J Sign Process Syst (2016) 85:307–323
DOI 10.1007/s11265-015-1057-6

Measuring and Modeling the Power Consumption of Energy-Efficient FPGA Coprocessors for GEMM and FFT

Heiner Giefers[1] · Raphael Polig[1] · Christoph Hagleitner[1]

Received: 9 November 2014 / Revised: 18 June 2015 / Accepted: 30 September 2015 / Published online: 15 October 2015

Abstract In this paper we analyze the power consumption and energy efficiency of general matrix-matrix multiplication (GEMM) and Fast Fourier Transform (FFT) implemented as streaming applications for an FPGA-based coprocessor card. The power consumption is measured with internal voltage sensors and the power draw is broken down onto the systems components in order to classify the energy consumed by the processor cores, the memory, the I/O links and the FPGA card. We present an abstract model that allows for estimating the power consumption of FPGA accelerators on the system level and validate the model using the measured kernels. The performance and energy consumption is compared against optimized multi-threaded software running on the POWER7 host CPUs. Our experimental results show that the accelerator can improve the energy efficiency by an order of magnitude when the computations can be undertaken in a fixed point format. Using floating point data, the gain in energy-efficiency was measured as up to 30 % for the double precision GEMM accelerator and up to 5× for a 1k complex FFT.

✉ Heiner Giefers
hgi@zurich.ibm.com

Raphael Polig
pol@zurich.ibm.com

Christoph Hagleitner
hle@zurich.ibm.com

[1] IBM Research – Zurich, Ruschlikon, Switzerland

Keywords FPGA GEMM accelerator architecture · FPGA FFT accelerator architecture · FPGA stream processing · Energy efficiency analysis · CPU/FPGA power measurements · Hybrid CPU/FPGA system · Power profiling · Power estimation

1 Introduction

Recent trends in data analytics and ever growing data repositories rise the need for efficient high performance computing (HPC) solutions. Although chip vendors are keeping up to adhere to their multi-core roadmaps, large scale HPC systems are facing major challenges![11]. Big data is expanding faster than Moore's Law [25] and thus even massively parallel algorithms that can efficiently spread out over large-scale supercomputers will encounter limitations. Future HPC systems must reduce their energy requirements while keeping on to deliver more processing performance. One well-established approach to improve the efficiency of computer systems is to apply special purpose coprocessors, commonly referred to as hardware accelerators. All of the top ten systems on the November 2014 edition of the Green500 list exhaustively use accelerator cards to to increase their floating point performance. As a rule of thumb, the energy efficiency of an accelerator is proportional to its degree of specialization and thus, application specific integrated circuits (ASICs) are normally more efficient than software programmable accelerators [15]. In an ASIC the functional units are tailored to a specific problem and the overhead of fetching and orchestrating instructions is eliminated However, problem-specific accelerators are of limited value for HPC systems that are being designed for widespread use. Field programmable gate arrays (FPGAs) have been effectively deployed in HPC systems and they promise to deliver a good trade-off between performance and programmability. Several studies reveal that FPGA-based accelerators can improve the energy efficiency compared to CPU and even GPU solutions [12, 20, 27, 29]. However, used in a hybrid environment, the energy consumed for computations on the FPGA device is only a part of the energy budget spend for solving the compute problem. Host memory accesses, data movement and accelerator-support tasks running on the CPU can contribute a substantial amount of energy. A detailed analysis of the system is required to understand the sources of power consumption and to improve the energy efficiency.

In this paper we study the performance, power and energy of two widely applied linear algebra operations on FPGA and CPU. The general matrix-matrix multiplication (GEMM) is a fundamental routine applied as a core-function in many numerical methods. Many important HPC applications can be reduced to dense matrix operations so that GEMM codes are typically highly tuned for specific CPUs and, moreover, CPUs are designed to efficiently support this type of operation [3, 6, 22].

The Fast Fourier Transform (FFT) is one of the most important computational kernels used in applications domains such as physics, engineering, applied mathematics, and signal processing [4]. The FFT greatly reduces the complexity of computing the Discrete Fourier Transform (DFT). However, the lower operation count comes at the price of more non-local, strided data accesses. Modern computer systems typically suffer from this kind of access patterns so that FFT computations are limited by data movement rather than by the arithmetic operations.

In case of the GEMM kernel both, the CPU and the FPGA can easily explore the inherent data-level parallelism and the comparison demonstrates the costs for the actual multiply-accumulate operations on the two devices. The peak performance for floating point operations is almost an order of magnitude higher on the CPU and thus, the FPGA is hardly able to accelerate matrix operations for which optimized software implementations exist. Taking into account that the idle power consumption can easily exceed 70 % of the power consumed during high utilization, the best performing solution tends to be also the most energy-efficient one. From this observation one could conclude that the *race-to-completion* policy is always the best choice in terms of not only performance but also energy. However, this conclusion does not hold if we assume that a server node is operating on various workloads simultaneously. Our results show that floating point arithmetic can be more energy-efficient on the FPGA than on a mature server system if only the power increase compared to an idle baseline power is considered.

The use of an FPGA based coprocessor is particularly advantageous for workloads that do not map well to the CPU. Multi-threading approaches diminishing returns when the overhead costs outweigh the computational benefits of parallel FFT execution. By applying dedicated processing pipelines that efficiently implement the strided access patterns inside the FPGA fabric we can explore a high degree of parallelism at full I/O bandwidth. The main contributions of this paper are summarized as follows:

- We propose and explore streaming based architectures for GEMM and FFT on FPGA and present implementations for a custom FPGA accelerator card. The architectures are highly configurable and can be adapted to various data formats and problems sizes.
- A power model is introduced that allows for estimating the FPGA power consumption at a high-level. We verify the model based on actual measurements and demonstrate the feasibility of providing proper predictions in very early stages of the design process.
- We study the power consumption of the hardware architecture and provide a detailed power breakdown of the system's components that are actively used for computation.
- We compare the energy efficiency of the FPGA based implementations against software versions from mature linear algebra and FFT libraries tuned for the host system. Our results show that double precision GEMM (DGEMM) computations on the FPGA can improve the energy efficiency by 29 % in specific cases. The energy efficiency of 1D FFTs is improved by up to 17.8× when a custom data format is used and 1.2× to 5× when a standard floating point format is applied.

2 Compute Platform

All experiments were performed on a PowerLinux 7R2 server running Fedora 17. The dual 8-core POWER7 processors operate at 3.55 GHz and are capable of executing 64 threads in parallel. The CPU has 32kB L1 and 256 kB L2 cache per core, all cores within a chip share a 32MB L3 cache.

A self-developed FPGA coprocessor card is attached to the system via the GX++ I/O bus that can sustain a unidirectional data transfer rate of 3.2 GB/s. As shown in Fig. 1, the card is equipped with two FPGAs, both of which are Altera Stratix-IV devices. Using two separate chips helps to ensure a clean separation between system and user content of the accelerator design. The user part of the design runs on a EP4SGX530 device and can access the host's virtual

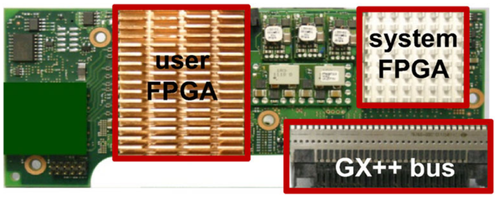


Figure 1 Custom build FPGA coprocessor card for POWER7 systems.

memory space via DMA channels and memory mapped I/O (MMIO) registers . The second FPGA (EP4SE360) implements an MMU that translates virtual addresses through the assistance of privileged software on the host allowing the accelerators to access the main memory and to operate in a common virtual address space with the applications running on the processor.

Figure 2 shows the system organization with hardware resources (FPGA accelerator card and main memory) and the software stack. Whenever a data stream requires to access a cache line mapped to a new memory page, the system FPGA writes the effective address of the request into an MMIO register. As being part of the host application, the translation server continuously reads this register using the device driver of the card and converts the virtual address to a physical address. The MMU on the system FPGA reads the physical address back and uses the page frame for DMA to the host memory.

3 Hardware Implementation of Benchmarking Kernels

This section provides an overview of our selected kernels and their implementation on the FPGA coprocessor.

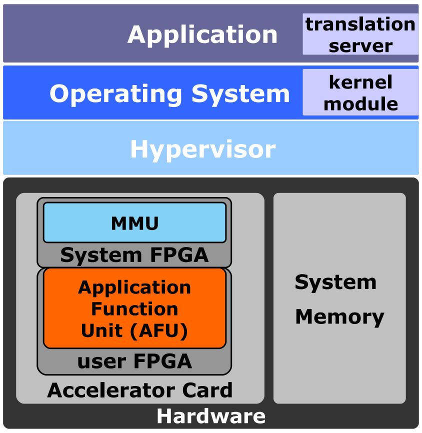


Figure 2 System organization with software managed Direct Virtual Memory Access (DVMA) stack.

3.1 General Matrix-Matrix Multiplication

The Basic Linear Algebra Subprograms (BLAS) is a prevalent collection of elementary linear algebra operations [17]. BLAS routines are the core of many other linear algebra libraries and are greatly used in, e.g., LAPACK, Matlab, and R. The most efficient BLAS implementations are normally provided by chip vendors who tune the operations for their architectures. In contrast, auto-tuning generic BLAS codes for specific targets do not afford expert knowledge about the processor's architecture and has been shown to be competitive with the hand-tuning approach. For our experiments we use the ATLAS implementation of the BLAS library [28], tuned for the target POWER7 server node as well as the IBM Engineering and Scientific Subroutine Library (ESSL)[18].

