FPGA vs. Multi-Core CPUs vs. GPUs: Hands-on Experience with a Sorting Application

Cristian Grozea[1] *, Zorana Bankovic[2], and Pavel Laskov[3]

[1] Fraunhofer Institute FIRST,
Kekulestrasse 7, 12489 Berlin, Germany
cristian.grozea@first.fraunhofer.de
[2] ETSI Telecomunicación, Technical University of Madrid,
Av. Complutense 30, 28040 Madrid, Spain
zorana@die.upm.es
[3] Wilhelm Schickard Institute for Computer Science
University of Tuebingen, Sand 1, 72076 Tuebingen
pavel.laskov@uni-tuebingen.de

Abstract. Currently there are several interesting alternatives for low-cost high-performance computing. We report here our experiences with an N-gram extraction and sorting problem, originated in the design of a real-time network intrusion detection system. We have considered FPGAs, multi-core CPUs in symmetric multi-CPU machines and GPUs and have created implementations for each of these platforms. After carefully comparing the advantages and disadvantages of each we have decided to go forward with the implementation written for multi-core CPUs. Arguments for and against each platform are presented – corresponding to our hands-on experience – that we intend to be useful in helping with the selection of the hardware acceleration solutions for new projects.

Key words: parallel sort, FPGA, GPU, CUDA, multi-core, OpenMP, VHDL

1 Introduction

Low-cost high-performance computing is a recent development that brings computing power equivalent to former supercomputers to ordinary desktops used by programmers and researchers. Harnessing this power, however, is non-trivial, even with a growing availability of tools for facilitating the transition from traditional architectures. Successful use of possibilities offered by modern parallel architectures is still largely application-dependent and more often than not necessitates rethinking of programming paradigms and re-design of software.

In this contribution, we describe the experience of a practical transition to multi-core architecture in a specific application that requires high performance and low latency - real-time network intrusion detection. The goal of a network

* corresponding author

intrusion detection system (IDS) is to detect malicious activity, e.g. buffer overflow attacks or web application exploits, in incoming traffic. Both performance and latency are of crucial importance for this application if decisions must be made in real time whether or not to allow packets to be forwarded to their destination.

An anomaly-based network IDS ReMIND [3] developed in our laboratory is designed for detection of novel, previously unseen attacks. Unlike traditional. signature-based IDS which look for specific exploit patterns in packet content, our IDS detects packets with highly suspicious content. The crucial component of our detection algorithms is finding of matching subsequences in packet content, a problem that requires efficient algorithms for sorting such subsequences.

The goal of the project described in this contribution was to accelerate existing sequence comparison algorithms (see e.g. [26, 18] for details) to work at a typical speed of an Ethernet link of 1 Gbit/s by using parallel architectures.

There exist at least four alternatives in this field: FPGA devices and various boards using them (ranging from prototyping boards to FPGA based accelerators, sometimes general purpose, sometimes specialized) [29]; multi-core CPUs, with 2, 3, 4 cores or more [10]; many-core GPUs, with 64,128, 240 or even more cores [25]; the Cell processor [13, 32]. The former three solutions were considered and implemented in the course of the current study.

Let us now describe the specific setting of the problem for which an acceleration was sought. The ReMIND system receives packets from a network interface and, after some preprocessing, transforms them into byte strings containing application-layer payload. For a decision to be made, a set of N-grams (substrings of length N) must be extracted from each string (the values of N that work best are in typically in the range of 4 to 8), and compared to the set of N-grams in the prototype; the latter set has been previously extracted from payload of normal packets. The comparison essentially amounts to computing the intersection of two sets of N-grams. An efficient linear-time solution to this problem involves lexicographic sorting of all N-grams in the incoming string (linear-time, low constants). The sorting component takes a string and is supposed to return some representation of sorted N-grams in this string, for examples an index set containing the positions of the respective N-grams, in a sorted order, in the original string[4]. The incoming strings can be up to 1480 bytes long (maximal length of an Ethernet frame), and to achieve 1 Gbit/s speed, approximately 84,000 full-length packets must be handled per second. Processing of single packets can be assumed independent from each other since, in the simplest case, decisions are made independently on each packet.

