Exploiting mixed-mode parallelism for matrix operations on the HERA architecture through reconfiguration

Xiaofang Wang and Sotirios G. Ziavras

Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102, USA
{xw23, ziavras}@njit.edu

Abstract

Recent advances in multi-million-gate platform FPGAs have made it possible to design and implement complex parallel systems on a programmable chip (PSOPCs) that also incorporate hardware floating-point units (FPUs). These options take advantage of resource reconfiguration. In contrast to the majority of the FPGA community that still employs reconfigurable logic to develop algorithm-specific circuitry, our FPGA-based mixed-mode reconfigurable computing machine can implement simultaneously a variety of parallel execution modes and is also user programmable. Our HERA (HEterogeneous Reconfigurable Architecture) machine can implement the SIMD (Single-Instruction, Multiple-Data), MIMD (Multiple-Instruction, Multiple-Data) and M-SIMD (Multiple-SIMD) execution modes. Each processing element (PE) is centered on a single-precision IEEE 754 FPU with tightly-coupled local memory, and supports dynamic switching between SIMD and MIMD at runtime. Mixed-mode parallelism has the potential to best match the characteristics of all subtasks in applications, thus resulting in sustained high performance. We evaluate HERA's performance by two common computation-intensive testbenches: matrix-matrix multiplication (MMM) and LU factorization of sparse Doubly-Bordered-Block-Diagonal (DBBD) matrices. Experimental results with electrical power network matrices show that the mixed-mode scheduling for LU factorization can result in speedups of about 19% and 15.5% compared to the SIMD and MIMD implementations, respectively.

*This work was supported in part by the U.S. Department of Energy under grant DE-FG02-03CH11171.

1 Introduction

With the recent achievement of multi-million-gate platform FPGAs to contain richer embedded feature sets, such as plenty of on-chip memory, DSP blocks and embedded microprocessor IP cores, FPGA-based reconfigurable computing is going through a revolution. While the majority of the FPGA community still employs FPGAs to design and implement algorithm-specific circuitry, FPGAs have made their way into the high-performance computing world. It has become feasible to build high performance, low power PSOPCs at affordable costs. The peak floating-point performance of FPGAs has outnumbered in the last two years that of modern microprocessors and is growing much faster than the latter [1]. Recent research efforts in the design and implementation of FPUs [2, 3] on state-of-the-art FPGAs provide evidence to this effect. However, we have not seen major research efforts in this new direction and few FPGA-based computing systems incorporating FPUs have been published. This new approach requires extensive expertise in computer architecture, parallel processing, and digital and FPGA-based designs to yield high performance. Numerous research efforts have been exploring high-level approaches to compile applications into reconfigurable architectures [4].

At the same time, the promise of billion-transistor chips in a few years' time presents new opportunities in computing that bring up a main issue: what kind of (micro)architectures has the better promise for scalable performance under technology scaling? Conventional microarchitectures are approaching a performance limit due to the limited ILP (Instruction Level Parallelism) in real programs [5]; their large power dissipation is a major problem as well. Also, wire delays decrease much slower than transistor switching times for deep sub-micron processes. As a result, a major shift from ILP to TLP (Thread Level Parallelism) is present in both the industry and research communities. On the other hand, ASIC-based single-chip multiprocessors have gained popularity [6-7]. However, a high volume is required to amortize the high development and NRE (nonrecurring engineering) costs of ASIC-based approaches, especially for deep sub-micron designs. Also, the ever-shortening product cycles and the high design complexity of such solutions limit their viability [7]. In contrast, FPGA technologies provide a great opportunity to system designers to combine the high-performance of ASIC designs with the programming flexibility of microprocessors [8]. In fact, recently introduced reconfigurable arrays for ASIC Systems-on-a-Chip (SoCs) reduce their design complexity, and add programmability and flexibility [9]. More importantly, by taking advantage of the dynamic reconfiguration of FPGAs we could customize the hardware at runtime to match the characteristics of the applications [10].

From the application's point of view, the performance of given computing systems is not optimal for most subtasks due to the system's expected unsuitability; different subtasks in an application normally require different architectures for high performance. SIMD and MIMD are the two fundamental and complementary parallel modes of execution. SIMD's superior ability for data parallelism, often enhanced with low inter-PE communication and synchronization overheads, make it superior to MIMD in performing fine-grain tasks [11-12]. Many numerical analysis algorithms, such as large-scale matrix multiplication and LU factorization, have a very high degree of structured, fine-grain parallelism and can benefit substantially from the SIMD mode. However, due to SIMD's implicit synchronization, SIMD machines are often under-utilized for applications involving dynamic parameters and an abundance of conditional statements. On the other hand, MIMD machines consisting of independent PEs are good at conditional branching. However, the PE independence property of MIMD makes programming cumbersome. For applications prone to SIMD execution, the need to explicitly synchronize the PEs in MIMD realizations produces substantial overheads. Furthermore, every PE in MIMD requires its own program memory. Mixed-mode heterogeneous computing [13], where the machine's operational mode (i.e., SIMD, MIMD or

M-SIMD) changes dynamically as needed by individual subtasks in an application, integrates effectively most of the SIMD and MIMD advantages while alleviating their major drawbacks.

This paper presents our mixed-mode HERA computer, which is coarse-grain, dynamically reconfigurable and supports floating-point arithmetic. It is based on Xilinx Virtex II and Virtex II pro platform FPGAs. Every PE in HERA is built around an IEEE 754 single-precision FPU. The system can be reconfigured dynamically at runtime to support a variety of independent or cooperating computing modes, such as SIMD, MIMD and M-SIMD, to best match in the time spectrum all subtask characteristics in a single application. Also, every PE is tightly coupled with its own local data and program memories. The low communication overheads and very low cost of our approach add much needed promise to the high-performance, low-cost computing field.

HERA's target domains are large-scale, computation-intensive, matrix-based scientific and engineering applications. Many common problems appearing in areas such as power engineering, bioinformatics, structural analysis, circuit simulation, traffic simulation, fluid dynamics and chemical engineering can be formulated as the recurring solution of systems of equations. The corresponding matrix-based algorithmic solutions are often computation intensive and present major challenges to current computing systems. General-purpose processors do not perform well in these problems since they are not optimized to meet their needs. Substantial gains achieved by specialized processors for these problems justify our research efforts. MMM and the LU factorization [14] of large matrices are two examples of computationally expensive primitives in these areas, requiring time O(N^3), where N x N is the matrix size; they play such an important role in many applications that their efficient implementation has long been the focus of many research efforts. In this paper, we show that by employing an innovative data partitioning scheme and mixed-mode scheduling on our 64-