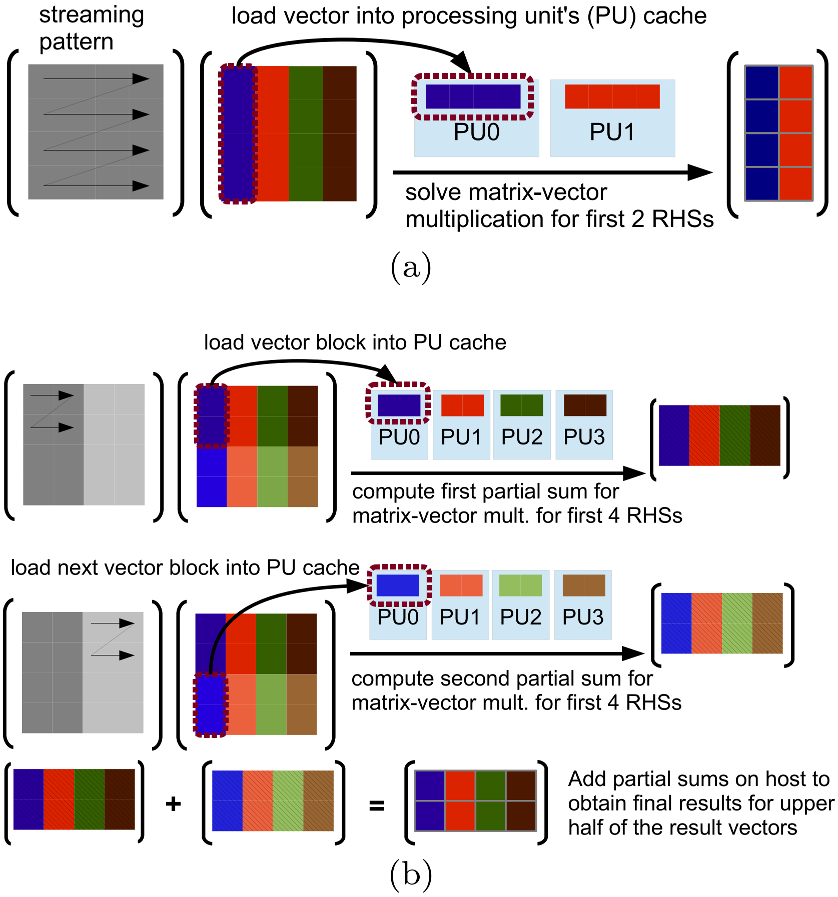
BLAS routines are generally divided into three levels. BLAS Level-1 provides vector operations, e.g., addition, scaling, norm or dot product. Matrix-vector and matrix-matrix operations are provided by Level-2 and Level-3, respectively. From a functional perspective, routines from a lower level serve as building blocks for the next higher level of the BLAS layers. However, BLAS functions are normally implemented individually because simply using the lover-level routines in more complex operations generally has negative effects on performance.

Our FPGA implementation involves routines from all levels of BLAS[1]. The **dot product** (or inner product) of two vectors u and v is the sum-of-products of corresponding vector components $\sum u_i v_i$ and is the basic operation in a general matrix-vector multiplication (**GEMV**). A full general matrix-matrix multiplication (**GEMM**) can be computed by separately invoking GEMV for every vector component of the right-hand side (RHS) matrix.

Operations on vectors and matrices typically have a high degree of inherent data-level parallelism. The key to performance for almost every linear algebra function implementation is to exploit the memory subsystem in order to keep the functional units busy. Our proposed architecture uses the on-chip memory for caching the vector components of the RHS and streams the left-hand side (LHS) matrix from the host. Figure 3a shows an example of a two compute unit (CU) implementation. Each CU is capable of storing an entire vector component and computes a corresponding final result vector. The FPGA's block ram memory is limited (few Megabyte) and would, normally, pose a tight constraint on the possible number of parallel CUs. However, applying a simple blocking scheme allows us increase the number of parallel units by balancing the computational and memory resources used by one CU. The design depicted in Fig. 3b

[1] BLAS Level-2 and Level-3 functions support in-place addition of the result matrix/vector and scaling via scalar parameters. The FPGA architecture as presented in this paper is optimized for the basic matrix multiplication but can be extended to support these features.

Figure 3 Blocking schemes for matrix-matrix multiplication. Scheme 3(a) requires access entire right-hand side columns.


streaming pattern
load vector into processing unit's (PU) cache
PU0
PU1
solve matrix-vector multiplication for first 2 RHSs
(a)
load vector block into PU cache
PU0
PU1
PU2
PU3
compute first partial sum for matrix-vector mult. for first 4 RHSs
load next vector block into PU cache
PU0
PU1
PU2
PU3
compute second partial sum for matrix-vector mult. for first 4 RHSs
+
=
Add partial sums on host to obtain final results for upper half of the result vectors
(b)

can implement twice the number of CUs by reducing the local storage capabilities per unit. Following the steps in Algorithm 1, the matrix-matrix multiplication is done in an iterative process.

Algorithm 1 Blocked matrix-matrix multiplication scheme

Require: Matrix $A_{n,m}$, Matrix $B_{m,p}$, Blocking b, #CUs s
Ensure: Matrix $C_{n,p}$ with $A \cdot B = C$

```
repeat
  Select next $s$ RHS vectors to compute
  for all Compute units $s$ do
    repeat
      Load next sub-vector of size $m/b$ into cache
      Stream sub-matrix of size $b \times (m/b))$ to computes
      Accumulate partial results on host
    until Vector done
  end for
until All vectors computed
```

GEMM software performance The POWER7 core can compute 4 double-precision multiply-add operations in parallel, accounting for 8 FLOPS per cycle per core. The resulting aggregate peak performance of the system is 454.4 GFLOPS.

We measured the actual performance for DGEMM on the host using the self-tuned ATLAS library and the manually tuned ESSL library. For larger data sets (in the order of GBs) the ESSL delivered the best performance peaking at 157 GFLOPS. The ATLAS version was in average 6 % slower than ESSL. However, if the RHS matrix was much smaller, ATLAS performed slightly better as ESSL. We intent to use the GEMM kernel in a liner system solver and are specifically interested in matrix-matrix multiplication problems where the LHS matrix is a dense quadratic matrix and the RHS is a relatively small set of vectors. For this configuration, the ATLAS DGEMM function performs best and is used for our experiments. Doing repeated calls to DGEMV is not an option, because it would eliminate the benefit from cache tiling and data reuse. A test for 24 RHS vectors showed that DGEMM performed an order of magnitude better compared to executing 24 DGEMV calls in sequence.

GEMM Hardware Architecture The proposed hardware architecture is composed of parallel CUs, each of which solving a matrix-vector sub-problem of the overall kernel.

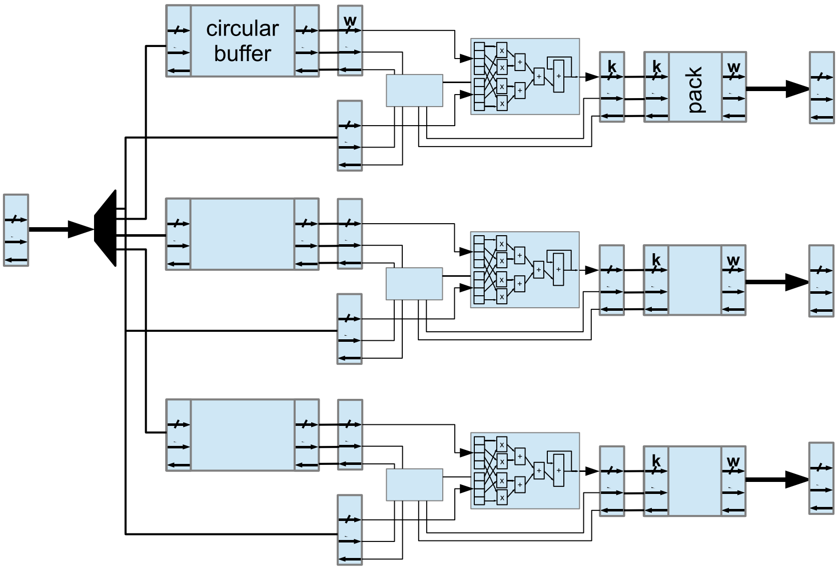
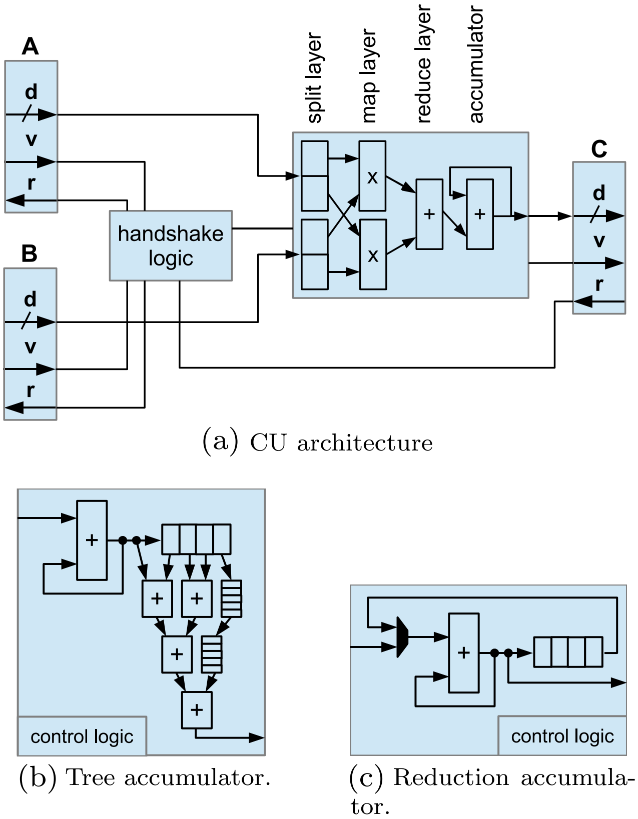


Figure 4 GEMM accelerator architecture. A single input stream is used to load the circular buffer units and to broadcast the matrix stream among the CUs. Each CU produces an individual output stream to the host.