We will now proceed with the description of the particular methods and implementations followed by the presentation and discussion of experimental results.

[4] Returning a sorted set of N-grams itself blows up the size of the data by a factor of up to N.

2 Methods

The following hardware platforms were available for our implementation:

- FPGA: Xilinx evaluation board ML507 equipped with a Virtex-5 family XC5VFX70T FPGA, maximum clock frequency 0.55 GHz, 70 thousand logic cells.
- CPUs: Dell Precision T7400 with two Xeon 5472 quad-core, clock speed 3GHz and 16GB of RAM.
- GPUs: Two Nvidia Quadro FX 5600, with 1.5 GB of RAM and 128 v1.0 CUDA shaders each - the clock of the shaders: 1.35 GHz.

The details of a hardware design and/or algorithms for each platform are presented below.

2.1 FPGA

The ML507 board used offers a PCIe 1x connection to the hosting PC, and the setup was to capture data from network on the PC, send data over the PCIe to the FPGA board, sort it there, and send the results back to the PC. All communication with the board had to be implemented as DMA transfers, for efficiency reasons and in order to overlap communication and processing both on the PC side and on the FPGA side. This requirement proved to be very difficult to fullfill as the support of Xilinx did not include the sources of a complete application + driver solution, just a performance demo with some of the components only in binary form [4].

Let L be the length of the list to sort. It is interesting to note that the FPGA could do all comparisons between the elements of an L-long sequence in $O(1)$ time complexity and $O(L^2)$ space complexity, by specifying a comparator for every pair of elements in the list.

When one targets efficiency on a serial or multi-core CPU implementation, a good order and good constant algorithm is chosen and this usually solves the problem. When working with FPGAs, the things are more complicated. A smart but more complex algorithm can solve the problem in less steps, but the maximum clock speed usable depends on the complexity of the design/algorithm implemented, so the net effect of using a smarter algorithm can be of slowing down the implementation. This is exactly what has happened to us.

The complex sort-merge sorting This algorithm was specialized in sorting $L = 1024$ elements lists. In a first stage, the incoming data is grouped into groups of 8 items, sorted in $O(1)$ with the extensive comparators network described above. Every 4 such groups are merged with a 4-way merging network producing a 32-elements group. The 32 groups of 32 elements each are repeatedly merged with a tree of parallel 4-way merging nodes, which outputs on its root in sorted sequence the 1024 items. Despite the clever design and the tight and professional VHDL coding of this algorithm, it went past the resources of the FPGA chip

we used. After a synthesis process (the rough equivalent for FPGA of compiling to object files) that took two days (not unheard of in the FPGA world), the numbers reported for resources usage were much beyond the available resources: 235% LUTs, 132% block RAMs, max frequency 60MHz. While the maximum frequency was above the 50MHz needed with this design, everything else was beyond physical limits. Only a bigger FPGA chip could have accommodated the design – and indeed it did fit (if only barely) on the biggest FPGA of the Virtex-5 family, the FX200T. But, as we didn't have this much more expensive one, we had to look for possible alternatives.

The bitonic sort Batcher's bitonic sort algorithm is an early algorithm [5] that has implementations on most types of parallel hardware. Its time complexity is $O((log_2 L)^2)$, its space complexity is $O(L(log_2 L)^2)$, as it consists in $(log_2 L)^2$ stages of $L/2$ comparators. While the time complexity was maybe acceptable for our problem (although for L as low as 1480, $(log_2 L)^2$ is only about 10 times smaller than L), the space complexity was not acceptable. We have used the perfect shuffling technique of Stone [30] to collapse all these stages to a single one through which the data is looped $(log_2 L)^2$ times. But, the shuffling circuits where still using too much of the FPGA area (more than available), so we considered one more option.

The insertion sort The parallel insertion sort algorithm is very simple: the sequence to sort is given element by element. At every step all stored elements equal to or bigger than the current input element shift replacing their right neighbor, leaving thus an empty place for the insertion of the current input element.

We implement the needed comparison and conditional shifting by creating a "smart cell" that can store one element and does the right thing when presented with an input. The whole parallel insertion sort is implemented as a chain of such smart cells, as long as the maximum length of the list to sort, and with appropriate connections.