PE HERA machine, Cannon's MMM algorithm [15] can be efficiently adapted for matrices of arbitrary shape and size. In [16-17], we carried out the parallel LU factorization of sparse DBBD matrices on an in-house developed MIMD configurable multiprocessor employing the Altera Nios® IP core; several tree networks interconnected the processors. In contrast, we demonstrate here the beneficial effect of mixed-mode scheduling in HERA for improved performance. Our preliminary HERA design was reported in [18]. We show here our improved HERA architecture and study its performance for parallel MMM and the LU factorization of large sparse matrices appearing in real-world power flow analysis. Many other application areas, such as signal and image processing, could also benefit from HERA provided that: (a) it employs a generalized inter-processor connection structure which is realized by a mesh network augmented with global buses and limited shared-memory capabilities; and (b) it is user programmable via a matrix-oriented optimized instruction set. The paper is organized as follows. Section 2 briefly discusses related work. The architecture of HERA is presented in Section 3. Section 4 contains implementation details. A brief discussion on programming systems like HERA is presented in Section 5. In Section 6, we illustrate mixed-mode scheduling for MMM and LU factorization on HERA. Performance results for MMM on a 64-PE HERA are provided and compared to those of two Dell PCs. Parallel LU factorization of sparse DBBD matrices and relevant SIMD, MIMD and mixed mode scheduling on a 36-PE HERA are also included. These experimental results and theoretical analysis are employed to prove the strengths of HERA with mixed-mode scheduling. Conclusions follow in Section 7.

2 Related work

Work related to HERA includes: FPGA-based custom designed machines, single-chip multiprocessor ASIC machines and mixed-mode machines.

FPGA-based machines: FPGA-based computing machines have demonstrated considerable performance gains over general-purpose microprocessors for many computation-intensive applications [8, 19]; they also have lower costs and more flexibility than ASIC designs. Most of them take advantage of the fine-grain architecture in earlier FPGAs and are fully customized for a specific class of applications. Due to limited resources in prior FPGAs, the fine-grain functional units in such machines most often were not program accessible and their overall processing capabilities were rather limited. Changes in the application often forced the development of a new FPGA configuration, a task foreign to end-application developers who are often versatile only in HLL-based coding. A small change in the algorithm requires the full reconfiguration of the hardware, which takes significant time. However, we expect this approach to continue playing a major role in application areas where field programmability by the user is not required or is needed rarely.

Moreover, most machines targeted bit-level multimedia and DSP applications, where floating-point operations are not often necessary. Because floating-point units consumed a very large proportion of resources in earlier FPGAs, very few machines supported floating-point arithmetic. As a result, most FPGA-based efforts in MMM [20-21] and LU factorization on systolic arrays [22] targeted matrices with fixed-point data. Recent advances in platform FPGAs have inspired floating-point implementations of MMM and LU factorization [2, 23, 41]. However, they followed the traditional approach where the circuitry is only applicable to the specific algorithms invented by the developer and the largest matrices studied were of very small size (8 x 8 and 48 x 48). In [2], full reconfiguration of the FPGA is needed for larger matrices. In contrast, HERA targets a large variety of high-performance, low-cost scientific and engineering problems involving large-scale matrices, where floating-point representation is needed to deal with large dynamic data ranges. HERA is reconfigured by the application program at run time via general-purpose instructions optimized for matrix computations. The ultimate goal of HERA is to dynamically match applications through architecture reconfiguration and resource management. Also, the architecture of platform FPGA devices facilitates coarse-grain datapaths in HERA in contrast to previous fine-grain designs.

Single-chip multiprocessor ASIC machines: Previous coarse-grain reconfigurable ASIC machines include MorphoSys [24], RAW [25] MATRIX [26], etc. These systems are mostly tailored to the needs of multimedia and DSP applications. Generally they follow the SIMD, MIMD or VLIW (Very-Long Instruction Word) model exclusively and datapaths are eight bits wide. Few of them have been actually implemented. Ref. [24] provides a comparison of such systems. HERA distinguishes itself from these systems in that it supports a variety of execution modes at run time and incorporates floating-point arithmetic as well; also, every PE is tightly coupled with plenty of local memory and shared on-board memory. Recent research efforts in this direction include Hydra [27], SCMP [28] and SCALE [6], etc. Hydra is designed around complex superscalar processors. SCALE combines vector processing and multithreading. In contrast, HERA's philosophy is to control the complexity of PEs in order to scale up the design, to reuse the resources through reconfiguration and to maximize the number of PEs. HERA bears some similarity with SCMP in that they both have a 2D-mesh layout and a tightly coupled local memory. While SCMP is a multiprocessor deprived of global communication channels, HERA contains a hierarchal bus system to efficiently support global communication. Of course, HERA features lower cost and lower design complexity than these ASIC approaches.

Mixed-mode machines: Several prototypes of mixed-mode systems [13] were implemented before the mid 1990's by employing COTS microprocessors. Recent years have not seen related research. In these systems, the processors serving as control units and the computing PEs were most often exclusively defined at static time and there was an interconnection network between these two groups. Thus, the sole assignment of a processor either as a control unit or a PE was fixed throughout execution. This can cause performance degradation due to inefficient utilization of the available resources. Also, these machines suffered from high communication overheads. Mixed-mode was implemented as a programming technique due to the employment of COTS components. In contrast, mixed-mode execution in HERA receives direct support from the PE's instruction set that sits on top of a suitable hardware design. Every PE in HERA is equipped with its own control unit so that not only the whole system is dynamically partitionable, like previous mixed-mode systems, but also the role of each PE can be changed dynamically at runtime by using an HERA instruction, as needed.

Our purpose here is to show that HERA is a novel architecture unifying the flexibility of reconfigurable logic, mixed-mode parallel execution and the programmability of microprocessors. The key features include: FPGA-based, coarse-grain and pipelined PEs supporting floating-point arithmetic, general-purpose instructions, mixed mode parallelism, tightly coupled local (on-chip) memory and flexible global communications. All these are at a low cost.

3 HERA: a reconfigurable mixed-mode parallel computer

3.1 System organization

Fig. 1 shows the general diagram of our HERA machine with *m* x *n* PEs interconnected via a 2-D mesh network. Most matrix-based computations are well structured and map naturally to the 2-D mesh which is easily implemented by FPGA place and route processes. We employ fast, direct NEWS (North, East, West and South) connections for communications between nearest neighbors. Nearest PE pairs on the same row or column can also communicate through one port of the data memory of the PEs to the west and north. Since every PE also has a Local Control Unit (LCU), most of the instruction decoding is carried out by the LCU. By giving the decoding work to the LCUs, we avoid broadcasting a large number of control signals to all the PEs. We still need global communication. Every column has a Cbus and all the Cbuses are connected to the Column Bus.

The interconnection network in multiprocessors unequivocally has a great impact on the performance. It must match substantially the communication requirements of the application; this was our objective when we designed HERA for matrix applications. The reconfiguration capability of FPGAs makes possible the evaluation of various communications schemes at low cost. This topic along with performance results are presented in a recently submitted paper for an IP-based multiprocessor design [36].

