FPGA: What's in it for a Database?

Rene Mueller Jens Teubner
Systems Group, Department of Computer Science, ETH Zurich
Haldeneggsteig 4, 8092 Zurich, Switzerland
{rene.mueller,jens.teubner}@inf.ethz.ch

ABSTRACT

While there seems to be a general agreement that next years' systems will include *many* processing cores, it is often overlooked that these systems will also include an increasing number of *different* cores (we already see dedicated units for graphics or network processing). Orchestrating the diversity of processing functionality is going to be a major challenge in the upcoming years, be it to optimize for performance or for minimal energy consumption.

We expect *field-programmable gate arrays (FPGAs or "programmable hardware")* to soon play the role of yet another processing unit, found in commodity computers. It is clear that the new resource is going to be too precious to be ignored by database systems, but it is unclear *how* FPGAs could be integrated into a DBMS. With a focus on database use, this tutorial introduces into the emerging technology, demonstrates its potential, but also pinpoints some challenges that need to be addressed before FPGA-accelerated database systems can go mainstream. Attendees will gain an intuition of an FPGA development cycle, receive guidelines for a "good" FPGA design, but also learn the limitations that hardware-implemented database processing faces. Our more high-level ambition is to spur a broader interest in database processing on novel hardware technology.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems; C.5 [**Computer System Implementation**]: VLSI Systems

General Terms

Design

Keywords

FPGA, hardware acceleration, data processing, VLSI

1. INTRODUCTION

The idea of using a domain-specific hardware co-processor to accelerate database processing is not new. The idea of a "database machine" [6] failed in the late 70s: with substantial up-front costs (engineering efforts, money, and time), tailor-made hardware components could no longer keep pace with improvements that general-purpose CPUs saw at the time.

Today, thirty years later, the bottlenecks have shifted. General-purpose CPUs will no longer (significantly) benefit from increasing clock speeds. At the same time, recent technology advances have eliminated the up-front cost for new chip designs almost entirely, making the idea of a database co-processor worth a reconsideration.

In this tutorial, we look at *FPGAs* (*field-programmable gate arrays*), a promising instance of such technology advances. Naively spoken, FPGA chips consist of a number of logic gates whose wiring can be programmed by software (more details below). The programmed chip can then be used, *e.g.*, as a hardware-accelerated implementation for specific compute or control tasks.

FPGAs are already used successfully in image and signal processing, where they excel with *low latency* and a high degree of *parallelism* (hence, throughput). From formerly highly specialized system designs, they are increasingly becoming part of commodity setups. We expect them to soon play a similar role as dedicated network or graphics processors play today: as part of a *heterogeneous multi-core* system, FPGAs will act as one particular processing unit, ready to solve data- or compute-intensive sub-tasks in hardware.

1.1 FPGAs for Database Co-Processing

Our interest here is in the application of FPGAs for *database co-processing.* The data-intensive nature of database tasks makes them a particularly good fit for FPGA-based processing. Streaming databases, *e.g.*, may benefit from the *low latencies* that FPGA implementations can provide even under high load. More traditional systems can use their existing set-oriented query formulations to exploit the high degree of *parallelism* inherent to programmable hardware.

Unfortunately, the potential of FPGAs is still not very widely known. One reason may be that the processing model of FPGAs—and hence the way they are controlled by software—is very different to the traditional von Neumann architecture that computer scientists are used to deal with.

1.2 Purpose of this Tutorial

The main purpose of this tutorial is to give an introduction into FPGA-accelerated database processing. Attendees receive an overview of the technology, some technical background, and—most importantly—serviceable guidelines to judge the applicability of an FPGA-based solution to a given problem, the quality of a solution, or learn how to get started with an own solution.

We do not expect any particular background from tutorial attendees (those with a weak remembrance of their early digital circuit design classes may better appreciate some of

	Virtex-II Pro XC2VP30	Virtex-5 XC5VFX200T
Lookup Tables (LUTs)	27,392	122,880
Flip-Flops (registers)	27,392	122,880
Block RAM (kbit)	2,448	16,416
18-bit Multipliers	136	384
PowerPC Cores	2	2
maximum clock speed (MHz)	≈ 100	≈ 550
release year	2002	2006

Table 1: Selected characteristics of Xilinx FPGAs.

the technical aspects we show, however). Parts of the tutorial will also contain technical material, but attendees may easily skip these parts and still follow the remainder of the tutorial.