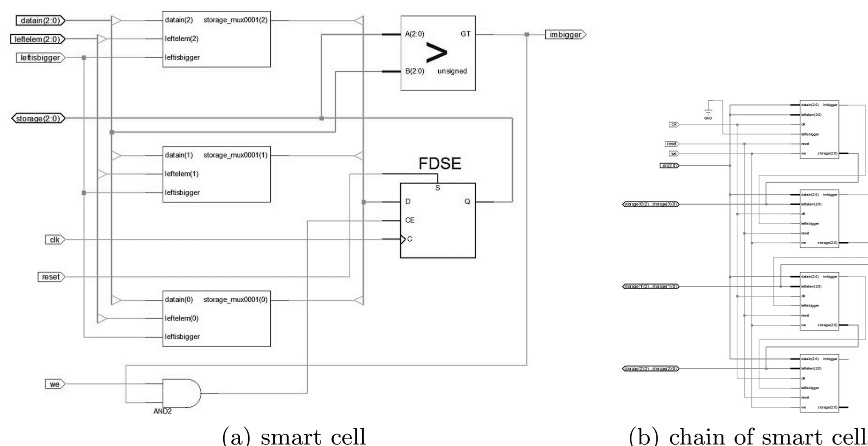
The circuit for a "smart cell" able to process elements of three bits in size is given in Figure 1(a). The circuit for a full chain of 4 smart cells that can sort lists of length 4 of 3-bit elements is shown in Figure 1(b).

Please note that the time complexity of one step is O(1), i.e. constant, but the amount of resources needed (comparators, multiplexers, gates) is directly proportional to the length of the list to sort. Overall, the time complexity is O(L), the space complexity is O(L) as well.

The VHDL code for this implementation is given in Appendix 1. It uses the construction "generate" to create multiple instances of "smart cells" and to connect appropriately their pins.

2.2 Multi-core CPUs

We have used OpenMP, which makes parallelizing serial programs in C/C++ and Fortran fairly easy [9]. A fragment from the implementation of an early bench-



(a) smart cell (b) chain of smart cells

Fig. 1. (a) The circuits of a smart cell used in the parallel insertion sort (here for 3 bit elements); (b) Parallel insertion sort as a structure composed of many smart cells (here for processing 4 elements of 3 bits each).

mark is given in Appendix 2. As one can see in the code we provide, we don't try to split the individual sorting tasks, but to dispatch them to multiple threads (in groups of 16) that get dispatched on the cores and on the CPUs. While for forking threads and splitting the workload of a *for loop* to those threads OpenMP is very easy to use, for the anomaly detection system we needed a producer-consumer setup, where the network data acquisition would acquire data and send it in some load-balancing fashion to the worker threads. Producer-consumer setups are notably difficult to implement in OpenMP, because of limited locking and signaling mechanisms, mainly because it lacks condition variables. Therefore, while we kept using OpenMP, we have added message_queue from the open-source library *boost* to implement the queues needed for the producer-consumer paradigm.

2.3 GPUs

Our Nvidia GPUs are programmable with CUDA [24, 20], which combines C/C++ on the host side with C-like kernels that run in parallel on the cores of the GPU. A C++ STL like framework on top of CUDA is the open-source library *Thrust* [1]. We have used this library to implement our test programs. As a result these are remarkably concise, as can be see in Appendix 3.

3 Results and Analysis

The performances obtained have been: FPGA – processing 1.875 Gbit/s, communication 1 Gbit/s; multi-core CPU – 2 Gbit/s overall; GPU – 8 Mbit/s overall, including processing and communication.

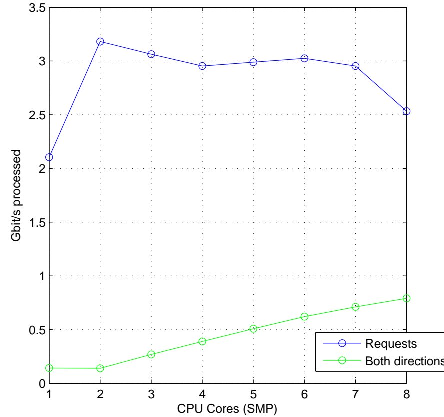


Fig. 2. The performance of the multicore solution: the highest volume of network traffic that can be processed as a function of the number of cores used.