The total number of PEs in HERA is determined by the available resources in target FPGA devices and the resource requirements of the application. The computing fabric is controlled by the system Sequencer that communicates with the host processor via the PCI bus. Interrupt logic between the Sequencer and the host processor is implemented. The host can access the on-board DDR II SRAM and the on-chip memories of each PE. Each SRAM chip is owned exclusively by a group of PEs. The Global Control Unit (GCU), included in the system Sequencer, fetches instructions from the global program memory (GPM) for PEs operating in SIMD. Due to the presence of FPGAs, besides that the system operating mode is reconfigurable at runtime, the capabilities of each PE and the number of PEs can be easily reconfigured based on the application's requirements. The host can load different FPGA images in the same C/C++ code to finish different subtasks at runtime. Thus, FPGAs provide another dimension of flexibility to optimize the hardware to match the specific characteristics of the applications.

3.2 PE architecture

We adopted a RISC load-store architecture for our PE to save hardware resources. Fig. 2 shows the block diagram of the PE. All data paths are 32 bits. The PE contains several major components: a 7-stage, pipelined, 32-bit floating-point function unit (FFU), an LCU, 32-bit dual-port local program memory (LPM), 32-bit dual-port local data memory (LDM) and eight NEWS communication ports. Our HERA design implements IEEE 754 single-precision pipelined floating-point operations in each PE. We employed a 3-stage pipeline in the floating-point adder, subtractor and multiplier, and a 27-stage pipeline in the floating-point divider. Our design philosophy for instructions is to have a small and highly optimized instruction set for our target applications, while not losing generality for the sake of programming efficiency. The current set of about 30 instructions, shown in Table 1, can be classified into six major groups: *integer arithmetic*, *floating-point arithmetic (add/sub/mul/div)*, *memory access*, *jump* and *branch*, *PE NEWS communications and system control*. All the instructions follow a three-field general format and are 32 bits wide. The instructions support immediate, register and base addressing. HERA supports both global and local PE masking. Every PE in the processor array is assigned an ID number that serves in global masking. The last seven bits of all the instructions select a particular PE or a group of PEs. Every PE holds a mask bit and computes the mask value with every instruction. A specific bit in instructions selects between global and local masking. The destination register of *Get_N/E/W/S* instructions or the source register of *Send_N/E/W/S* instructions can also be one of the four NEWS_OUT registers. This way, data can bypass a PE to reach the next PE because we can use shared NEWS registers between PE pairs. Each PE comprises 32 32-bit general-purpose registers (GPRs) and several system registers: local instruction register (LIR), local program counter (LPC), data memory address register (DMAR), program memory address register (PMAR), local status register (LSR), local masking register (LMR) and 1-bit operating mode register (OMR). Similar to some other RISC processors, the R0 GPR is fixed at zero.

The operating mode of each PE is configured dynamically by the host processor through its OMR by using the *Configure* instruction: “0” indicates SIMD and “1” sets the PE into

MIMD. All PEs operate in SIMD when powered up. To switch a PE to MIMD from SIMD, the Sequencer first distributes the instructions to the LPM of the PE through the Column Bus and Cbus, and then sends a *JumpI* instruction to the PE with the starting address in the MIMD code. OMR is set to 1. To switch back to SIMD, OMR is reset to "0" and the PE then listens for the broadcasting of a global instruction. The data in the registers and memories remain intact during switching. The instructions come from GPM in SIMD and from LPM in MIMD. The masking in the SIMD mode can use the PE's ID number and/or LMR.

Table 1: The instruction set
#xxx: Memory address of 11 bits; **#*****: Immediate operand flag; **###**: PE ID number

Types	*Instructions*	*Descriptions*
Integer Arithmetic	ADD/SUB/MUL/DIV Rs, Rt, Rd	Integer add/sub/mul/div
	ADDI/SUBI/MULI/DIVI Rs/Rt, #***, Rd	Integer add/sub/mul/div with immediate data
Floating-point Arithmetic	FP_ADD/SUB/MUL/DIV Rs, Rt, Rd	Floating-point add/sub/mul/div
	FP_ADDI/SUBI/MULI/DIVI Rs/Rt, #***, Rd	Floating-point add/sub/mul/div with immediate data
Branch and Jump	JumpI #xxx	Jump to an immediate memory address
	JumpR offset(Rx)	Jump to the memory location at ((Rx) + offset)
	BNE Rs, Rt, #xxx	If (Rs) /= (Rt), then jump to the location at #xxx
Memory	LW (x) Rd, #xxx OR offset(Rx)	Load Rd with a 32-bit data from a memory address included in the instruction or computed by ((Rx) + offset) from memory source x (LDMs, LPM, SRAM)
	SW (x) Rs, #xxx OR offset(Rx)	Store 32-bit data in Rs to a memory address included in the instruction or computed by ((Rx) + offset) from memory source x (LDMs, LPM, SRAM)
PE Communication	Send_N/E/W/S Rs	Send the content of Rs to N/E/W/S_out port
	Get_N/E/W/S Rd	Get data from a N/E/W/S_in register into Rd
System Control	Sys_call (x)	Generate a soft call from PE(x) to the Sequencer and exchange information with the Sequencer.
	Standby	Pause and resume the execution of a PE
	Distribute ###, size	Distribute data to the LDM of every PE
	Collect ###, size	Collect data from the LDM of every PE
	Configure	Configure the computation mode and other system functions
	NOP	No operation

3.3 Memory configuration

The sizes of LPM and LDM are determined by the number of memory blocks in the FPGA device. They are configured as dual-ported 32-bit memories. Fig. 3 shows the connections for the two memory ports of LPM and LDM. We try to make the data memory as large as possible in order to reduce the data I/O time. The A port of LPM is connected to one Cbus, and serves as the interface to the Sequencer and host processor. This way, the host processor has access to all LPMs. It sends application programs to every PE through this port from the Main Memory if the PE is to operate in MIMD. Port B of an LPM can be accessed by the local PE to get the instructions. An interesting feature of our design is the LDM interface. Our matrix algorithms usually involve frequent accesses of intermediate results in nearest neighbors. The NEWS network can efficiently support single-word communication. However, it may take many cycles to transfer a large amount of data when using NEWS connections. Based on our experience with matrix algorithms on our IP-based multiprocessor machine, where PEs often use results from their east and north neighbors, we employed a shared dual-ported memory to address this problem. The A port of LDM is accessed by the local PE, and the B port is shared with the neighbors to the south and east. A PE can directly write to or read from the LDMs of its west and north neighbors via their B ports. Thus, we eliminate block data transfers between nearest neighbors. Another important use of the shared port is to pipe the data assigned to each PE into its LDM at system initialization. The A port of the LDM attached to the first PE on every row can also be accessed via the data bus.