1.3 Tutorial Outline and Setup

Our tutorial is organized in four units whose contents we sketch in Sections 2–5. Roughly speaking, we provide the necessary background about the inner workings of an FPGA in order to build one's own FPGA circuits in Unit 1. In Unit 2, we then show how FPGAs can be used to support database tasks, which gives us the background to assess the trade-offs in FPGA-assisted database processing in Unit 3. In Unit 4, we see what work has been done in the field and how we can benefit from that.

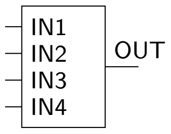
We do not expect attendees to bring experience in FPGA development. Our tutorial includes live demonstrations of an FPGA development cycle to give an intuition of what it takes to build hardware circuits for database tasks.

2. FPGA BASICS

A basic understanding of the hardware internals is important to judge the trade-offs that occur when building an FPGA circuit. In the first unit of this tutorial, we give a very brief overview about the inner workings of the underlying hardware and highlight some particular features which are most relevant in a database context.

2.1 FPGA Building Blocks

The heart of every FPGA chip is a number of *lookup tables (LUTs)*, a programmable logic unit with k inputs and a single output (where k is typically between four and six, see right). A LUT can be programmed to implement *any* Boolean function with k inputs (internally, the LUT uses a 2^k-row lookup table that lists every possible input combination and the desired output). From a pool of lookup tables, in combination with a programmable *interconnect fabric*, arbitrary logic circuits can be created in hardware, based on a specification in software.



The logic part of an FPGA (lookup tables and interconnect fabric) is paired with means to keep *state* during processing. Most chips provide two different types of on-chip memory: *flip-flops* (or *registers*) provide storage for a single bit each and are wired directly into the logic part; memory in larger quantities is provided by distributed *block RAM* (or *BRAM*), configurable RAM units spread across the chip.

Oftentimes, FPGA chips are equipped with additional standard functionality, such as hardware multipliers or even full-fledged CPU cores. In Unit 3 of the tutorial (Section 4), we will use the built-in PowerPC core on Xilinx Virtex chips to connect FPGA-accelerated functionality to traditional CPU-style processing. In Table 1, we listed some characteristics of two FPGA chips sold by Xilinx (a major FPGA vendor).

2.2 Strengths and Limitations

All available resources can be programmed and used in an extremely flexible manner. Later in this tutorial, we are going to emphasize the inherent *parallelism* in FPGAs, the possibility to operate circuits in an *asynchronous* mode of execution, and the use of FPGAs as *content-addressable memories* for highly efficient key-value lookups.

Limits to this flexibility are set by the availability of each type of resources. The most apparent limitation is the number of lookup tables of a given FPGA chip, which can limit the complexity of the logic circuit that can be programmed to it. Other potential limits include memory requirements, the availability of additional in-silicon functionality, or the bandwidth of the interconnect fabric.

In comparison with general-purpose CPUs, FPGAs typically operate at significantly lower clock frequencies (around 100 MHz). This has very positive effects when it comes to the consumption of electrical energy, where FPGAs excel over CPUs by up to two orders of magnitude. On the flip side, there is a risk of an inferior execution speed. Consequent use of parallelism and asynchronous execution, however, typically more than compensates for the reduced clock frequency.

2.3 Programming FPGAs

The specification of a user logic is described using a domain-specific programming language, a *hardware description language* (HDL), the most popular ones being *VHDL* and *Verilog*. Given the HDL description of a logic component, software tools can *synthesize* the design into logic circuits and load them onto an FPGA chip. As a development aid, circuits can also be *simulated* in software with high accuracy based on their HDL description.

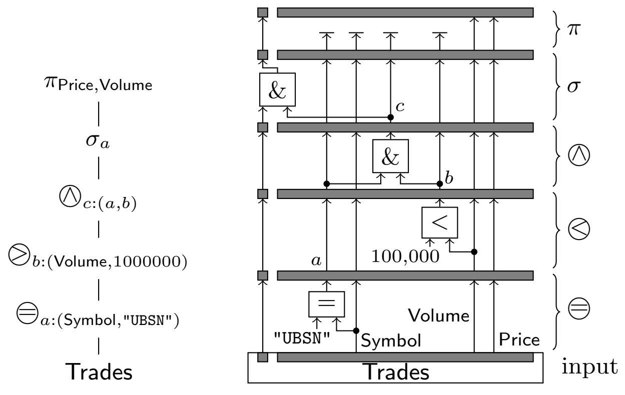
Since HDL code essentially describes the behavior of a hardware circuit, programming HDL is considerably different than writing sequential code in a commodity programming language. In the tutorial, we present VHDL examples that illustrate an FPGA development cycle and we give helpful hints to avoid stepping into typical pitfalls.