As depicted in Fig. 4, a CU is attached to a circular buffer that serves as a vector cache for a current block of the RHS matrix. Before each run of the GEMM kernel, the buffers get initialized by the host application. When the vector data is present, the LHS matrix is broadcasted to all CUs in a streaming fashion. The CUs calculate the dot-products for each matrix row and forward the results to a pack unit. The pack unit combines $128/s$ consecutive data words into one DMA word and enqueues the data into a DMA put-channel.

The CU core implements a vectorized MAC operation and is sketched in Fig. 5a. Module interfaces are based on a valid/ready handshaking mechanism. The input data width of the CU is fixed to the width of a DMA word of 128-bit. In the first part of the processing pipeline, the DMA words are split into $128/s$ sub-words that match the bit-size s of the native data type. Corresponding vector entries are multiplied in the map layer and combined to a single sum in the reduce layer. The reduce layer is implemented using $128/s - 1$ adders arranged in a binary tree of depth $log_2(128/s)$. We use a balanced tree of adders which is efficient in hardware but, as the sequence of add operations is reordered, can lead to floating-point inconsistency due to rounding errors. For 64-bit data types, the reduce layer consist of a single adder unit.

In the case of floating-point data the most critical part of the processing pipeline is the final accumulator stage. A single-cycle floating-point adder runs at a very low frequency and would compromise performance. In contrast, a pipelined adder can be clocked much faster but, used as an accumulator, has to block the inputs for multiple cycles in order to wait for the current summation result. As we are only interested in the final dot product of the input vectors and do not need to compute a correct stream of prefix sums at the out port of the accumulator we can apply FPGA optimized accumulation schemes. Every result coming from the reduction layer is directly forwarded to one input of a k-stage pipelined adder. The output of the adder is feed-back to the second input port. During the dot-product operation



(a) CU architecture

(b) Tree accumulator.

(c) Reduction accumulator.

Figure 5 Architecture of the GEMM CUs (**a**) and the two alternative accumulator architectures (**b**)–(**c**).

Table 1 FPGA device utilization for various GEMM architectures using double precision float data types.

#CUs	Util.	ALUTs	FFs	Mem.[kb]	DSPs	F_{max}
2 (tree)	17 %	37,432	69,090	1,731	40	259.61
4 (tree)	34 %	72,903	133,623	2,920	80	256.81
6 (tree)	50 %	109,379	197,735	4,110	120	251.64
8 (tree)	67 %	144,122	262,976	5,300	160	233.05
10 (tree)	81 %	179,615	327,659	6,490	200	222.05
12 (tree)	97 %	215,356	369,404	7,680	240	196.19
2 (fsm)	6 %	11,343	25,762	1,731	40	262.67
4 (fsm)	11 %	20,714	46,826	2,920	80	257.86
6 (fsm)	16 %	30,079	67,745	4,110	120	257.2
8 (fsm)	21 %	39,759	89,958	5,299	160	257.0
10 (fsm)	26 %	49,164	111,562	6,489	200	257.0
12 (fsm)	31 %	58,477	132,951	7,678	240	257.4
14 (fsm)	36 %	67,922	154,212	8,868	280	251.89
16 (fsm)	42 %	79,095	178,303	10,057	320	249.75
18 (fsm)	47 %	89,150	200,921	11,246	360	253.04
20 (fsm)	52 %	98,815	222,520	12,436	400	257.2
22 (fsm)	57 %	108,378	244,313	13,625	440	246.37
24 (fsm)	63 %	119,583	270,364	14,814	480	249.63

the register stages of the adder pipeline get populated with partial sums which are the constituents of the actual prefix sum. When a complete vector stream (indicated by an *end*-flag, not depicted in Fig. 4) has been consumed by the accumulator, the partial sums get shifted into a $(k-1)$-stage shift register.

In this paper we study two alternative implementations to reduce the partial sums to the final result. The first solution uses an adder tree to sum up the partial results from the buffer. As depicted in Fig. 5b, the accumulated values are shifted into a tree of adder units. If the adder's delay is not a power of two, we need additional delay buffers to compensate for the incomplete binary tree. The complete accumulator has a delay of $(\lceil log_2 k \rceil) + 1) \cdot k$ clock cycles.

Figure 5c shows the second solution that is based on a resource efficient reduction mechanism but involves a higher latency. In contrast to the tree based design the reduction accumulator uses only one k-stage pipelined adder unit. When the last partial sum is present at the result port of the adder unit, we shift the first entry of the partial sum buffer into the adder. The procedure is repeated $k-1$ times to compute the final sum in $k(k-1)$ clock cycles.

For integer data, a single-cycle adder core is used as accumulator block and thus, no specific accumulation scheme is required. All arithmetic cores for the design were taken from the Altera Megafunctions [1]. The architecture is highly parameterizable and allows for scaling the number of CUs, the buffer size, and the data type of the kernel.

GEMM Resource Utilization: We have synthesized various GEMM designs and varied the accumulator type and the number of CUs. For all design points the cache size per CU was fixed to 64kB. The resource utilization results are summarized in Table 1. The required pipeline depth for a 250MHz double precision floating point adder from the Megafunction core library is 14, the multipliers have an 11-stage pipeline. Compared to the CUs that use the reduction based accumulation scheme (fsm), the tree-based designs (tree) use 13 additional adder cores per CU. As the adder units are mapped to the logic resources of the FPGA, the GEMM CUs that are using the tree design consume much more logic resources than the fsm variants. Due to the demands of the inter-FPGA coupling we set the frequency constraint of the user logic to 250MHz. The designs meet the timing requirements until a certain utilization level. As the tree based accumulators lead to much higher resource occupation, we could only generate valid designs of up to 6 parallel CUs.

3.2 Fast Fourier Transformation

The discrete Fourier transform (DFT) is one of the most widely used algorithms in signal processing and plays an important role in many scientific and technical applications. The fastest known methods for computing a DFT are referred to as fast Fourier transformations (FFTs). The most well-known FFT is the Cooley-Tukey algorithm [8] that uses a recursive divide & conquer approach to reduce the computational complexity of a DFT operation of N samples from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$. An overview of serial and parallel FFT algorithms is given in, e.g. [7].

DFTs are frequently applied in embedded systems as well as in applications from the HPC domain and thus, many highly optimized FFT implementations exist. Special purpose embedded systems use hardware implementations of the FFT to optimize for performance and energy efficiency. As hardware FFTs are often fixed to a specific data type or transformation size, they are normally not used in general purpose systems. Instead, optimized software implementations are applied. While the FFT method greatly reduces the operations required to compute a DFT, the all-to-all communication patters as required for the divide & conquer algorithm generally hampers performance. FPGA accelerators have the potential to bridge this performance gap, because they approach the efficiency of fixed hardware FFTs but are still fully programmable.

FFT Software Performance In order to assess the performance of software based FFTs on the POWER7 server, we compiled the most recent version of FFTW3 [13] (fftw-3.3.4) from source to built a serial version and a threaded SMP version of the library. The multi-threaded library the uses the POSIX pthreads implementation.

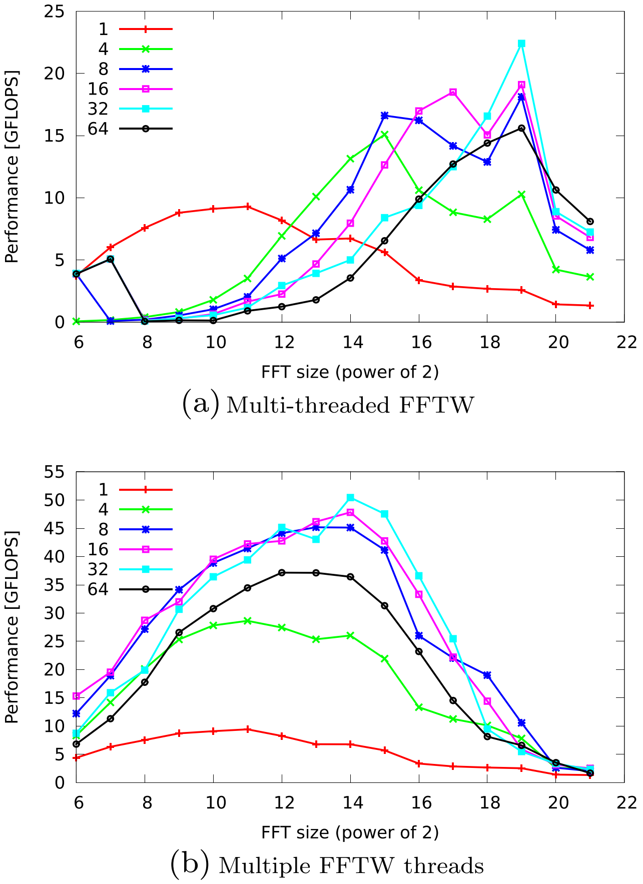
In our benchmarks, we vary the FFT size from 64 to 2 million complex float samples and use up to 64 threads. For each problem size, we prepare batched data of 800 MB and compute the FFT out-of-place over the whole data array to avoid caching effects.

Figure 6 presents benchmark results for two different parallelization methods of the FFT. The GFLOPS performance measure is based on an estimated operation count of $5N \log(N)$ [7].

In Figure 6a, we show the performance of the multi-threaded FFTW3 library for different FFT-dimensions and varying number of threads. In this case, we work on a single FFT problem at a time, but the FFT computation itself is parallelized. The multi-threaded FFTW generally delivers the best performance for larger FFTs and a higher number of threads.