4 Implementation and floating-point performance

Our first implementation was carried out on the high-performance WILDSTAR II-PCI FPGA board from Annapolis Micro Systems [29]. The board is populated with two Xilinx XC2V6000-5 Virtex II FPGA devices and 24MB of DDRII SRAM memory (12 chips). The board communicates with the host computer via a PCI bus interface. Every PE was assigned 4KB for LPM and 8KB for LDM. The interface to the PCI bus operates at 133MHz and the datapath is 64 bits wide. For the sake of comparison, similar to [23] we synthesized and implemented our design using the Xilinx ISE 5.2i toolset for an XC2VP125-7 FPGA. The divider in [23] uses a look-up table based reciprocator and a multiplier that result in precision errors. Table 2 shows that our FPU components generally result in higher frequency of operation; the overall latency and resource consumption are always smaller for our design.

Table 2: IEEE single-precision floating-point performance and resource utilization

Functional Unit		*Area (Slices)*	*Frequency (MHz)*	*Latency (Cycles)*
Add/Sub	XC2V6000-5	389	163.2	3
	XC2VP125-7	388	184.1	3
	XC2VP125-? [23]	402/425/520*	100/150/220*	6/12/16*
Multiplier	XC2V6000-5	135	172.5	3
	XC2VP125-7	135	199.5	3
	XC2VP125-? [23]	130/201/229*	100/150/220*	4/7/10*
Division	XC2V6000-5	915	172.2	27
	XC2VP125-7	923	197.9	27

* Based on three special-purpose designs for LU factorization

The completely integrated HERA system operates at 147MHz with the Xilinx ISE tools and 35 PEs fit in one XC2VP125 device that contains 55,616 slices. It becomes 125MHz on the Annapolis WILDSTAR II-PCI board. There are 33,792 slices available in the XC2V6000 FPGA. Our controller of the SRAM chips consumes 980 slices. Also, about 1550 slices are used by the Sequencer and the interface to the host, and the bus controller takes around 1750 slices. For each PE, the local control logic requires approximately 128 slices. Note that the resource utilization of the system components varies slightly with the system and PE configuration, as well as other implementation objectives (e.g., speed or area constraints). We removed the subtractor and divider from each PE in the case of MMM, thus allowing us to implement 64 PEs in the two FPGAs. For LU factorization with floating-point arithmetic, 36 PEs did fit in the two devices. Our hardware design was implemented in VHDL and can easily retarget other FPGA boards. About 50 APIs are implemented to facilitate the communication between the C/C++ application running on the host and the parallel program on HERA.

5 Programming

A comprehensive application development and programming platform is needed to efficiently take full advantage of HERA’s hardware benefits and improve productivity. Conventional parallel programming models based exclusively either on shared-memory or message-passing create a clear boundary between software and hardware. The seminal advantage provided by HERA is its flexibility to customize the architecture for a given application, so such a boundary is an obstacle. Moreover, low power has become a first-order priority that has to be taken into account in the customization process. We have not seen appropriate programming environments supporting mixed-mode parallel computing and power-aware adaptations. Nevertheless, customizable multiprocessing programming environments targeting chip multiprocessors (e.g., [33-34]) could be modified to work with HERA. We have done some relevant work for configurable systems in general. To support the implementation of conventional programming environments for configurable parallel systems, our design in [37] implements directly in hardware MPI (Message-Passing Interface) primitives; this creates a framework for efficient parallel code development involving data exchanges independently of the underlying hardware implementation. This process can also support the portability of MPI-based code developed for more conventional platforms (e.g., PC clusters) to configurable multiprocessors. Our design takes advantage of the effectiveness and efficiency of one-sided RMA (Remote Memory Access) communications.

We have recently focused our attention on the development of efficient resource management tools to maximize the performance of HERA for a given FPGA-application pair. An integrated framework has been created for HERA hardware optimization, and data-parallel application mapping, scheduling and execution. Preliminary results will appear in [35]. There are five major phases in this framework: task profiling where the behavioral description of the application is analyzed to construct a task flow graph, system synthesis, task coding using the target system’s instruction set, system implementation on FPGAs and dynamic, simultaneous resource-efficient task decomposition, mapping, scheduling and execution. The implementation on FPGAs follows the same procedure as any VHDL-based design methodology.

In addition, for a single HERA PE we could borrow ideas related to reconfigurable instruction set processors (RISPs) that contain reconfigurable logic in some of their functional units [39]. Code generation tools for RISPs could be used to create and evaluate reconfigurable instructions and select those that minimize reconfiguration time [38]. Several RISP HLL compilers are presented in [38]. The next section goes directly into the mapping problem for two specific applications, namely MMM and LU factorization of sparse DBBD matrices.

6 Mapping Applications onto HERA

6.1 Generalized Cannon's MMM algorithm

6.1.1 Data partitioning and mapping

Cannon's MMM algorithm [15] is for a memory efficient parallel implementation on torus-connected processor arrays, where each processor communicates directly with its immediate neighbors in the four NEWS connections directions. The original algorithm assumes that the input matrices and the partitioned matrix blocks are all square. In our implementation, however, matrices ***A*** and ***B*** for ***A*** x ***B*** can be of any shape and size (still, the number of rows in ***A*** and the number of columns in ***B*** should be the same).

Assume PEs are organized in a *q* x *q* 2-D torus. Let ***A*** and ***B*** be matrices of size *N1* x *N2* and *N2* x *N3*, respectively. We assume that the on-chip memory can store $3m^2$ floating-point elements. To be able to store complete blocks from the input and output matrices, the maximum size of a matrix block should be *m* x *m*. Let $p1 = \lfloor N1/(q*m) \rfloor$, $p2 = \lfloor N2/(q*m) \rfloor$ and $p3 = \lfloor N3/(q*m) \rfloor$. In general, we first partition ***A*** and ***B*** into a 2 x 2 block-based matrix as shown in the example of Fig. 4, in such a way that the sizes of ***A(1,1)*** and ***B(1,1)*** are $\{p1*(q*m)\} \text{ x } \{p2*(q*m)\}$ and $\{p2*(q*m)\} \text{ x } \{p3*(q*m)\}$, respectively. The remaining blocks ***A(2,1)***, ***A(1,2)*** and ***A(2,2)*** of ***A*** are decomposed into blocks with maximum dimension *m*. ***B*** is partitioned similarly. Blocks ***A(1,1)** and **B(1,1)*** are then partitioned into $p1 \text{ x } p2$ and $p2 \text{ x } p3$ blocks of size $(q*m) \text{x} (q*m)$ again and are distributed into the processors in a cyclic checkerboard-like fashion.