The FPGA communication implemented achieved 2 Gbit/s with Bus-Master DMA transfers (approx. 1 Gbit/s in each direction). The parallel insertion sort was the only one fitting into our FPGA chip, when restricted to sorting of 256 64bits elements (8-byte-grams) or sorting 512 48bits elements (6-byte-grams). It did so for a maximum clock speed of 240MHz. This was more than the 128MBytes/s needed to process a full-speed 1 Gbit line. Still, retrieving data from FPGA board requires more than 8 bits of output per input byte, when there are more than 256 elements to sort. This would then require more than 2 Gbit/s communication speed between the CPU and the FPGA board. The communication constraints and the difficulty to adapt sorting to FPGA led us to investigate the alternatives.

The multi-core CPU implementation achieved from the first tests 2 Gbit/s. The results of the complete network intrusion detection system prototype, where time is spent also on tasks other than the N-gram extraction and sorting are shown in Figure 2, where the numbers are estimated offline by running on previously captured network traffic. Added to the graph is a curve “both directions” which models the worst-case traffic, completely unbalanced such that the requests are much longer than the replies. Please note that in general (for example for the HTTP protocol), the requests are short and the replies longer. We have also tested the prototype on a quad-core QuickPath Interconnect-enabled Intel E5540@2.53GHZ machine; the QPI architecture did not lead to a supplementary acceleration, probably because our solution doesn’t require much synchronization or data passing from one core to another.

The GPU solution had the latency so high that only about 1% of the desired speed has been obtained (1000 sorts/second of 1024 element long lists). Our profiling linked most of this latency to the memory transfers between the memory

space of the CPU and that of the GPU. While the radix sorting in *Thrust* has been reportedly outperformed [19], speeding it up will not have a big impact on the total time. For this reason we decided that it makes little sense to investigate further the GPU alternative as a possible improvement over the multi-core CPU version. The GPUs are a better fit for sorting large vectors.

4 Related Work

Previous work on sorting on FPGA include generating automatically optimal sorting networks by using quickly reconfigurable FPGAs as evaluator for a genetic programming system [17], with a follow-up where the genetic search is also done on the FPGA [16]. Human designed sorting networks were published e.g. in [12] (where the authors write that "The results show that, for sorting, FPGA technology may not be the best processor choice") and [21]. A recent paper where the tradeoffs involved in implementing sorting methods on FPGA are carefully considered is [7].

The state-of-the-art GPU sorting algorithms in CUDA (radix sort, merge sort) are explained in [27], and are included in the open-source library CUDPP[28] starting with version 1.1.

Most GPU application papers compare to a CPU implementation, although most often with a single core one - reporting in this way the best speed-up values. In [8] the GPU, FPGA and multi-core CPU implementations for solving three problems (Gaussian Elimination, DES - Data Encryption Standard and Needleman-Wunsch) are compared. Unfortunately, although the GPU used was the same family with the one we used, and the same holds true for the CPU, the authors used a much older FPGA (Virtex II Pro, maximum frequency 100MHz), which could have biased the results. Another paper comparing all three platforms we tested (and supplementarily a massively parallel processor array, Ambric AM2000) is [31] where various methods for generating random numbers with uniform, Gaussian and exponential distributions have been implemented and benchmarked. Another interesting factor is introduced in the comparison, the power efficiency (performance divided by power consumption) and here FPGAs were the leaders with a big margin. A very extensive overview of the state of the art in heterogeneous computing (including GPU, Cell BEA, FPGA, and SIMD-enabled multi-core CPU) is given in [6]. The obvious conclusion one gets after surveying the existing literature is that there is no clear winner for all problems. Every platform has a specific advantage. What is most interesting, beyond the price of the devices – which is fairly balanced for the devices we used – is how much progress is one researcher expected to make when entering these low-cost HPC technologies fields on the particular problem of interest, and this depends most on how easy is to develop for these platforms.