Figure 6b depicts the performance when using the serial version of the FFTW library but starting multiple threads for different input data sets. In contrast to the case in (a), multi-threading on task level can improve the runtime for small FFTs and reaches the highest performance for small to medium size FFTs and a medium number of threads.

FFT Hardware Architecture We use the Spiral framework to generate problem specific FFT cores for FPGA implementation [24]. The Spiral tool requires a problem specification, like the size and the data type of the FFT, as well as some architecture parameters as input. With the parameters, a user describes the algorithm's radix (basic



(a) Multi-threaded FFTW

(b) Multiple FFTW threads

Figure 6 Single precision complex FFTW benchmarks on a dual POWER7 node with 1 to 64 threads using multi-threaded FFTW3 (**a**) and multiple FFTW3 threads (**b**).

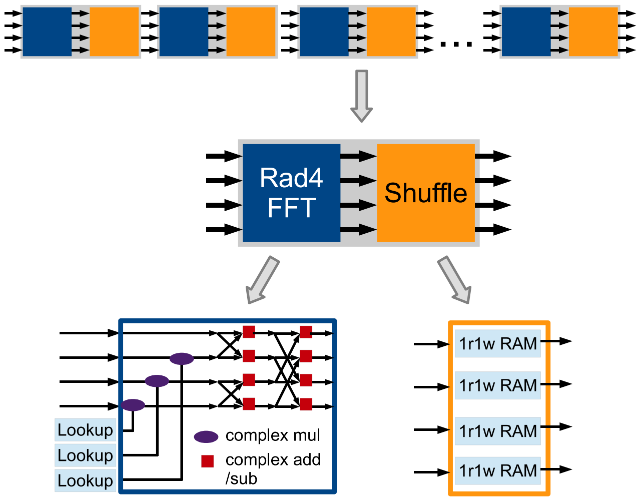


Figure 7 FFT fully streamed processing pipeline with radix 4 based computation unit.

Table 2 FPGA device utilization for the two FFT designs using 16-bit signed integer and single precision floating point data types.

Design	Util.	ALUTs	FFs	Mem.[kb]	DSPs	F_{max}
32k 16-bit	5 %	7,896	20,780	16,920	160	250.69
1k float	12 %	25,068	46,506	3,774	128	251.45

block size), the number of input samples to the core and whether the pipeline is organized as an iterative or fully streamed architecture.

As depicted in Fig. 7 our implementations make use of fully streamed architectures which allows us to input a fixed amount of samples into the design in every clock cycle. For an N-point FFT, the architecture is constructed of $\mathcal{O}(\log N)$ cascaded stages, each of which consisting of a computation unit and a permutation unit. The computation unit implements the butterfly scheme for the selected radix organization. The *twiddle factors* are stored in ROM units implemented in the embedded SRAM blocks on the FPGA. Local ROM look-ups avoid the need of streaming twiddle factors from the input throughout the pipeline.

The permutation of samples at the output of a processing stage is conducted with the help of RAM modules. Depending on the *stride* at a specific FFT stage, different size RAM modules get instantiated and controlled by appropriate address generators. For larger strides, deeper RAMs must be used to ensure that the data can be streamed in-order to the subsequent stage.

The number of input samples to the FFT pipeline is chosen such that the bandwidth of the DMA interface could be fully utilized. At peak, the service FPGA can deliver 16 byte per 250 MHz clock and thus, at most 2 samples per clock cycle can enter the user FPGA when using a single precision complex data type.

The FFT pipeline is embedded into a general streaming design that connects to the DMA interface to the service FPGA. From the host, the user passes pointers to the source and target arrays of the FFT data as well as a scalar indicating the number of transformations in sequence. The parameters are sent to the FFT kernel via MMIO writes. When all parameters are set, the kernel requests the input samples through the DMA interface.

In this paper we study two contrasting FFT pipelines, a radix 2 based floating point design and a radix 4 based 16-bit fixed point design. The floating point kernel computes the FFT over 1024 complex samples and produces a result every 512 clock cycles with a latency of 1617 clock cycles. The fixed-point kernel is designed for 32k-point FFT. Due to the smaller width data format, we can forward 4 complex samples to the pipeline per clock cycle and a result is generated every 8192 cycles. Although the latency of the operators is much lower than for the deeply pipelined floating point cores, the latency of this design is fairly high and amounts to 22554 clock cycles. The latency of the data path is dominated by the permutation units which have a doubled delay in every following FFT stage.

FFT Resource Utilization The FPGA resource utilization for two different FFT kernels is shown in Table 2. The first design computes a 1D FFT over 32k complex 16-bit fixed-point samples, the second implementation applies floating point data on a complex 1k FFT. Both designs consume a fairly small logic footprint on the Stratix-IV device. The memory utilization of about 15.5Mb (which corresponds to 80 % of the embedded block memory resources) is much higher for 32k-sample design. This is expected because the permutation RAMs exponentially grow with the number of FFT stages.

4 Power Measurement Framework

In this section we describe how we measure power consumption on POWER platforms and present the applied energy profiling methodology.

4.1 Amester Tool

Amester is a research tool to remotely collect power, temperature, fan speed, and performance data from IBM servers and to control power management policies in those servers. The sensor data and power management capabilities are provided by EnergyScale, a system-level, out-of-band power management solution for IBM POWER systems [23]. The core of the EnergyScale implementation runs on a dedicated microcontroller and continuously measures voltage and current to calculate the power draw. Additionally, it uses temperature sensors to measure heat as well as performance counters to determine the characteristics of workloads. As depicted in Fig. 8, Amester establishes a network connection to the flexible support processor (FSP) of the server node in order to read out the sensor data from the EnergyScale system.

To allow for high frequency sampling of sensor information, the FSP can be configured to save the data into a local trace buffer which is read out after workload execution or a fixed measurement time. Via the amester tool, we configure the FSP to dump a selected set of sensor registers into the trace buffer. In our case, the subset comprises the available power sensor registers and a time stamp. The sensors a

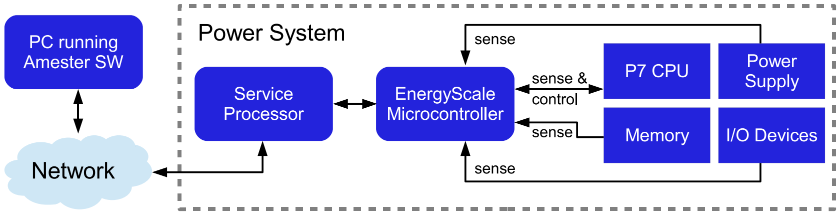


Figure 8 General structure of Amester and the EnergyScale architecture.

can be recorded with multiple sampling rates. We use a 1ms absolute value and a 32ms average in our experiments.

4.2 Profiling Methodology

When running a benchmark, we trace the power for the entire node (P_{sys}), the I/O subsystem (P_{io}), the two processor packages ($P_p = P_{p0} + P_{p1}$) and the DIMMs (P_{mem}). The system power includes overhead for cooling and storage. For any benchmark, we trace the power consumption during idle (P^I) and active operation (P^A). The idle power includes the static power for the FPGA configuration. For that reason we also determine a baseline power consumption P_{io}^B using an *empty* FPGA configuration and compute the static power of the FPGA design as $P^S = P_{io}^I - P_{io}^B$.

With no FPGA coprocessor card attached the I/O domain consumes on average 37.0W. When we plug the card and load and empty configuration onto the user FPGA, the I/0 power consumption increases to about 49.28W resulting in a 12.28W idle power consumption by the FPGA card. This number includes the static power of the PCB and all devices as well as the additional power drawn on the bus to keep up the link.

4.3 Platform Characterization

We trace the benchmark kernels with Amester and align the sensor data and function invocations with the help of real-time tags. The DVFS mode of the POWER7 was activated for all of the presented test cases. Figure 9 shows an example power trace for the two processors, the memory and the I/O subsystem power. Each sample point depicts a 32ms average. In idle operation, the first CPU (CPU0) consumes roughly 36W, the second CPU (CPU1) draws around 31W. After an idle phase of 2 seconds, in which the user FPGA is configured with a baseline idle design, the actual benchmark is executed. First, the user FPGA is reconfigured with a GEMM solver design via the I/O bus. During reconfiguration the I/O subsystem power slightly drops, because the FPGA is switched into the configuration mode. The highest increase in terms of power consumption is measured on the memory sub-domain which represents the power consumed by the DIMMs. In the init phase of the benchmark, the host prepares a test data set of random numbers. In this phase, the FPGA is configured with the user logic but in an idle mode. We can observe that the idle power dissipation significantly depends on the FPGA

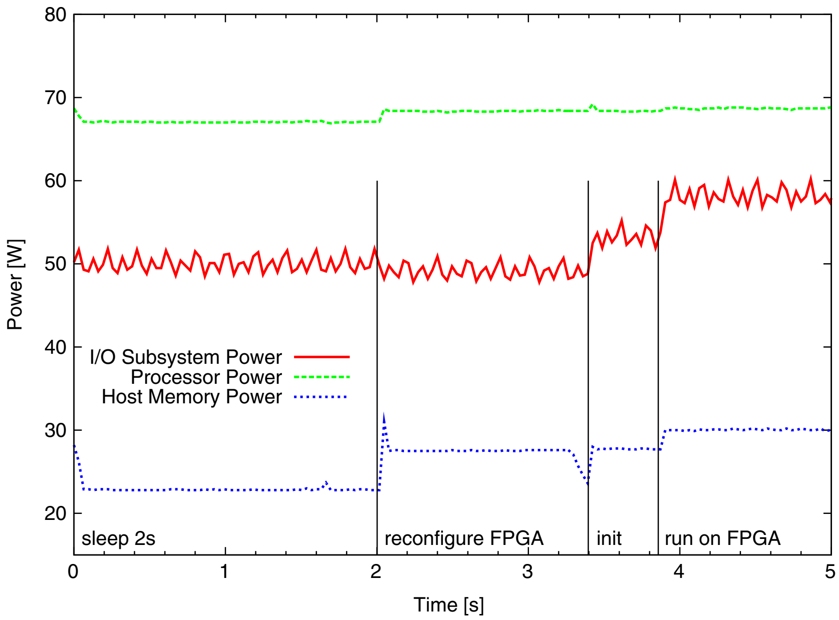


Figure 9 Power trace for CPU, memory, and I/O power consumption during idle, reconfiguration, and computation steps.