6.1.2 Mixed-mode scheduling on HERA

If ***A*** and ***B*** are square, and can be partitioned into an integer multiple of *q* blocks, then Cannon's algorithm works best in the SIMD mode; all the PEs are then busy all the time, except during the initial alignment. If ***A*** and ***B*** are not square or cannot be partitioned in such a way that *N* (the matrix dimension) is a multiple integer of *q* * *m*, then the multiplication of the border blocks is not efficient in the SIMD mode since the sizes and numbers of blocks are irregular. Then, some PEs are idle while other PEs are busy at some point because SIMD is an implicitly synchronous mode. We solved this problem by changing the computation mode of the PEs. Also, we skip the initial alignment by assigning data blocks in a pre-skewed way. Because our PE is pipelined, we assume that multiplication, addition and shift operations all take one clock cycle, T_{clk}. The total execution time for Cannon's procedure on one partition is approximately $T_c(n) = q*(2T_{shift} + T_{mul} + T_{add}) = 2(\frac{n^3}{q^2} + \frac{n^2}{q})*T_{clk}$, assuming that the size of the submatrix is *n* x *n*.

The dynamic mixed-mode scheduling procedure for our modified Cannon's algorithm on HERA is as follows. (1) Carry out block multiplications involving ***A(1,1)*** * ***B(1,1)*** by using Cannon's algorithm; the total time is about $p1*p2*p3*T_c(n)$. All the PEs are configured into SIMD and take part in this step. (2) If the size of ***A(1,2)*** and/or ***B(2,1)*** is larger than ½($q*m$), then carry out ***A(1,1)*** * ***B(1,2)*** and/or ***A(2,1)*** * ***B(1,1)*** in SIMD using Cannon's procedure. (3) We define a job as a multiplication of two blocks. Jobs are divided into two groups: SIMD and MIMD jobs. SIMD jobs are those corresponding to similar numbers of operations on the PEs. The remaining jobs go to an MIMD queue. Count the number of jobs and their associated numbers of operations in the remaining work. Determine the IDs of PEs that will work in the SIMD or MIMD mode based on the job information. (4) Configure individual PEs in the system into either the SIMD or MIMD mode based on the decision in the previous step. The system now works in the mixed mode. Assign the SIMD jobs to the PEs running in the SIMD mode and distribute the MIMD jobs to the PEs running in the MIMD mode.

For the calculation of the quadrants in the resulting matrix, ***A(1,1)*** * ***B(1,1)***, ***A(1,1)*** * ***B(1,2)*** and ***A(2,1)*** * ***B(1,1)*** consume most of the execution time. In all the steps, except Step 1, data locality has priority in job assignments.

6.1.3 Performance results and analysis

We first implemented on HERA our mixed-mode MMM scheduling for square matrices of size up to 1000 x 1000. In this experiment, the PEs form an 8 x 8 mesh. Our target application is data intensive and the size of the core computation code for the PEs is less than 2KB. In order to reduce the data I/O time, the PEs on a row share an on-board SRAM chip for data storage and the eight on-board SRAM chips can work in parallel. The remaining on-board SRAM chips service exchanges with the Sequencer. Data are pumped into the LDM memory of PEs on the same row in a pipelined fashion. The LPM of the PEs is also used for data storage in SIMD. The on-chip memory used for data storage is divided into three sections for the two input blocks and the results, respectively. Cannon's algorithm is very efficient in data reuse and we do not have to stop the computation to perform data transfers during the application of Cannon's basic procedure. At the end of this procedure, we have to collect the results and redistribute new blocks to the PEs. The largest matrix block for this procedure has size $q * m$ and the total number of transferred items during I/O is $p1 * p2 * p3$. From the equations for these parameters, we can see that the performance is limited by the size of the LDM memory. The experimental results on HERA are plotted in Fig. 5. Previous MMM work on FPGAs targeted fixed-point data and the floating-point performance on platform FPGAs in [2] was shown for a very small 8 x 8 matrix; therefore, we cannot compare HERA's performance with other related work on FPGAs. We coded instead in C block-based MMM for a commercial PC with a 2GHz Intel Pentium 4 processor; comparative results are included in Fig. 5. The code for the PC was optimized with various techniques, such as using best block sizes for the L1 and L2 caches, compiler flags and copy optimization. Nevertheless, it is fair to compare the performance of architectures implemented with similar silicon processes. The FPGA in our experiments (i.e., the XC2V6000), was manufactured with a 0.12um/8-layer process and was released in early 2001. The Pentium III 1GHz processor was manufactured with similar technology. The Intel MKL [43] library is one of the best for matrix operations; it is highly tuned to the architecture of the Intel processors. We could not find related work on the performance of MKL on the Pentium III. We show instead in Fig. 5 the MKL-based performance of a 1.5GHz Pentium 4 processor on double-precision MMM [42]. If we assume that the best-case performance for single-precision MMM is double that for double-precision MMM and we also scale appropriately the results in [42] to target the 1GHz Pentium III processor, we can find out that HERA's performance is worse by about 30% in the worst case. However, this finding should be expected given the fact that MKL is a highly optimized library for the Intel architecture. Also, if we consider the fastest and largest FPGA device in the same family (i.e., the XC2V8000-5), the FPGA's expected performance is an improvement of about 30%. We expect dramatic performance improvement in the near future by employing more advanced FPGAs, such as the recently released Virtex 4 XC4VLX200 that doubles the logic and the on-chip memory resources and can run at 500MHz. [1] showed that significant growth of floating-point performance is favored by FPGA realizations rather than general-purpose microprocessors; actually, the former surpassed the latter in performance around 2003-2004. Recent double-precision matrix multiplication results on the XC2V125-7, the most advanced FPGA in the Virtex II Pro family, are based on synthesis without actual FPGA runs [41]; they demonstrated 15.6GFLOPs performance with a system frequency of 200MHz and a 400MB/s bandwidth. Recent DSP results showed a sustained performance of about 700MFLOPS for the 300MHz TMS320C6713 (released in 2004) [40]. Of course, a major drawback of DSP processors is their much higher energy consumption as compared to FPGAs [23].

The speedup of the parallel implementation on the 64-PE HERA over the sequential one on the 1-PE HERA is shown in Fig. 6. The speedup for the 100 x 100 case is much lower because the ratio of computation to communication times is much lower than for the other cases. We could improve the speedups further in all cases by using a bigger local memory since the complexity of multiplication on a single PE for a pair of blocks is O(N^3) and that of communication (i.e., shifting) is O(N^2), where N x N is the block size. With increases in the problem size, the speedup and, of course, the efficiency stabilizes in a very narrow range. We also evaluated the performance of our mixed-mode scheduling for a variety of non-square matrices. The multiplication of irregular matrices is required in the parallel LU factorization of sparse DBBD matrices (described in the next section). In this algorithm, the factored border blocks in each 3-block group, which are irregular in size, are multiplied to generate an intermediate matrix block of the same size of the last block, which is accumulated and added to the last block before its factorization begins. SIMD mappings, where all the PEs work in the SIMD mode all the time were also implemented for these matrices; the results are shown in Table 3. These real-world matrices are used to solve power flow equations. From this table, we can see that our dynamic mixed-mode scheduling can greatly boost performance when the multiplication of irregular matrices is needed.