5 Discussion and Conclusion

5.1 Comparing the difficulty of programming and debugging

As far as the FPGA is concerned, it can be configured in the language VHDL in two styles: “behavioral” (corresponding to the procedural style of CPU programming) and “structural” (one abstraction layer lower). If in the beginning the ”behavioral” style might look appealing, after hitting the area limitation hard barrier one is supposed to grasp such definitions “FDSE is a single D-type flip-flop with data (D), clock enable (CE), and synchronous set (S) inputs and data output (Q)” [2] and work almost exclusively in a structural fashion, where the allocation of the very limited resources is more under the control of the designer and not of the compiler – so previous experience with digital electronics helps. In other words, “programming” FPGAs is not really for regular computer programmers, as the programming there is actually describing a structure, which only sometimes can be done by describing its desired behavior. If the general purpose logic cells are to be saved, then explicit modules on the FPGA such as DSP units and memory blocks have to be referenced and used and this shifts the competencies needed even more towards the field of digital electronics and further away from the one of computer science.

Some of the FPGA tools are overly slow. We mentioned that it took us two days to synthesize the most complex sorting algorithm we created for FPGA – by using Xilinx ISE 10. Switching to Synplify reduced the synthesis time to 50 minutes. Still a lot by software engineering practice (software compilation rarely takes that long). Even worse, while alternative tools can cover the chip-independent stages in the FPGA workflow, the chip-dependent stages like map, place and route can be done usually only with the vendor's software tools – and these stages are slower than the synthesis.

We think FPGA is still an interesting platform, for being energy efficient. It is unlikely that the speed issues with the FPGA workflow tools can be solved completely, as the problems they try to solve are NP-complete (resource allocation, place & route)[33]. This issue gets worse with the size of the FPGA. While our FPGA drifts towards entry-level ones, bigger ones could have been worse for this reason.

As far as the multi-core CPUs are concerned, programming them can be done with standard compilers – newest versions of the main C/C++ compilers like GNU gcc and Intel's one have all OpenMP support. Adding parallelism to loops is fairly easy in OpenMP. The possibility to implementing a correct and complete producer-consumer setup is unfortunately considered outside of the scope of the OpenMP framework. Debugging and tuning OpenMP multi-threaded code is not as easy as for the serial code, of course, but it's not overly difficult, especially with the aid of tracing tools like VampirTrace [22] used by us. The entry level barrier is not too high, but having parallel programming knowledge helps – especially understanding race conditions and the synchronization mechanisms.

One of the main problems with programming the GPUs is the instability of the frameworks and the fact that most are vendor-specific. OpenCL [23] may

change this in time, but for now we have used CUDA, which is the natural choice for Nvidia graphic chips. CUDA itself has now reached its third major version in two years, following the advances of the hardware. While CUDA is suitable for developing code, debugging code is much more complicated. There is an emulated device mode which turns to some extent debugging CUDA into debugging many CPU-side threads, but it is much slower than the non-emulated mode. Apart from the difficulty to debug code, another criticism to the CUDA framework was that it is (still) too low-level, making the implementation and the tuning of complex systems overly difficult. Systems like *Thrust*, PyCUDA [15] and BSGP [14] aim to fix this. The need to transfer data between the CPU's memory and GPU's memory is also a major disadvantage of the GPU when used as a computing coprocessor, as these transfers introduce undesirable latencies. In CUDA, for a limited set of devices, which share the memory with the CPU (and are thus not top performing ones), page-locked host memory can be mapped to the GPU memory space, reducing these latencies. On the other hand, the dedicated GPU memory is higher speed (at least bandwidth-wise), so this is just a trade-off with outcomes to be tested case by case. While knowing C/C++ is enough to start CUDA programming, getting good performance requires leveraging the hardware primitives/structures meant for graphics (e.g. texture memory) - the compilers do not do this for the user, so having experience with graphics programming does still help.

5.2 Conclusion

To conclude, FPGA is the most flexible but least accessible, GPU comes next, very powerful but less flexible, difficult to debug and requiring data transfers which increase the latency, then comes the CPU which might sometimes be too slow despite multiple cores and multiple CPUs, but is the easiest to approach. In our case the multi-core implementation offered us the best combination of compatibility, high bandwidth and low latency, therefore we have selected this solution for integration into the ReMIND prototype.