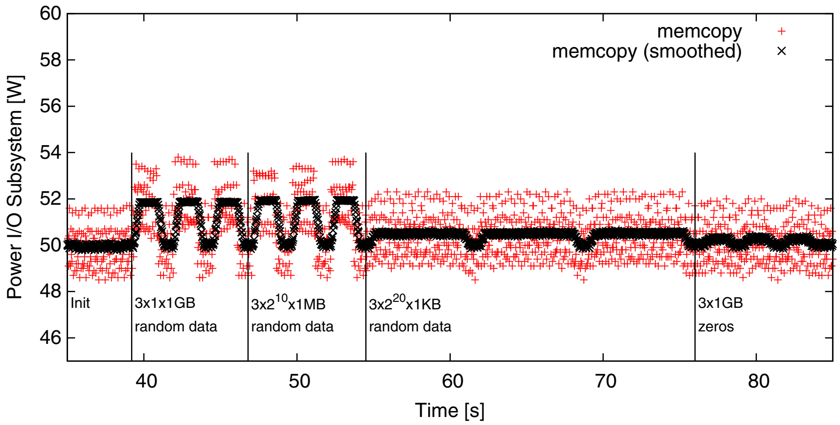


Figure 10 I/O subsystem power consumption for various streaming patterns.

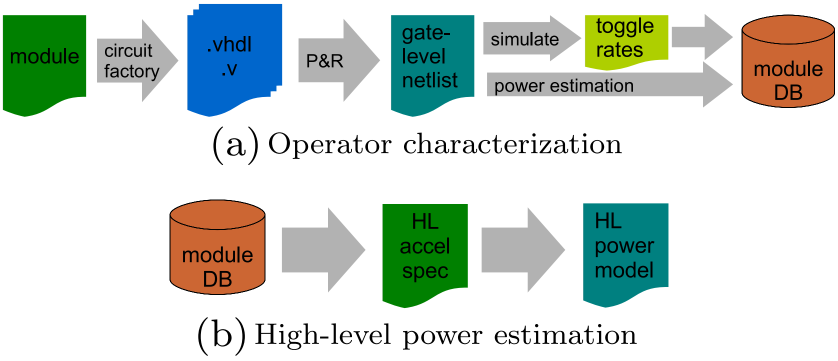
configuration. The GEMM design utilizes a large portion of the FPGA resources and leads to an increased power draw of up to 5.8W from the I/O power domain. When the application invokes the coprocessor call, the FPGA consumes an additional amount of dynamic power due to the switching activity on the device. During runtime the CPU power only slightly increases because the host application consists of a light-weight task that is mainly responsible for virtual addresses translation.

To characterize the overhead of kernel invocations from the host side, we run a memcopy kernel with different input parameters. The memcopy design simply redirects data that is streamed from the host to the FPGA back to the host memory. Figure 10 presents a power trace for a series of benchmark cases, each of which is repeated 2 more times to examine for consistency. In the beginning, we prepare a random data set of 1GB. For the first experiment, we call the kernel for the whole 1GB chunk of data. In the second example, we divide the data into 1024 1MB blocks and run the kernel 1024 times each. For the third benchmark we further reduce the input data size of a kernel invocation to 1 KB. Figure 10 shows a substantial overhead for smaller input data and many kernel executions. The fourth experiment illustrates how the system behaves, when only zeros are copied over the bus. In this case, dynamic energy consumption only occurs in the control path of the system and almost no switching activity on the data path parts of the bus and the FPGA design is caused.

With the platform characterization experiments we substantiate three observations. i.) The static power consumption significantly depends on the FPGA configuration. If the FPGA accelerator is not used by the system, the configuration should be switched to an ilde design consuming only minimal static power. ii) Running a kernel for small input data sets results in a substantial overhead due to the control operations of the host. For the GEMM design, this observation implies that the vector cache for the solver units should not be chosen too small. iii) The input data has strong influence on the switching activity and thus, the dynamic energy consumption of the bus and the FPGA design. One has to ensure to use realistic input data sets when comparing different design alternatives.

5 Power Estimation Framework

The detailed analysis of the kernel FPGA implementations helps us to investigate the feasibility of early power predictions for operator-dense FPGA designs. Our power model is based on pre-computed power estimates for a multitude of arithmetic and memory cores. Figure 11a shows how we populate a module database with utilization results and



(a) Operator characterization

(b) High-level power estimation

Figure 11 High-level power modeling flow.

Table 3 Power estimation for the floating point 1k FFT (upper part) and the 16-bit fixed point 32k FFT (lower part).

Modules	#Modules	P_{static}	$P_{dynamic}$	P_{total}
RAM	18	501	60	561
ROM	14	21	9	30
FPMULT	36	651	198	849
FPADD	58	1187	818	2005
Total	126	2361	1085	3446
RAM	36	555	157	712
ROM	34	28	25	53
MULT	80	371	11	382
ADD	160	1245	53	1298
Total	310	2199	230	2445

All numbers in mW.

power estimates. Among the analyzed cores are integer and floating point modules for various numeric operations as well as RAM, ROM and FIFO memory blocks. We generate a bulk of synthetic designs comprising of a single core module by systematically varying the parameters of the IP core and instrument the Altera Quartus II tools to synthesize these designs. After place&route, we carry out a detailed power estimation using the Altera PowerPlay tool [2] and extracted the static, dynamic, and clock power consumption of the operator module under the assumption of fully randomized input data. We perform the analysis for every specific FPGA device and populate a power database with the collected results.

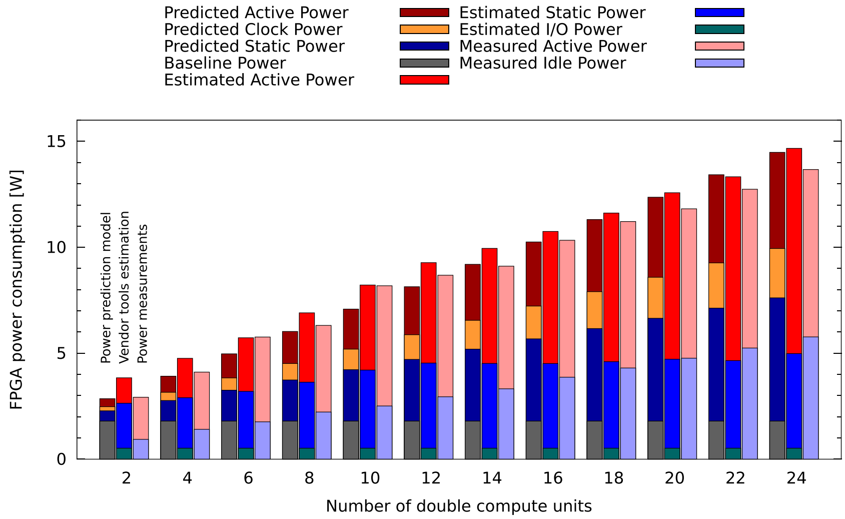
In order to provide a rough power prediction for a specific problem, we analyze the high-level architecture and add up the estimates for the dominant modules in the design. In the DGEMM architecture, the multipliers, adders, and the FIFO memory of the vector cache are the major components that will occupy the vast majority of the utilized FPGA resources. Thus, we use the power estimates for corresponding modules (in terms of frequency, bit width, FIFO depth, pipeline stages, etc.) from our power database (Table 3).

Figure 12 shows predicted, measured, and estimated power consumption for DGEMM CUs using 2 to 24 CUs. For each design point, the left bar shows the static and active power as predicted by our model. The active power further splits into parts that are consumed by the clock network and those which are due to the switching activity on the device. By measurement, we identified a baseline power of approximately 1.8W that is due to the DMA interface logic and the GX++ bus communication. The middle bar depicts the power consumption estimate for the implemented design as generated by the PowerPlay tool. The estimates are in the line with our power measurements as shown by the right bars in the figure.

For the DGEMM example, our power model of precharacterized cores delivers predictions that come remarkably close to the measured numbers. The static power calculated by our model without even knowing the RTL specification of the architecture is almost equal to the numbers reported by the PowerPlay tool after place and route. The most challenging part of the prediction is to determine a realistic switching activity factor α to be used for calculating the active power. We set α to 0.125 which corresponds to the exact toggle rate of a 16-bit counter. This constant has been established as a reasonable approximation for the average toggle rate and is also used as a default in the prevalent FPGA vendor tools for vectorless estimations.

For the FFT case study, we compare the fixed point and the floating point design variants. The pipelined multiplication and addition operators dominate the power consumption in the floating point kernel. Our tool estimates the power draw of the floating point operators to 2.854W which correspond to more than 80 % of the total power consumption. When adding the 1.8W baseline power increase the 1k

Figure 12 Power consumption of DGEMM designs using 2 to 24 compute units (CU). For each design point, the left stacked bar shows the power consumption as estimated by our power prediction model. The middle bar depicts the power as estimated by the Altera PowerPlay tool after place and route and the right bar presents the power as measured with the Amester framework on the POWER7 node.