3. USING FPGAS

A potential use of FPGAs in a database context is the processing of *data streams*. The data flow in such a system thereby directly translates into the physical wiring on the hardware side.

3.1 Asynchronous Execution

To demonstrate the typical structure of a resulting hardware execution plan, let us consider a stock trading scenario, where we process a stream that contains stock trades. An application is now interested in all trades for UBS stocks



(a) Algebra plan. (b) Compiled hardware plan.

Figure 1: Algebraic representation of Query Q_1 and resulting hardware execution plan following a synchronous execution model.

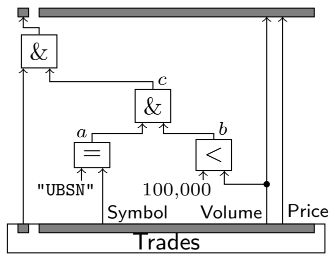


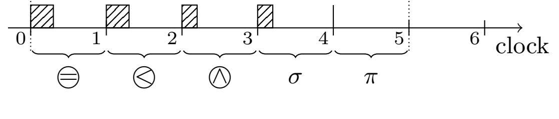
Figure 2: Optimized (asynchronous) hardware execution plan for Query Q_1.

that had a volume greater than 100,000:

```
SELECT Price, Volume
  FROM Trades                                          (Q1)
 WHERE Symbol = "UBSN" AND Volume > 100000 .
```

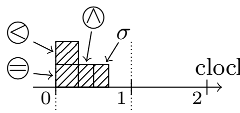
This user query can be translated into a pipelined hardware circuit by compiling its algebraic representation (Figure 1(a)) into logic components operator-by-operator. The hardware circuit resulting from a translation of Query Q_1 is illustrated in Figure 1(b). In this illustration, arrows ↑ indicate wires in the FPGA interconnect fabric and correspond to the data flow in the algebraic plan; square boxes represent logic circuits that implement comparison or a Boolean 'and' operation; and flip-flop registers serve as buffers for intermediate results and are represented as gray-shaded boxes.

The hardware plan in Figure 1(b) follows a strictly *synchronized* execution mode. Every clock cycle, each operator retrieves data from its input registers, processes it, then stores it in its output registers where the data is going to be picked up in the following clock cycle. The *timing diagram* of the circuit thus looks as follows:



Only a fraction of every clock cycle is used to perform actual work (indicated by ▨). After that, each operator waits for synchronization with the clock signal.

This is essentially what CPUs do, too. With dedicated hardware we can do better. Figure 2 illustrates the result of optimizing the plan in Figure 1(b) to operate in an *asynchronous* execution mode. By eliminating intermediate flip-flop registers, all sub-circuits may now operate without any external clock constraints. In the corresponding timing diagram,



,

it is easy to see how this leads to a reduction in execution time from six clock cycles down to a single cycle. Observe also how the optimization also enabled a *parallel* execution mode for the two (independent) predicates on Symbol and Volume.

This example demonstrates how significantly more work can be packed into a single clock cycle in an FPGA-based system, as opposed to general-purpose CPUs with fixed and strictly synchronized instruction sets. In practice, this typically more than compensates for the lower speeds FPGAs operate on.

3.2 Content-Addressable Memory

The main advantage of using FPGAs for data processing is their intrinsic parallelism. Among others, this enables us to escape from the *von Neumann bottleneck* (also called the *memory wall*) that classical computing architectures struggle with. In the common von Neumann model, memory is physically separated from the processing CPU. Data is acquired from memory by sending the location of a piece of data, its *address*, to the RAM chip, then receiving the data back. In FPGAs, flip-flop registers and block RAM are distributed across the chip and tightly wired to the programmable logic. In addition, lookup tables can be re-programmed at runtime and thus be used as additional distributed memory. As such, the on-chip storage resources of the FPGA can be accessed in a truly parallel fashion.

A particular use of this potential is the implementation of *content-addressable memory* (CAM). Other than traditional memory, content-addressable memory can be accessed by data values, rather than by explicit memory addresses. Typically, CAMs are used to resolve a given data item to the address it has been stored at. More generally, however, the functionality can implement an arbitrary key-value store with constant (typically single-cycle) lookup time.