References

1. Thrust, `http://code.google.com/thrust`.
2. Xilinx FDSE, `http://www.xilinx.com/itp/xilinx7/books/data/docs/s3esc/s3esc0081_72.html`.
3. Project ReMIND, `http://www.remind-ids.org`, 2007.
4. Xilinx application note XAPP1052 (v1.1), `www.xilinx.com/support/documentation/application_notes/xapp1052.pdf`, 2008.
5. KE Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.
6. A.R. BRODTKORB, C. DYKEN, T.R. HAGEN, J. HJELMERVIK, and O.O. STORAASLI. STATE-OF-THE-ART IN HETEROGENEOUS COMPUTING (draft, accepted for publication). *Journal of Scientific Programming.*

7. R.D. Chamberlain and N. Ganesan. Sorting on architecturally diverse computer systems. In *Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 39–46. ACM, 2009.
8. S. Che, J. Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Symposium on Application Specific Processors*, 2008.
9. L. Dagum and R. Menon. Open MP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.
10. Jack Dongarra, Dennis Gannon, Geoffrey Fox, and Ken Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, February 2007.
11. C. Grozea, C. Gehl, and M. Popescu. ENCOPLOT: Pairwise Sequence Matching in Linear Time Applied to Plagiarism Detection. In *3rd PAN WORKSHOP. UNCOVERING PLAGIARISM, AUTHORSHIP AND SOCIAL SOFTWARE MISUSE*, page 10.
12. J. Harkins, T. El-Ghazawi, E. El-Araby, and M. Huang. Performance of sorting algorithms on the SRC 6 reconfigurable computer. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology*, pages 295–296.
13. H.P. Hofstee. Power efficient processor architecture and the Cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262. San Francisco, CA, 2005.
14. Q. Hou, K. Zhou, and B. Guo. BSGP: bulk-synchronous GPU programming. In *ACM SIGGRAPH 2008 papers*, page 19. ACM, 2008.
15. A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, A.D. Sarma, D. Nanongkai, G. Pandurangan, P. Tetali, et al. PyCUDA: GPU Run-Time Code Generation for High-Performance Computing. *Arxiv preprint arXiv:0911.3456*, 2009.
16. J. KORRENEK and L. SEKANINA. Intrinsic evolution of sorting networks: A novel complete hardware implementation for FPGAs. *Lecture notes in computer science*, pages 46–55.
17. JR Koza, FH Bennett III, JL Hutchings, SL Bade, MA Keane, and D. Andre. Evolving sorting networks using genetic programming and the rapidlyreconfigurable Xilinx 6216 field-programmable gate array. In *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems & Computers*, volume 1, 1997.
18. T. Krueger, C. Gehl, K. Rieck, and P. Laskov. An Architecture for Inline Anomaly Detection. In *Proceedings of the 2008 European Conference on Computer Network Defense*, pages 11–18. IEEE Computer Society, 2008.
19. N. Leischner, V. Osipov, and P. Sanders. GPU sample sort. *Arxiv preprint arXiv:0909.5649*, 2009.
20. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, pages 39–55, 2008.
21. J. Martinez, R. Cumplido, and C. Feregrino. An FPGA-based parallel sorting architecture for the Burrows Wheeler transform. In *ReConFig 2005. International Conference on Reconfigurable Computing and FPGAs, 2005.*, page 7, 2005.
22. M.S. Muller, A. Knupfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W.E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. *Proceedings of the Minisymposium on Scalability and Usability of HPC Programming Tools at PARCO 2007, to appear*, 2007.
23. A. Munshi. The OpenCL specification version 1.0. *Khronos OpenCL Working Group*, 2009.

