global, etc)
- CPU optimized for many threads, low performance per thread
  - Maximize thread-level parallelism
  - CPUs made of simple executing components
    - Multicore, systolic arrays
    - Massive amount of processing units
  - More cores, more throughput
A few more words on memory latency

• One of the key points if we want to increase speed
• CPUs can stall if data is not available

• Some numbers
  – DRAM: 60ns, SRAM 1ns
  – Do not confuse access time with throughput, which is usually rather high
    • Throughput: frequency * channels * bus width
    • Example: DDR3-1333, 3 channels, 64 bits bus: 32GB/s
  – A CPU that has a memory access per instruction runs at a virtual speed of 16.7MHz (independently from its actual clock)

• Shall we create “asynchronous code”?
How do we hide memory latency?

- **Traditional CPU**
  - Instruction prefetching
  - Data/instruction caches
  - Out-of order instructions
  - Branch prediction
  - Speculative loads
  - Miss Information/Status Holding Register (MSHR)
    - Missing memory reference is stored in a MSHR, and the following instructions are executed instead

- **Packet processing CPUs**
  - Massive number of threads
  - Zero-overhead thread switching
    - If a thread stalls because of a data miss, another thread is scheduled instead
    - Scheduling done automatically by the hardware, with no overhead
    - No modifications required from the programmer
  - It’s a sort of (automatic) “asynchronous code”
The best processing architecture is...

- **Intel hardware!**
- Not for technical merits, but thanks to scale economies
  - Data centers use Intel (right now)
  - Most users use Intel
- Custom architectures are justified only in case of very specific needs
  - Extremely costly
  - Do not use latest manufacturing process (e.g. 25nm)
  - Difficult to evolve
  - Difficult to engineer (billions of transistors)
- We should look carefully at the solutions that will come here in the next few years
Emerging trends in new general purpose CPUs...

- SISD model “obsolete” (Single Instruction Single Data)
  - No longer competition for GigaHertz
- MIMD model (Multiple Instruction Multiple Data) or SPMD (Single Process...)
  - Even smartphones have many cores
- SIMD model (Single Instruction Multiple Data)
  - Vector instructions
- SIMT mode (Single Instruction Multiple Threads)

- At the end, it’s a matter of parallelism
  - Instruction-level parallelism
  - SIMD (vector) parallelism
  - Multi-core parallelism

- Nothing about memory (right now)
... and some available products

Probably, future processors will be heterogeneous (e.g., include different cores with different execution models, SIMD vs. MIMD) and will offer complex, application-manageable memory hierarchies, data placement and task allocation.
How can we exploit parallelism in packet processing?

• New algorithms in order to process data over multiple cores
  – E.g., regular expressions (NFA)
  – Reduced latency

• Pipeline processing

• Distribute independent packets over different cores
  – Necessity to identify concurrency hazards
Let’s concentrate on GPUs

- Emerging processing paradigm
  - First proposed for graphic-rich experience, but used in other fields as well
- GPU main points
  - Slow cores, but massive parallelism
    - Support limited divergent paths
  - High memory bandwidth
    - Latency may be high
GPU internals: overview

- Streaming Multiprocessor (e.g., 27)
- Scalar Processors (e.g., 8)
- Shared Memory (e.g., 16KB)
- Device Memory (e.g., 1GB)
GPU: Execution vs Control logic

- Limited amount of control logic
  - Simple instructions, limited amount of caches
- Many elementary execution units
GPU internals: details
Execution model: SIMT

- **Single Instruction Multiple Threads**
  - All threads execute the same instruction
    - Single instruction can control multiple processing elements
    - Each thread has its own registers
  - Support for diverging execution paths
    - Predicate execution in case of limited divergence
    - Threads executed in sequence otherwise
      - Done automatically
  - Much more flexible than SIMD

```plaintext
ins1
ins2
if
  ins3
  ins4
else
  ins5
  ins6
ins7
...
```

Memory

No write back
**Memory**

- Different memories
  - Speed, latency, visibility from different Scalar Processors
- Limited amount of caches (not used in our approach)
- All allow random access, but...
  - No penalties in shared memory
  - Automatically coalesced accesses in device memory (if possible)
    - Trouble in case truly random access
- Device memory latency hidden by:
  - Coalesced accesses (less accesses $\rightarrow$ less latency)
  - Instructions dominate over the number of memory accesses
  - New threads are automatically scheduled when a thread is waiting for data
Coalesced accesses

1 Coalesced accesses  2 Coalesced accesses  Sequential accesses
CUDA

• Compute Unified Device Architecture
  – Support for different hardware

• Handles parallelism transparently
  – Programmer does not need to know the number of physical multiprocessors/scalar processors

• Handles memory access transparently
  – Access to a single memory cell from different threads
  – Memory coalescing
Threads, blocks and the hardware

Threads associated with Block 1

Threads associated with Block 2

Device Memory
Thread, Blocks, Grid

- **Thread**
  - Elementary execution unit

- **Block**
  - Group of threads than have the same shared memory
  - Different blocks can share data only in the Device Memory
  - All execute the same program, but their instruction pointers are independent
  - No simple way to communicate

- **Grid**
  - Group of blocks (mainly logical)
  - All blocks execute the same program
Programming model (1)

• Hardware supports:
  – Threads, scalar processors, multiprocessors

• Software defines:
  – Threads, blocks, (grids)

• Programmer instantiates:
  – Blocks: programs executed on different data, on different multiprocessors
  – Threads: programs executed on “similar” data, on the same multiprocessor
  – Many copies of the same program executed in parallel
Programming model (2)

- Automatic mapping of the software model on the hardware
  - Threads in a block will reside on the same scalar processor
  - We may have N software threads and M hardware threads (M<N)
    - Required in order to hide the latency of the Device Memory
- Depends on:
  - Number and availability of physical devices
  - Threads/blocks waiting for data
- No cost for thread switching
GPU kernel execution

- Four steps
  1. the DMA controller transfers data from host (CPU) memory to device (GPU) memory
  2. a host program instructs the GPU to launch the kernel
  3. the GPU executes threads in parallel
  4. the DMA controller transfers resulting data from device memory to host memory

- Some of those phases can be executed concurrently (e.g., DMA transfer while processing previous batch)

- Kernel: program executed on each core
  - Unfortunate name chosen by nVidia; it has nothing to do with OS kernels
Conclusions

• Do we really need dedicated processing architectures?
• Most needs can be satisfied with Intel hardware
  – Or... let’s try so squeeze every bit out of them
• The future will see a convergence of many processing models
• GPUs may be very interesting platforms for some packet processing algorithms
• Need to rewrite the algorithms from this platform
• Another challenge for the future: vector processing