Table 3: HERA execution times for irregular matrices under different execution modes
(Clock frequency: 125MHz)

Matrix Dimensions			*HERA in SIMD mode, sec*	*HERA in mixed-mode, sec*	*Improvement, %*
N1	N2	N3			
105	101	113	0.0115	0.0103	10.1
201	215	323	0.0864	0.0723	16.3
324	599	315	0.3699	0.3366	9.8
05	611	613	0.9254	0.7853	15.1
509	301	201	0.2163	0.1894	12.4
677	202	677	0.5037	0.4749	5.7
711	713	403	1.3344	1.1584	13.2
955	957	976	5.6077	5.1872	7.5

6.2 Parallel LU factorization of DBBD matrices

6.2.1 Overview of the algorithm

LU factorization is a widely employed direct method that solves a large system of simultaneous linear equations presented in the form $\boldsymbol{Ax} = \boldsymbol{b}$; A is an N x N nonsingular matrix, $\boldsymbol{x}$ is a vector of N unknowns and $\boldsymbol{b}$ is a given vector of length N. It works as follows. We first factorize $\boldsymbol{A}$ so that $\boldsymbol{A} = \boldsymbol{LU}$, where $\boldsymbol{L}$ is a lower triangular matrix and $\boldsymbol{U}$ is an upper triangular matrix. Their elements can be determined by $L_{ij} = (A_{ij} - \sum_{k=1}^{j-1} L_{ik} * U_{kj}) * \frac{1}{U_{jj}},\ for\ j \in [1, i-1]$ (1) and $U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik} * U_{kj},\ for\ j \in [i, N]$ (2), respectively, if $\boldsymbol{L}$ is to have all 1's on its diagonal. Once $\boldsymbol{L}$ and $\boldsymbol{U}$ are formed, the unknown vector $\boldsymbol{x}$ can be identified by forward reduction and backward substitution, respectively, using the two equations $\boldsymbol{Ly} = \boldsymbol{b}$ and $\boldsymbol{Ux} = \boldsymbol{y}$.

Since LU factorization is a computation-intensive procedure, its parallel solution has been a quite active research area. Unfortunately, our early research [16] has revealed that for the large sparse matrices, such as those appearing in power engineering most of the current parallel solvers perform poorly; these solvers soon make these matrices dense because of fill-ins (i.e., zero elements that receive new values) during the factorization. By taking advantage of the physical characteristics in power matrices, we can reorder the power matrices into the DBBD form shown in Fig. 7 by applying a heuristics-based partitioning scheme [30].

In the DBBD form, the ***Aik***'s represent matrix sub-blocks and all the non-zero elements in the matrix appear only inside these sub-blocks. For every fixed *i*, the blocks ***Aii***, ***Ain*** and ***Ani*** are said to form a 3-block group, where $i \in [1, n\text{-}1]$ and $n \le N$. ***Ann*** is known as the last block. The ***Aii***'s will be referred to as the diagonal blocks, and ***Ain*** and ***Ani*** will be called right border block and bottom border block, respectively, where $i \in [1, n]$. The sizes of all the blocks after ordering are determined by the physical characteristics of the matrices and the ordering parameters, such as the maximum number of nodes in a block. Since all non-border, off-diagonal blocks contain only 0's, if we apply Eqs. (1) and (2) to a DBBD matrix, we can find out that there will be no fill-ins in these blocks during factorization. Thus, the resulting matrix keeps the same DBBD form, as shown in Eq. (3).

$$\left\{\begin{bmatrix} L_{11} & 0 & 0 & 0 & 0 \\ 0 & L_{22} & 0 & 0 & 0 \\ 0 & 0 & L_{33} & 0 & 0 \\ 0 & 0 & 0 & \ldots & \ldots \\ L_{n1} & L_{n2} & L_{n3} & \ldots & L_{nn} \end{bmatrix} \begin{bmatrix} U_{11} & 0 & 0 & 0 & U_{1n} \\ 0 & U_{22} & 0 & 0 & U_{2n} \\ 0 & 0 & U_{33} & 0 & U_{3n} \\ 0 & 0 & 0 & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & U_{nn} \end{bmatrix}\right\} \quad (3)$$

where

$$A_{kk} = L_{kk}U_{kk}$$
$$U_{kn} = L_{kk}^{-1}A_{kn}$$
$$L_{nk} = A_{nk}U_{kk}^{-1},\ for\ k \in [1, n-1]$$
$$L_{nn}U_{nn} = A_{nn} - \sum_{k=1}^{n-1} L_{nk}U_{kn}$$

The calculations of L_{kk}, U_{kk}, L_{nk} and U_{kn} for different *k*'s (i.e., 3-block groups) are independent of each other. So we can distribute different 3-block groups to different processors to be factored in parallel, with no data exchanges until the factorization of ***Ann***. The last block, ***Ann***, requires data produced in all the right and bottom border blocks, so its factorization is the last step. We can see that the sparse DBBD matrix format presents great advantages for parallel implementation. In general, the DBBD-based parallel LU factorization includes four types of jobs: (1) *FAC*: Independent factorization of all the 3-block groups. (2) *MAC*: Independent multiplication of the factored border block pairs ($\boldsymbol{L_{nk}U_{kn}}$) and (local) accumulation of the partial products inside each PE to later produce the inner product $\sum_{k=1}^{m_i-1} L_{nk}U_{kn}$, where m_i is the total number of 3-block groups assigned to PEi and $\sum_{i=1}^{36} m_i = n-1$. Every resulting product has the same size as ***Ann***. (3) *PAC*: Parallel (global) accumulation of the partial products in all PEs after no more FAC or MAC tasks are left. (4) *LAST*: Parallel LU factorization in ***Ann*** upon finishing all other factorization and multiplication work; this work begins with the synchronization of the involved PEs. The last block is normally dense. Complete details of this algorithm can be found in [16].

6.2.2 Mixed-mode scheduling on HERA

The parallel LU factorization of sparse DBBD matrices involves irregular computation patterns and blocks of various sizes, as the result of the physical characteristics of the original matrices. However, many parts in the algorithm could still benefit from an SIMD implementation. As a natural consequence, a combination of appropriate parallel execution modes should give better results.

For this application, our HERA machine comprises 36 PEs, which are configured in a 6 x 6 mesh. To map an application algorithm onto a mixed-mode system, the main focus is on identifying the optimal mode of parallelism for each subtask. We should also take into account the costs incurred when switching between different pairs of modes: SIMD/MIMD, SIMD/M-SIMD and MIMD/M-SIMD. The following is the general scheduling procedure to carry out the parallel LU factorization of DBBD matrices on our mixed-mode machine.