FFT is estimated to consume 5.4W which is approximately 5 % higher than the actual measured power. The fixed-point design is, in contrast, heavily dominated be ROM and RAM modules. Although the module count is not much higher compared to the floating point case, the depth of the memories is much larger for the 32k FFT and, as depicted in Section 3.2 the FPGA memory resource are consumed up to 80 %. The 16-bit fixed point multipliers can be efficiently implemented in the embedded DSP blocks and also the adders map well to the FPGA fabric, which accounts for a very low power consumption for the arithmetic units. With our estimation tool, we compute a total power draw of 4.445W (after adding the baseline power) which is 27 % lower than the measured power consumption. This indicates, that the memory modules are not as well characterized as the arithmetic operators and that there is some need for further investigation.

6 Energy-Efficiency Analysis

This sections presents power and energy measurement results for the two application kernels. We measure multiple power domains and compare the recorded traces in order to derive a detailed breakdown of the system's power consumption. For both kernels, we compare the performance and energy efficiency tho optimized software implementations running on the same host.

6.1 GEMM

To assess the theoretic peak performance of DGEMM computations we first evaluate the maximum number of floating point units that could be implemented on the used FPGA. A double precision floating point multiplier occupies five 9x9 bit and one 36x36 bit DSP blocks on the FPGA, respectively. At full utilization the device can potentially fit 104 of these multipliers. The adder units are completely implemented with logic resources and the device is capable of easily fitting more than 100 of them. For our targeted clock frequency of 250MHz, the theoretical peak performance of a DGEMM kernel on the Stratix-IV GX530 device is $2.6 \cdot 10^{10}$ MAC Operations per second or 52 GFLOPS (Giga Floating Point Operations Per Second). However, to keep a reasonable amount of memory for the vector cache units we had to restrict the number of compute units to 24, which limits the the peak performance of the design to 24 double precision GFLOPS. The measured performance of the 24 compute unit design is approximately 15.3 GFLOPS or 64 % of the theoretical peak, which is due to the overhead of kernel invocations and the necessary vector load operations during which no computations can occur on the FPGA.

We analyze the energy efficiency of the GEMM architecture in terms of double precision FLOPS per Watt. The FLOPS/W measure is not always a proper energy efficiency metric because the number and kind of operations used to solve a problem can differ greatly for software and hardware programmable systems. However, as our GEMM unit implements a straightforward matrix-matrix multiplication the operation count is directly comparable to a software implementation. The floating point operation count for a GEMM kernel $f : \mathbb{R}^{m \times k} \times \mathbb{R}^{k \times n} \rightarrow \mathbb{R}^{m \times n}$ amounts to $2nmk$. We measure the runtime T of the kernel operations and the average power P^{avg} consumed during the solving process. The energy efficiency of kernel operation is computed as $2nmk/(T \cdot P^{avg})$.

It is an important decision which part of the power consumption is factored in the calculation of P^{avg}. One alternative is use P^{avg}_{sys} which corresponds to the full system average power during the active processing phase. A second alternative is to determine the dynamic power consumption $P^{avg}_{dyn} = P^S + \sum_{i \in \{io,p,m\}} \left(P^A_i - P^I_i\right)$, depicting the part of the processing power that is spend atop of the idle power in a specific domain (plus the static power of the FPGA configuration).

Figure 13 shows the dynamic power consumption measurements for the floating point and fixed point GEMM kernels using 2 to 24 compute units. The orange and green part reflect the dynamic power overhead measured in the CPU and memory power domains, respectively. As the architecture is designed to fully utilize the streaming bandwidth, the activity for accessing the memory and the address translation tasks on the CPU are similar for all variants. Due to the simpler and smaller footprint arithmetic units, the static and active power for the integer GEMM cores is significantly smaller compared to the floating point design.

In the power consumption breakdown for a 24 compute double precision GEMM design the static power of 12.28W

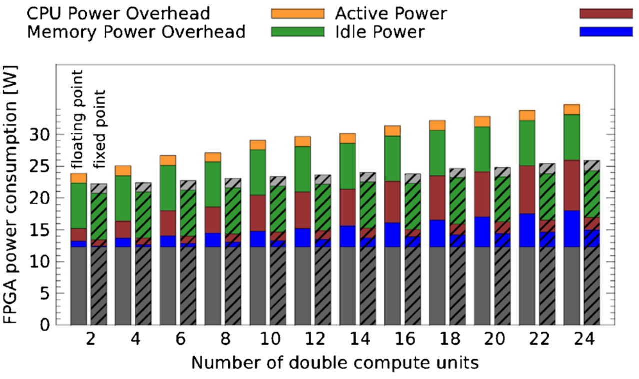
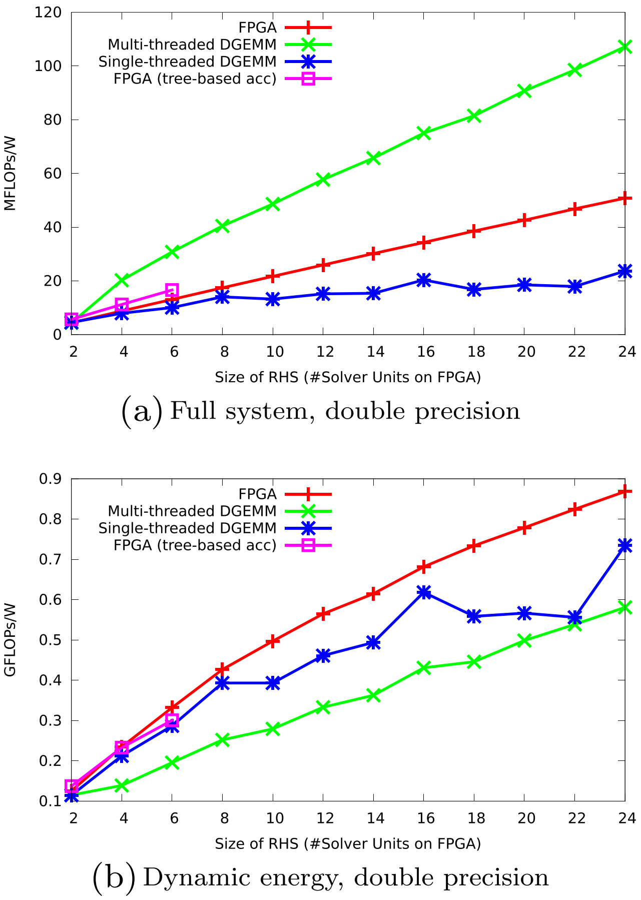


Figure 13 Measured power consumption for floating point and fixed point GEMM kernels using 2 to 24 compute units.

for the coprocessor card is the biggest constituent of the power draw. The static FPGA configuration power consumption of 5.77W is a critical part, because it is consumed continuously, also if the device is idle. As a consequence, the user should reconfigure the device with the baseline configuration after the application has finished to ensure that the static power is low if the device is unused. A substantial part of 7.20W atop of the average idle power is used in the memory modules on the host. The power overhead spend for the processing on the host CPU amounts to 1.62W. When the kernel is executed on the FPGA, the power consumption of the system is increased by 7.9W.

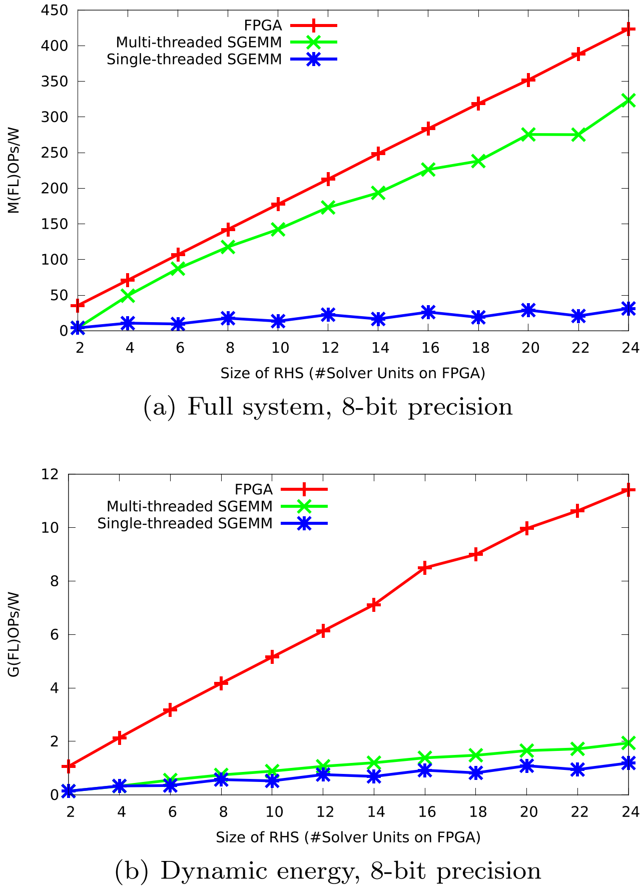
In Figs. 14 and 15 we compare P_{sys}^{avg} and P_{dyn}^{avg} for ATLAS based CPU versions and our FPGA implementation of the GEMM kernel using a RHS matrix of size 8000 × 8000 and 32000 × 32000. For 2 to 24 RHS vectors the ATLAS implementation achieves 1.2 to 35.3 GFLOPS when multi threading is used and 1.3 to 6.5 GFLOPS when the single threaded version of the library is applied. The performance of our DGEMM architecture ranges from 1.3 to 15.3 GFLOPS depending on the number of compute units.



(a) Full system, double precision

(b) Dynamic energy, double precision

Figure 14 Energy-efficiency of DGEMM kernels.



(a) Full system, 8-bit precision

(b) Dynamic energy, 8-bit precision

Figure 15 Energy-efficiency of GEMM kernels for 8-bit data types.

For a RHS size of 24 vectors, the CPU is 2.3 times faster than our 24-CU design and consumes 53 % less energy on the system domain (cf. Fig. 14a). However, as we do a full unload of the GEMM kernel, the host systems remains idle and could potentially be used for other workloads. The P_{dyn}^{avg} based metric excludes the portion of the power consumption that is spend during idle operation. Figure 14b reveals that the FPGA variant under this assumption is more energy-efficient than both CPU alternatives.