A traditional application domain of content-addressable memories is network processing, where they are used, *e.g.*, to implement routing tables or pattern matching for network firewalls. Network processing units therefore often ship with built-in CAMs. Bandi *et al.* [2] have explored the use of such devices for data stream processing and found content-addressable memories to be a suitable tool to perform frequent item detection on data streams. Other potential applications include all those that would typically depend on hash tables, such as duplicate elimination, aggregation, or joins.

4. FPGAS IN DATABASES SYSTEMS

To make a hardware-accelerated operator implementation accessible to a database system, it has to be wired up to conventional components and connected to, *e.g.*, a general-purpose CPU. To this end, commercially available FPGAs

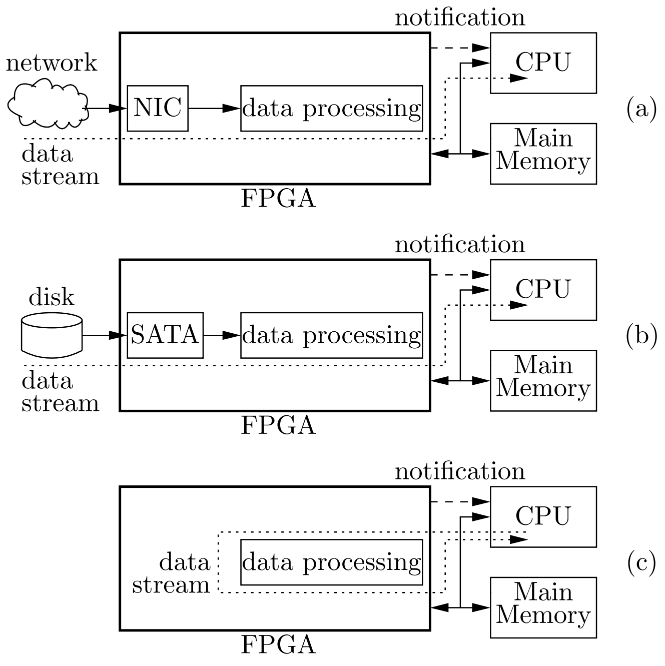


Figure 3: FPGA placements in data path: (a) between network interface and CPU; (b) between disk and CPU; (c) FPGA as co-processor.

often come with additional components (*hard cores*) that are implemented in silicon directly on the chip. Available hard cores may include, for example, on-chip memory or full-fledged CPU cores (cf. Table 1). Additional standard functionality can be programmed onto the chip by means of vendor-provided *soft cores* (such as additional CPUs, memory or NIC controllers). The FPGA chip itself usually comes pre-mounted on a board with additional interfacing components. Boards are available as standalone solutions, to be plugged into PCI Express slots, or even packaged as modules for HyperTransport CPU sockets.

Besides the physical integration of FPGA into a DBMS, the placement of the additional processing unit in a system's *data path* has important implications, and FPGAs offer a number of alternatives here.

Figure 3 illustrates three possible architectures. For latency-sensitive data stream processing operations, the FPGA can be inserted between the network interface (NIC) and the CPU as shown in Figure 3(a). By performing filtering and aggregation on the FPGA, the work for the CPU can be significantly reduced and, hence, the externally applicable load increased. In a short example, we demonstrate how this enables data processing at true network wire speed.

Application data filtering is, in principal, also possible using programmable network cards [21] that provide one or more dedicated processors directly on the card. However, unlike FPGAs, programmable network cards have been designed only for a quite narrow set of applications.

A similar application pattern is shown in Figure 3(b). Here, we assume a disk drive as the data source that is accessed and pre-filtered using an FPGA. The prototypic use case for such a setup are data warehouses, where table dimensionalities and access patterns often render the use of indexes infeasible and data essentially needs to be scanned. Again, the FPGA can help in offloading work from the CPU. The approach in Figure 3(b) was suggested in [7] and very closely resembles the one taken by data warehouse appliances such as the Netezza Performance Server (NPS) system [5].