Step 1 (SIMD & M-SIMD) Identify 3-block groups of comparable size and put them into different task queues. Divide and configure the system into M-SIMD based on the task information. Assign 3-block groups from each queue to the PEs working in the same SIMD group, and perform the *FAC* and *MAC* work on these groups until the number of remaining 3-block groups is less than the number of PEs (i.e., 36).

Step 2 (M-SIMD) Assign the remaining 3-block groups so that groups of comparable size go to the same column of PEs (see Fig. 1) and every PE has the largest possible number of idle nearest neighbors. This is an effort to facilitate the subsequent *PAC* work. If necessary, reconfigure the system into a different M-SIMD layout.

Step 3 (M-SIMD & MIMD) A PE is reconfigured into MIMD as soon as it finishes its work and no more 3-block group is waiting in the task queue.

Step 4 (M-SIMD & MIMD) Assign each PE in MIMD to the multiplication of a pair of (row and column) factored border blocks. Since the LDM has a shared port with its east and south neighbors, every idle PE will help its neighbors after it finishes its own work; no data transfer incurs in this process.

Step 5 (SIMD) After the factorization of all the 3-block groups and the multiplication of factored border blocks, reconfigure all the PEs again into the SIMD mode to carry out the *PAC* work.

Step 6 (SIMD) Factor the last block in the SIMD mode.

Fig. 8 shows a typical PE mode assignment in the above procedure for large DBBD matrices. When the number of tasks in one or more task queues is larger than 36, we begin with one or more single SIMD configurations, which is a special case of M-SIMD in Step 1.

6.2.3 Performance results and analysis

Experiments implementing the SIMD, MIMD and mixed-mode scheduling schemes were performed on the 36-PE HERA machine. Table 4 shows the characteristics of the test matrices from the Harwell-Boeing Collection in the Matrix Market [31]. The last matrix corresponds to a power network in North America. The performance results under these parallel execution modes are presented in Fig. 9. It is obvious that mixed-mode parallelism consumes less time for all the matrices and the advantage is more significant when the 3-block groups are highly irregular in size and shape, such as for the matrices of dimension

1723 and 7917. For the former (i.e., 1723) matrix, speed ups of 19.1% and 15.5% are obtained compared to the SIMD and MIMD implementations, respectively.

Table 4: Characteristics of the test matrices ordered into the DBBD form

Matrix	*PSADMIT*	*PSADMIT*	*BCSPWR09*	*BCSPWR10*	*N/A*
Dimension (*N*)	494	1138	1723	5300	7917
Total diagonal blocks (*n*)	27	67	42	125	51
Dimension of the largest diagonal block	20	20	50	50	198
Dimension of the smallest diagonal block	11	4	29	30	84
Dimension of the last diagonal block	45	100	134	577	517
Distribution of block sizes**	13(20),8(15), 5(12), 1(8)	22(20),26(15), 18 (12), 1(4)	10(50),14(40), 18(30)	30(50),40(40), 55(30)	12(180),12(150), 26(120), 1(84)

* NNZ: number of non-zero elements
**13(20) stands for 13 diagonal blocks of size close to 20 x 20.

Under the SIMD mode, some PEs are idle during the factorization of the 3-block groups and the multiplication of the border blocks. The total execution time is $T_{simd} = \sum_{i=1}^{m} T1_i\,(FAC + MUL) + \lceil \log_2 p \rceil T2 + T_{last}$, where $m = \left\lceil \frac{n-1}{36} \right\rceil$ and n is the total number of independent diagonal blocks. $T1_i\,(FAC + MUL)$ is the maximum execution time among the PEs for the i^{th} iteration of jobs. $T2$ is the time for a PE to perform one addition and one communication during the PAC work. T_{last} corresponds to the execution time for the last block. In MIMD, the *PAC* work may begin while some PEs are still working on *FAC* or *MAC* tasks. The worst case execution time is $T_{mimd} = \max_{1 \le PE_j \le 36} \{\sum_{i=1}^{m_j} T1_i\,(FAC + MAC)\} + \lceil \log_2 p \rceil T2 + T_{last}$, where $T1_i\,(FAC + MAC)$ is the execution time of the i^{th} iteration for PE$_j$ that processes m_j 3-block groups. From the equations, it is easy to see why MIMD performs better than SIMD for the matrices of dimension 1723, 5300 and 7917, and worse than SIMD for the rest of matrices. A disadvantage of this algorithm in MIMD execution is that half of each PE's LPM space is used for instructions despite the fact that the PEs run identical instructions most of the time.

This space could be used instead for data storage to reduce data transfer times. Another factor causing performance degradation as compared to mixed-mode execution is the increased number of instructions for synchronization. The processing of the last block is more efficient under SIMD than MIMD. Nevertheless, it contributes substantially to the total execution time. The last block requires a significant amount of data communications among PEs; the implicit synchronization in SIMD reduces the communication time. Fortunately, however, HERA's communication overhead in MIMD is not significantly higher than that for SIMD. However, MIMD tends to perform better than SIMD in this algorithm for large matrices where we have a good chance that matrix blocks are more irregular and sparse. Due to insufficient work, all modes perform comparably for the 494 x 494 matrix.

Previous relevant work has primarily focused on fixed-point implementations of LU factorization. However, the implementation in [23] involves FPUs as well and DSP processor comparisons. A circular linear array architecture is implemented, specifically for LU factorization. To resolve data dependencies, [23] assumes a stream of *s* "stacked" dense matrices; *s* has to be larger than the combined latency (in cycles) of the multiplier and subtractor units. The total latency is $s * n + s * n^2$ for n x n matrices and the throughput is one matrix per $n + n^2$ cycles. Also, shift registers are inserted in the datapaths that bypass the FPUs and the control logic must be able to delay the control signals. This design is inflexible since the number of processors and their storage space are specific to the matrix size.

We have to emphasize here that [23] attempts to maximize the throughput because it is an application-specific programmable circuit (ASPC) whereas HERA is a fully-programmable system. Results in [23] were reported for s = 10, 19 and 25. The latencies as shown in [23] for non-optimized code on the TMS320C6711 DSP processor are included in Table 5; matrices of various sizes were considered. Note that [23] only shows the "*effective* latency" which is actually the inverse of the throughput for processed matrices. It can be deduced that HERA's performance is much better than that of both systems.