The DGEMM designs that use the tree-based accumulator architecture can be compared up to 6 compute units, because we could only synthesize valid designs for up to this size. If P_{sys}^{avg} is used as the reference power consumption, the tree-based CUs are more efficient than the fsm-based versions. The reason for this is that the lower latency tree accumulators lead to a slightly shorter runtime and that the higher processing power of the FPGA device is negligible when the power consumption of the whole server node is factored in. If the metric uses P_{dyn}^{avg}, the fsm-based DGEMM CUs can compensate the lower performance by their significantly lower dynamic power.

When moving from double precision to a char type, the amount of data that has to get loaded from memory and piped through the FPGA shrinks by a factor of 8. Additionally, the static and dynamic power consumption of the FPGA designs substantially drops due to the simple 8-bit operators. Figures 15a and b shows how a shift to a lower precision data type translates in terms of operations per Watt. We compare the results for the 8-bit fixed-point CU to SGEMM function calls from the ATLAS library. This comparison is, from a functional perspective quite unfair, because it compares arithmetic on a small size fixed-point data type with floating point computations. However, the SGEMM function delivers the highest performance (and lowest energy consumption) for a matrix-matrix multiplication on the CPU. To the best of our knowledge, there is no suitable fixed-point library optimized for the POWER7 vector units and the development of such a library is out of scope for this paper. Concerning the energy spend on the entire system, our 8-bit GEMM kernel is roughly 29 % better than SGEMM. If only P_{dyn}^{avg} is considered, the FPGA solution is about 5 times better than the best pure CPU version.

6.2 FFT

In this section we present the power measurements and performance results for the FFT case study. We compare the performance of FPGA software implementations using the FFTW library. As for the GEMM benchmarks, we decided to study one design point on the FPGA that uses a fixed point data format. For many signal processing applications the accuracy of fixed point arithmetic is sufficient and thus, we illustrate to what extend a transition from a single precision floating point software-based FFT to a fixed-point FPGA-based FFT can improve the energy efficiency. Moreover we directly compare 1k-point FFT implementations on FPGA and CPU using the same data format and providing identical precision. A power breakdown for the FFT accelerators is depicted in Fig. 16. The static power consumption of the coprocessor card and the dynamic power of the host memories are the largest contributors to the power draw.

The first column of Table 4 shows the idle power of the system as traced with the amester framework when the node was running the OS with no user tasks. The power governor on the CPU was set to DVFS with the priority to power savings (indicated by $\ominus$ at the superscript) and thus, to a mode in which the node consumes the lowest amount of power. For the FFTW benchmarks we switched the power governor to a DVFS mode that favors performance (superscripted marked with $\oplus$). This leads to a higher average power draw but also significantly increases performance so that the performance-per-Watt is normally higher than in a lowest power mode. The numbers for S_s and H_s depict the results for a s-input complex FFT executed in software or hardware, respectively. For the 32k FFT, we pick the best-performing multi-threaded variants as well a sequential implementation. In the S^{thr64} case, we use 64 threads, with each thread working on a separate FFT, the numbers for S^{par16} show the performance for parallelized FFTW calls using 16 threads.

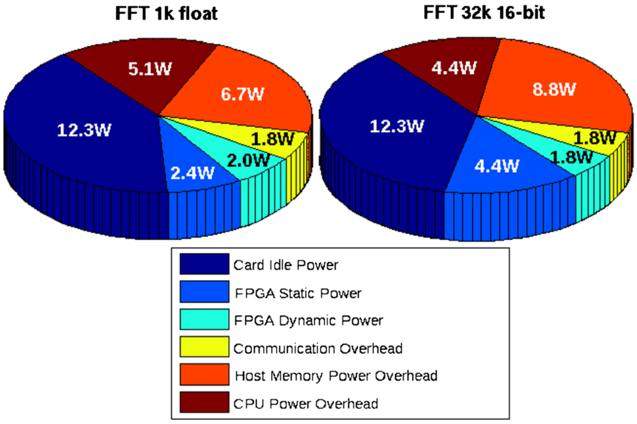


Figure 16 Power breakdown for the FFT accelerators.

The fastest software version for the 32k-FFT delivers 36.5 GFLOPS at 453W of system power. The serial version consumes 400W but only achieves about 5.5 GFLOPS. Using a fixed-point data format on the FPGA, we can increase the performance to about 54 GFLOPs while consuming only 15W more power than in idle mode. Thus, the energy efficiency in terms of performance per Watt can be increased by 2.3× if the total power of the node is used as reference power, or 17.8× if only the dynamic power consumption is factored in.

For the 1k-FFT case study, we also vary the mode of the system's power governor. As we can not increase the software performance for 1k-FFTs by using calls to the multi-threaded FFTW library (cf. 3.2), we analyze sequential and thread-parallel versions using 4 and 16 threads on the CPU. The power favoring DVFS mode (S^{-}) keeps the CPU in the lowest possible voltage/frequency mode and only increases clock speed under heavy load. This leads to a power drop of more than 100W but also significantly reduces the systems average performance. Hence, the change of the frequency scaling algorithm is normally not the first choice if the efficiency metric involves the total system power. If we apply the dynamic power as a baseline for efficiency, the lower power mode on the CPU delivers more than 8× better results.

The floating point kernel on the FPGA shows a performance of 17.48 GFLOPS for the 1k-FFT. In therms of dynamic energy efficiency the FPGA solution outperforms the most efficient software variants by 1.2× and

Table 4 Power consumption for FFTs.

Domain	$S^{\ominus Idle}$	$S^{\oplus seq}_{32k}$	$S^{\oplus thr64}_{32k}$	$S^{\oplus par16}_{32k}$	$H^{\ominus fxt}_{32k}$	$S^{\ominus seq}_{1k}$	$S^{\ominus thr4}_{1k}$	$S^{\ominus thr16}_{1k}$	$S^{\oplus seq}_{1k}$	$S^{\oplus thr4}_{1k}$	$S^{\oplus thr16}_{1k}$	$H^{\ominus flt}_{1k}$
P_{sys}	275.11	399.82	453.60	431.52	291.34	286.55	301.66	314.22	403.42	428.78	450.14	295.07
P_{io}	49.43	49.49	49.54	49.57	55.66	49.55	49.58	49.57	49.55	49.65	49.50	53.86
P_{p0}	35.44	96.27	132.34	116.08	37.25	37.64	41.88	49.19	97.16	111.32	127.65	37.24
P_{p1}	30.82	88.35	95.54	94.57	31.34	31.10	31.90	32.33	88.67	89.30	89.16	31.41
P_{mem}	23.01	29.05	41.17	36.55	29.40	30.61	38.80	45.90	31.89	41.96	46.72	30.49
$Perf$		5.56	36.51	12.78	53.87	6.27	20.14	38.73	9.46	29.18	41.37	17.48
MF/W		13.90	80.48	29.61	184.91	21.89	66.76	123.26	23.45	68.06	91.91	59.25
P_{dyn}		124.45	179.90	158.07	14.95	10.21	23.46	38.30	128.57	153.53	174.34	14.30
GF/W		0.045	0.203	0.081	3.604	0.614	0.858	1.011	0.074	0.190	0.237	1.222

P_{io}, P_{mem}, and P_{pi} depict the average power draw consumed in the I/O subsystem, the memory subsystem and in the two processors, respectively. P_{sys} is the total average power consumption of the server node and P_{dyn} the amount of power spend atop of the idle power off the node. The performance $Perf$ is given in GFLOPS, the energy efficiency in MFLOPS/W (MF/W) and GFLOPS/W (GF/W). The superscript text at the column headers depicts the set-up for the benchmark. ⊕ or ⊖ indicates whether the power governor was configured to favor power or performance, respectively. seq, thr, and par indicate the if and how the FFTW calls were parallelized. fxt represents a fixed point format for the FPGA FFT, and flt a floating point format.

5.1×, depending on the applied mode of the DVFS power governor.

7 Related Work

Kestur et al. [20] propose a BLAS Level-2 implementation on FPGA and compare the execution time, average power, and energy consumption with vendor tuned math libraries on CPU and GPUs. Their FPGA architecture stores the matrix and the RHS vector in on-chip memories limiting the solvable problem size to only 512kB for the matrix. The reported energy efficiency improvements of up to 293× compared to an Nvidia Tesla C1060 are impressive. However, the limitation of the matrix size leads to ignoring the effects of data I/O and to compromising the GPU's performance. Fowers et al. [12] evaluate the energy consumption of sliding window applications on CPU, GPU, and FPGA. Their findings reveal the suitability of FPGA accelerators for image computations and show that FPGAs can outperform multi-core CPUs and and GPUs in terms of energy efficiency for this specific class of applications. In contrast to our power-profiling method, the energy analysis is based on the Thermal Design Power (TDP) specifications of the studied devices and thus, a worst case approximation of the actual energy consumption.

Numerous floating point dense matrix-matrix multiplication architectures for FPGAs have been proposed in literature, e.g., [10, 21, 31, 32], and [29]. The floating point accumulator architecture presented in this work is akin to the reduction circuit presented in [30]. Another FPGA-optimized approach for floating point accumulation is given in [9]. The presented module is very resource efficient, but has to be scaled to the numerical requirements of the application.

A multi-core architecture optimized for GEMM and FFT workloads is presented in [26]. Using their programmable ASIC approach, the authors demonstrate an efficiency of up to 30 GFLOPS for a 2D FFT. Several papers discuss FFT implementations in FPGA technology. Hemmert and Underwood evaluate and compare multiple FFT implementations on FPGA [16]. The method described in [19] allows for trading between FPGA resource utilization and absolute latency for radix 2 based FFTs. Chen et al. propose an FFT architecture that explores pipeline parallelism and task parallelism in order to optimize for energy efficiency [5].