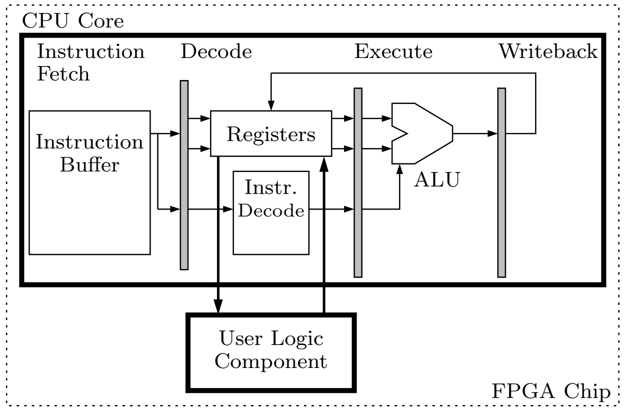


Figure 4: Direct connection of a user logic core to the 4-stage execution pipeline of an embedded soft core CPU.

Figure 3(c) illustrates the FPGA as a co-processor to the CPU. This approach allows the CPU to offload computation-intensive tasks to the FPGA. The FPGA is usually connected through a HyperTransport or PCIe bus. Communication between CPU and FPGA is realized through shared memory and notifications via interrupts. A commercial example that follows a similar approach is the MySQL Analytic Appliance offered by Kickfire [17]. The appliance contains an FPGA mounted on a PCIe card. Next to the FPGA chip, the card also contains a dedicated RAM chip with a high-bandwidth link to the FPGA.

The architecture shown in Figure 3(c) can be easily integrated in a traditional system. However, transferring data to and from the FPGA to main memory (shared with CPUs) can become very expensive. For large data transfers, the FPGA can even aggravate the memory bottleneck of traditional systems. Now, accesses from both CPU and FPGA have to arbitrated. In the tutorial, we outline a possible solution for this problem. It based on an even closer coupling of the custom logic with the CPU. Several embedded CPUs such as the PowerPC 450 (available on Virtex-5 FPGAs) or soft core CPUs [15, 18] allow direct coupling of custom logic to the data path of the execution pipeline. This approach literally extends functionality of the CPU itself. A simplified 4-stage execution pipeline of a soft core CPU is shown in Figure 4. A customized hardware circuit (user logic) is directly hooked into this pipeline and can provide accelerated implementations of performance-critical functionality. The custom functions can be accessed from CPU programs either through user-defined instructions or specific control instructions to interface between the CPU and the user core.

In summary, FPGAs provide a number of alternatives to be hooked into a database system. In the tutorial, we discuss the trade-offs that incur with such alternatives and provide an intuition for the best integration approach to solve a given database task.

5. EXISTING RESEARCH

FPGAs per se have received only little attention in the database community so far. In several ways, however, programming FPGAs resembles the way how modern, vector-oriented processors are used to accelerate database-typical tasks. The parallelism offered by FPGAs is very similar to the execution mode of SIMD architectures or graphics processors (GPUs). Thus, many solutions proposed for such architectures can straightforwardly be converted into circuits for FPGAs.

In particular, this includes the work about database processing in graphics processors. Govindaraju *et al.* [12] use the parallelism in GPUs to perform standard database operations. Bandi *et al.* [3] looked into GPU support for spacial queries. By actually rendering geometric shapes into the GPU memory, the GPU serves as a pre-processing filter for spacial queries. In [11], Govindaraju *et al.* used similar hardware to implement sorting efficiently. Their approach is similar to a recent work by Chhugani *et al.* [4], who exploit vectorized processing to implement sorting networks on a commodity CPU.

The Cell Broadband Engine explicitly favors the use of vector primitives by means of dedicated processing units (*synergistic processing units, SPUs*) with SIMD support. This has been used to accelerate various database tasks, such as sorting [8], join processing [9], or stream processing [19].

The use of SIMD instructions of commodity CPUs has been considered for database processing by Zhou *et al.* [20]. Recently, Johnson *et al.* [16] described how SIMD functionality can be used to evaluate multiple predicates within a single CPU instruction.

On the other end, a lot of work has been done in the hardware community, and a number of techniques have been developed from which an FPGA-accelerated database engine can directly benefit. Most importantly, this affects the field of *reconfigurable computing*. FPGAs can arbitrarily be reprogrammed at runtime and a number of systems have been built to perform this task intelligently and efficiently.

Here, we particularly want to point to the *PipeRench* project [10], a system that has been designed to support workloads with a stream-like data flow (such as database workloads typically are). If necessary, PipeRench will swap in and out parts of a hardware circuit if the available hardware resources are not sufficient to solve a given task.

In projects like Kiwi [13] or Liquid Metal [14], work is currently underway to use FPGAs as accelerators for general-purpose programming languages. There certainly is a significant overlap with database processing here, though database systems will likely be able to much more benefit from their strictly set-oriented query formulations.