24. J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. 2008.
25. J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. *PROCEEDINGS-IEEE*, 96(5):879, 2008.
26. K. Rieck and P. Laskov. Language models for detection of unknown attacks in network traffic. *Journal in Computer Virology*, 2(4):243–256, 2007.
27. N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing-Volume 00*, pages 1–10. IEEE Computer Society, 2009.
28. S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, page 106. Eurographics Association, 2007.
29. M.C. Smith, J.S. Vetter, and S.R. Alam. Scientific computing beyond CPUs: FPGA implementations of common scientific kernels. In *Proceedings of the 8th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD05)*. Citeseer.
30. HS Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, 100(20):153–161, 1971.
31. D.B. Thomas, L. Howes, and W. Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72. ACM, 2009.
32. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 9–20. ACM New York, NY, USA, 2006.
33. Y.L. Wu and D. Chang. On the NP-completeness of regular 2-D FPGA routing architectures and a novel solution. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 362–366. IEEE Computer Society Press Los Alamitos, CA, USA, 1994.

Appendix 1: Parallel Insertion Sort in VHDL for FPGA

```
library IEEE;use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;use IEEE.STD_LOGIC_UNSIGNED.ALL;
constant nrbitselem:integer:=3;
subtype elem is STD_LOGIC_VECTOR((nrbitselem-1) downto 0);
subtype elemp1 is STD_LOGIC_VECTOR(nrbitselem downto 0);
type vect is array(natural range<>) of elem;
type vectp1 is array(natural range<>) of elemp1;
entity insertsortsmartcell is
    Port ( datain : in elem;we: in std_logic;
        leftelem: in elem;leftisbigger: in std_logic;
        clk: in std_logic;reset:in std_logic;
        imbigger:buffer std_logic;storage:inout elem);
end insertsortsmartcell;
architecture Behavioral of insertsortsmartcell is begin
  imbigger<='1' when storage>datain else '0';
  process(clk,reset)
  begin if clk'event and clk='1' then
   if reset='1' then storage<=(others=>'1');
   else if we='1' then if imbigger='1' then if leftisbigger='0' then
      storage<=datain; -- insertion right here
      else storage<=leftelem;
     end if;end if;end if;end if;end if;
```

```
end process;end Behavioral;
entity insertsort is
  generic(abw:integer:=2);
    Port ( xin:elem;storage : inout vect(2**abw-1 downto 0);
        clk:std_logic;reset:std_logic;we:std_logic;xout:out elem);
end insertsort;
architecture Behavioral of insertsort is
signal isbigger: std_logic_vector(2**abw-1 downto 0);
begin
  a:for i in 0 to 2**abw-1 generate
    b:if i=0 generate
    cell0:entity work.insertsortsmartcell port map(xin,we,
     xin,'0',clk,reset,isbigger(i),storage(i));
    end generate;
    c:if i>0 generate
    cell:entity work.insertsortsmartcell port map(xin,we,
     storage(i-1),isbigger(i-1),clk,reset,isbigger(i),storage(i));
    end generate;end generate;
  xout<=storage(2**abw-1);
end Behavioral;
```

Appendix 2: OpenMP Benchmark for Sorting

This is a code fragment from a benchmark that proves that it is possible to extract the N-grams and sort those on the multi-core CPUs we used, at a speed higher than 1 Gbit/s. The sorting is virtual in the sense that no data is moved around, just indexes are reordered; full details including code not reproduced here are given in [11]. The OpenMP influence on the code is minimal: a header is included, the number of threads is specified, then through one or two pragmas the tasks are split between threads.

```
#include "omp.h"
#define fr(x,y)for(int x=0;x<y;x++)
omp_set_num_threads(4);//how many threads openmp will use
//fork the threads
#pragma omp parallel private(counters,startpos,ix,ox,v)
{fr(rep,125000/16){
#pragma omp for schedule(static,1)
  fr(rep2,16){
   ... //generate an array, then sort it with serial radix sort
}}}
```

Appendix 3: Benchmark of Sorting on GPU using Thrust

```
//included: <thrust/device_vector.h>, <thrust/host_vector.h>, <thrust/functional.h>, <thrust/sort.h>
int main(void){const int N = 1024;int elements[N] = {1,3,2};
  thrust::host_vector<int> A(elements,elements+N);thrust::device_vector<int> B(N);
  int s=0;for(int rep=0;rep<1000;rep++){
    thrust::copy(A.begin(), A.end(), B.begin());thrust::sorting::radix_sort(B.begin(), B.end());
    thrust::copy(B.begin(), B.end(), A.begin());s=s+A[0]+1;}
std::cout<<s<<" ";return 0;}
```