To analyze the power consumption of FPGA-based systems, researchers apply external power meters (e.g., [20]), power estimation tools (e.g., [5, 29]), or published data (e.g., [12, 26]). Our power-profiling methodology differs from the presented approaches and allows for analyzing the hardware in a much finer grained manner. In contrast to the majority of related work, our energy efficiency analysis includes system level components, such as the power overhead spend on the CPU for accelerator-supporting tasks, the host memory and the I/O links. This paper reviews and extends our previous work [14], including a detailed analysis of pipelined FFT architectures and a more detailed presentation of our power estimation framework.

8 Conclusion

The paper presents a performance, power, and energy analysis of GEMM and FFT kernels on FPGA. Our results show that offloading arithmetic kernel functions to the FPGA can increase the energy efficiency of the system. Using the non-intrusive, high-resolution sampling of different power domains on the server we can break down the power draw to the systems main components. A substantial part of the energy is consumed by the memories on the host, mainly because of the streaming nature of the kernel design. We leave it to future work to analyze if the memory costs can be optimized by the use of local DRAM storage on the card. The evident strength of the FPGA becomes apparent when non-standard data types are used within the kernel.

If it is sufficient to execute the matrix multiplications or FFTs in in an integer format, the use of the custom FPGA design improves the performance and energy efficiency of the system. We utilize low precision GEMM codes within an iterative refinement framework that applies a simple integer format in the inner loop whilst guaranteeing a high precision of the computed result using a floating point outer loop. Fourier transformations are frequently applied to fixed point data but as CPUs and libraries are tuned for floating point arithmetic, the raw data is normally converted before processing. We demonstrate an 18× gain in energy efficiency on a real system when offloading the fixed point FFT to an FPGA coprocessor. The launch of floating point hard macros in FPGAs will help to achieve similar results for floating point data. Even with current devices which implement floating point units with the general logic resources we are able to show efficiency gains of up to 5× for a 1k complex FFT.

The insights gained in this work contribute to different further studies. The power-profiling of standardized library functions on different devices can be integrated into an energy-aware operating system that dynamically directs function calls to processing devices in order to deliver the best energy efficiency with respect to the parameter set of the call and the current system state. Accurate estimation tools are essential for automating the hardware/software co-design process. Our power model shows a good prediction quality for the floating point DGEMM and FFT kernels and can be used by profiling tools to analyze the effects of offloading kernel functions to an FPGA based accelerator.

References

1. Altera Corp. (2013). *Floating-Point Megafunctions*: User Guide.
2. Altera Corp. (2013). Quartus II Handbook Version 13.1. ch. PowerPlay Power Analysis.
3. Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Croz, J.D.u., Hammerling, S., Demmel, J., Bischof, C., & Sorensen, D. (1990). LAPACK: A Portable Linear Algebra Library for High-performance Computers. In *ACM/IEEE Conf. on Supercomputing (SC'90)*.
4. Brigham, E.O. (1988). *The Fast Fourier Transform and Its Applications*: Prentice-Hall.
5. Chen, R., Park, N., & Prasanna, V.K. (2013). High throughput energy efficient parallel FFT architecture on FPGAs. In *High Performance Extreme Computing Conference (HPEC)* (pp. 1–6): IEEE.
6. Choi, J., Dongarra, J., Pozo, R., & Walker, D. (1992). ScaLAPACK: A Scalable Linear Algebra for Distributed Memory Concurrent Computers, LAPACK Working Note 55.
7. Chu, E., & George, A. (2000). *Inside the FFT Black Box. Serial and Parallel Fast Fourier Transform Algorithms*: CRC Press.
8. Cooley, J.W., & Tukey, J.W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, *19*, 297–301.
9. de Dinechin, F., Pasca, B., Cret, O., & Tudoran, R. (2008). An FPGA-specific approach to floating-point accumulation and sum-of-products. In *Int. Conf on Field-Programmable Technology (FPT'08)*: IEEE.
10. Dou, Y., Vassiliadis, S., Kuzmanov, G.K., & Gaydadjiev, G.N. (2005). 64-bit Floating-point FPGA Matrix Multiplication. In *Int. Symp. on Field-programmable Gate Arrays (FPGA'05)*: ACM.
11. Esmaeilzadeh, H., Blem, E., Amant, R.St., Sankaralingam, K., & Burger, D. (2011). Dark Silicon and the End of Multicore Scaling. In *Int. Symp. on Computer Architecture (ISCA)*.
12. Fowers, J., Brown, G., Cooke, P., & Stitt, G. (2012). A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications. In *Int. Symp. on Field-programmable Gate Arrays (FPGA'12)*: ACM.
13. Frigo, M., & Johnso, S.G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, *93*(2).
14. Giefers, H., Polig, R., & Hagleitner, C. (2014). Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid CPU/FPGA system. In *Application-specific Systems, Architectures and Processors (ASAP)* (pp. 92–99): IEEE.
15. Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., & Horowitz, M. (2010). Understanding sources of inefficiency in general-purpose chips. In *Int. Symp. on Computer Architecture (ISCA)*.
16. Hemmert, K.S., & Underwood, K.D. (2005). An analysis of the double-precision floating-point FFT on FPGAs. In *Field-Programmable Custom Computing Machines (FCCM)* (pp. 171–180): IEEE.
17. http://www.netlib.org/blas.
18. IBM Corp. (2012). ESSL Guide and Reference.
19. Inggs, G., Thomas, D., & Winberg, S. (2012). Exploring the latency-resource trade-off for the Discrete Fourier Transform on the FPGA. In *Field Programmable Logic and Applications (FPL)* (pp. 695–698): IEEE.
20. Kestur, S., Davis, J., & Williams, O. (2010). BLAS Comparison on FPGA, CPU and GPU. In *Annual Symposium on VLSI (ISVLSI)*: IEEE.
21. Kumar, V., Joshi, S., Patkar, S., & Narayanan, H. (2009). FPGA based high performance double-precision matrix multiplication. In *Int. Conf. on VLSI Design*: IEEE.
22. Lawson, C.L., Hanson, R.J., Kincaid, D.R., & Krogh, F.T. (1979). Basic linear algebra subprograms for fortran usage, *5*(3).
23. McCreary, H.-Y., Broyles, M.A., Floyd, M.S., Geissler, A.J., Hartman, S.P., Rawson, F.L., Rosedahl, T.J., Rubio, J.C., & Ware, M.S. (2007). Energyscale for IBM POWER6 microprocessor-based systems. *IBM Journal of Research and Development*, *51*(6), 775–786.
24. Milder, P., Franchetti, F., Hoe, J.C., & Püschel, M. (2012). Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems*, *17*(2), 15:1–15:33.
25. Moore, G.E. (1965). Cramming more components onto integrated circuits. *Electronics*, *38*(8).

26. Pedram, A., McCalpin, J., & Gerstlauer, A. (2014). A highly efficient multicore floating-point FFT architecture based on hybrid linear algebra/FFT cores. *Journal of Signal Processing System*, *77*(1-2), 169–190.
27. Putnam, A., Caulfield, A., Chung, E., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., & Burger, D. (2014). A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Int. Symp. on Computer Architecture (ISCA)*.
28. Whaley, R.C., & Petitet, A. (2005). Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, *35*(2), 101–121.
29. Zhang, W., Betz, V., & Rose, J. (2012). Portable and scalable FPGA-based acceleration of a direct linear system solver. *ACM Transactions on Reconfigurable Technology Systems*, *5*(1), 6:1–6:26.
30. Zhuo, L., Morris, G., & Prasanna, V. (2007). High-performance reduction circuits using deeply pipelined operators on FPGAs. *IEEE Transactions on Parallel Distributed Systems*, *18*(10), 1377–1392.
31. Zhuo, L., & Prasanna, V.K. (2005). High Performance Linear Algebra Operations on Reconfigurable Systems. In *ACM/IEEE Conf. on Supercomputing (SC'05)*: IEEE.
32. Zhuo, L., & Prasanna, V.K. (2007). Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Transactions Parallel Distributed Systems*, *18*(4), 433–448.

Heiner Giefers received his diploma and PhD degrees from the University of Paderborn in 2006 and 2012, respectively. After receiving his PhD he held a position as technical consultant for FPGA-accelerated high-performance computing at the Paderborn Center for Parallel Computing (PC^2). In 2013 he joined the IBM Zurich Research Laboratory where he is working on energy-efficient architectures, reconfigurable computing and hardware-software codesign.

Raphael Polig received his Dipl.-Ing. degree in electrical engineering and information technology from the Technische Universität München, Munich, Germany, in 2008. The same year he joined the IBM Research & Development Center in Böblingen, Germany, as a circuit design engineer working on high performance SRAM based caches for IBMs POWER and z processors. He led efforts for transistor level design automation in cooperation with the University of Bonn, Germany. In 2012 Raphael joined the Accelerator Technologies group at IBM Research - Zurich, Switzerland, as a Ph.D. student focusing on reconfigurable architectures for text-based analytics.

Christoph Hagleitner received his Masters and Ph.D. degree in Electrical Engineering from the Swiss Federal Institute of Technology (ETH), Zurich in 1997 and 2002, respectively. In 2003 he joined the IBM Zurich Research Lab where he leads the Accelerator Technologies group. The main focus of his research are high-performance and energy-efficient hardware accelerators for business and high-performance computing applications. His research agenda spans topics from novel computing paradigms (e.g., NEM switches for logic applications) to application-level acceleration (e.g., text analytics acceleration for the processing of unstructured documents). He has authored and co-authored 80+ papers in refereed journals and conference proceedings and written/edited several book chapters.