Hardware implementations for data mining problems have also been proposed. For example, Baker *et al.* [1] present a implementation of the Apriori algorithm on FPGAs based on *Systolic Arrays*. Items are streamed through a linear array of processing elements, each implemented as a circuit on the FPGA. Initially, frequent 1-item sets are inserted at the input of the array. The candidate generation, pruning and the computation of the support are performed in the units as the items stream through the array. At the output, the support information is extracted and fed back into the input. This process repeats until the final candidate set is found. Each array element contains memory that stores the candidates whose support is currently being processed.

6. ABOUT THE AUTHORS

Both authors are actively working on FPGA-accelerated database processing in the context of the *Avalanche* project. The Systems Group at ETH is involved in a larger industry collaboration, where we currently build a stream processing engine with extremely low latency at substantial throughput rates.

René Müller. After an undergraduate degree in electrical engineering, René Müller obtained a MSc in computer science from ETH Zurich. Since 2006, he is a PhD student at ETH Zurich, working on embedded data processing and wireless sensor networks. In his previous work, he developed *SwissQM*, a virtual machine-based stream processing platform for sensor networks.

Jens Teubner. Graduated with a PhD from TU München in 2006, Jens Teubner worked at the IBM T. J. Watson lab from 2007–2008. Since 2008, he is a postdoc at ETH Zurich, working on hardware-accelerated data processing. Most of his earlier work revolved around scalable XML processing. He was a co-founder of the *Pathfinder* XQuery compiler project.

7. REFERENCES

[1] Zachary K. Baker and Viktor K. Prasanna. Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs. In *Proc. of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, 2005.

[2] Nagender Bandi, Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Data Stream Algorithms using Associative Memories. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 247–256, Beijing, China, June 2007.

[3] Nagender Bandi, Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Processing Spacial Data Using Graphics Processors. In *Proc. of the Int'l Conference on Very Large Databases (VLDB)*, Toronto, ON, Canada, 2004.

[4] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU architecture. *Journal of the VLDB Endowment/Proc. VLDB 2008*, 1(2):1313–1324, 2008.

[5] Netezza Corp. `http://www.netezza.com/`.

[6] David DeWitt. DIRECT—A Multiprocessor Organization for Supporting Relational Database Management Systems. *IEEE Trans. on Computers*, c-28(6), June 1979.

[7] Mark Franklin, Roger Chamberlain, Michael Henrichs, Berkley Shands, and Jason White. An Architecture for Fast Processing of Large Unstructured Data Sets. In *Proc. of the IEEE Int'l Conference on Computer Design*, pages 280–287, 2004.

[8] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellSort: High Performance Sorting on the Cell Processor. In *Int'l Conference on Very Large Databases (VLDB)*, pages 1286–1297, Vienna, Austria, September 2007.

[9] Buğra Gedik, Philip S. Yu, and Rajesh Bordawekar. Executing Stream Joins on the Cell Processor. In

Proc. of the Int'l Conference on Very Large Databases, pages 363–374, Vienna, Austria, 2007.

[10] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[11] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 325–336, Chicago, IL, USA, June 2006.

[12] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast Computation of Database Operations using Graphics Processors. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, Paris, France, June 2004.

[13] David Greaves and Satnam Singh. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[14] Shan Shan Huang, Amir Hormati, David Bacon, and Rodric Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *European Conference on Object-Oriented Programming*, Paphos, Cyprus, July 2008.

[15] Xilinx Inc. *MicroBlaze Processor Reference Guide.* `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`.

[16] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-Wise Parallel Predicate Evaluation. *Journal of the VLDB Endowment/Proc. VLDB 2008*, 1(1):622–634, 2008.

[17] Kickfire. `http://www.kickfire.com`.

[18] Richard Neil Pittman, Nathaniel Lee Lynch, and Alessandro Forin. eMIPS: A Dynamically Extensible Processor. Technical Report MSR-TR-2006-143, Microsoft Research Redmond, October 2006.

[19] Dina Thomas, Rajesh Bordawekar, Charu C. Aggarwal, and Philip S. Yu. On Efficient Query Processing of Stream Counts on the Cell Processor. In *Proc. of the Int'l Conference on Data Engineering (ICDE)*, Shanghai, China, March 2009.

[20] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 145–156, Madison, WI, USA, June 2002.

[21] Netronome Flow Engine Acceleration Cards, White Paper. `http://www.netronome.com`.