Table 5. Latency comparison also involving a DSP processor (latency in ms)

Matrix Size	*Design in [23]* *f = 100MHz*		*HERA* *f = 147MHz*		*TMS320C6711 [23]* *f = 150MHz*
Number of matrices	1*	100	1**	100	1
8 x 8	0.06	6	0.052	0.156	11.5
16 x 16	0.26	26	0.431	1.293	23.0
24 x 24	0.58	58	1.210	3.630	55.2
32 x 32	1.02	102	2.920	8.760	87.5

* Average latency based on a stream of 100 matrices [23]. ** Single matrix

There are 35 PEs in HERA running at 147MHz (for the Xilinx ISE tools) and each PE can complete three floating-point operations per cycle. Therefore, HERA has a peak performance of 15.44GFLOPs whereas the performance of the TMS320C6711x family is 600-1500MFLOPs (for 100-250MHz frequencies) [32]. There are often more disadvantages to DSP processors that rely heavily on the clock frequency for high performance. FPGAs can offer much more flexibility in memory hierarchy and configuration, system architecture and processor microarchitecture, data formats, interconnection networks, etc.; an FPGA-based design can be optimized based on various metrics such as energy, throughput, latency, area, design time, budget, etc.

7 Conclusions

We have presented the design and implementation of our HERA architecture on platform FPGAs. HERA targets computation-intensive matrix operations. It takes advantage of reconfigurable logic and is flexible enough to allow for a mixed execution of various parallel processing modes, including SIMD, MIMD and M-SIMD. Performance results for MMM and the LU factorization of sparse DBBD matrices demonstrate that mixed-mode execution can match better dynamically the requirements of application algorithms throughout their lifespan. Thus, mixed mode can deliver much better performance than the pure SIMD and

MIMD execution modes. The results also show that the system efficiency stabilizes with increases in the problem size, which proves the good scalability of our architecture for these algorithms. Many applications can benefit from the high-performance and flexibility resulting from dynamic mixed-mode scheduling; this is especially true for programs rich in conditional and non-deterministic operations. Our work shows that new-generation FPGAs have made feasible the building of highly parallel complex systems supporting floating-point arithmetic. Of course, another major advantage is that these systems are easily accessible and portable. The ultimate goal of HERA is to maximize the performance by tuning the architecture to match the application through dynamic resource management and reconfiguration of FPGAs. With the anticipated speed and density improvements for FPGAs, low cost reconfigurable computers can enter mainstream parallel computing because they have the potential to narrow the growing performance gap between applications and chips.

8 References

1 Underwood, K.: ‘FPGAs vs. CPUs: trends in peak floating-point performance’, *12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, Feb. 2004, pp. 171-180.

2 Zhuo, L., and Prasanna, V. K.: ‘Scalable and modular algorithms for floating-point matrix multiplication on FPGAs’, *18th International Parallel and Distributed Processing Symposium*, April 2004, pp. 92-101.

3 Liang, J., Tessier, R., and Mencer, O.: ‘Floating point unit generation and evaluation for FPGAs’, *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003, pp.185 – 194.

4 Hannig, F., Dutta, H., and Teich, J.: ‘Regular mapping for coarse-grained reconfigurable architectures’, *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol.V, Montréal, Canada, May 2004, pp. 57-60.

5 Ronen, R., Mendelson, A., Lai, K., Lu, S-L., Pollack, F., and Shen, J.: 'Coming challenges in microarchitecture and architecture', *Proceedings of the IEEE*, March 2001, **89**, (3), pp. 325-340.

6 Krashinsky, R., Batten, C., Hampton, M., Gerding, S., Pharris, B., Casper, J., and Asanovic, K.: 'The vector-thread architecture', *IEEE 31st International Symposium on Computer Architecture,* Munich, Germany, June 2004, pp. 52-63.

7 Bergamaschi, R., Bolsens, I., Gupta, R., Harr, R., Jerraya, A., Keutzer, K., Olukotun, K., and Vissers, K.: 'Are single-chip multiprocessors in reach? ', *IEEE Design & Test of Computers*, Jan.-Feb. 2001, **18**, (1), pp. 82 – 89.

8 Compton, K., and Hauck, S.: 'Reconfigurable computing: a survey of systems and software', *ACM Comput. Surveys*, June 2002, **34**, (2), pp. 171-210.

9 Khawam, S., Arslan, T., and Westall, F.: 'Synthesizable reconfigurable array targeting distributed arithmetic for system-on-chip applications', *12th Reconfigurable Architectures Workshop, 2004*.

10 Tyrrell, A.M., Krohling, R. A., and Zhou, Y.: 'Evolutionary algorithm for the promotion of evolvable hardware', *IEE Proceedings-Computers and Digital Techniques*, July 2004, **151**, (4), pp. 267-275.

11 Parhami, B.: 'SIMD machines: do they have a significant future?' *Report on a Panel Discussion, 5th Symposium Frontiers Massively Parallel Computation,* McLean, LA, Feb. 1995, pp. 19-22.

12 Meilander, W. C., Baker, J. W., and Jin, M.: 'Importance of SIMD computation reconsidered', *17th IEEE Int'l Parallel Distributed Processing Symp. (IPDPS2003),* April 2003, pp. 266-273.

13 Siegel, H. J., Maheswaran, M., Watson, D. W., Antonio, J. K., and Atallah, M. J.: ‘Mixed-mode system heterogeneous computing’, in *Heterogeneous Computing*, Eshaghian, M. M. (Ed.), Artech House, Norwood, MA, 1996, pp. 19-65.

14 Duff, I. S., Erisman, A. M., and Reid, J. K.: *Direct methods for sparse matrices*, Oxford Univ. Press, Oxford, England, 1990.

15 Cannon, L. E.: ‘*A cellular computer to implement the kalman filter algorithm*’, Ph.D. Thesis, Montana State University, 1969.

16 Wang, X., and Ziavras, S. G.: ‘Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines’, *Concurrency and Computation: Practice and Experience*, 2004, **16**, (4), pp. 319-343.

17 Wang, X., and Ziavras, S. G.: ‘Parallel direct solution of linear equations on FPGA-based machines’, *11th IEEE International Workshop on Parallel and Distributed Real-Time Systems (Proc. of 17th IEEE International Parallel and Distributed Processing Symposium)*, Nice, France, April 22-26, 2003.

18 Wang, X., and Ziavras, S. G.: ‘HERA: A reconfigurable and mixed-mode parallel computing engine on platform FPGAs’, *16th International Conference on Parallel and Distributed Computing and Systems*, Boston, Massachusetts, November 9-11, 2004, pp. 374-379.

19 Tessier, R., and Burleson, W.: ‘Reconfigurable computing and digital signal processing: a survey’, *J. VLSI Signal Proc.*, May/June 2001, pp. 7-27.

20 Bensaali, F., Amira, A., and Bouridane, A.: ‘An FPGA based coprocessor for large matrix product implementation’, *2003 IEEE International Conference on Field-Programmable Technology*, Dec. 2003, pp. 292-295.

21 Yi, Y., Woods, R., and McCanny, J. V.: ‘Hierarchical synthesis of complex DSP functions on FPGAs’, *37th Asilomar Conference on Signals, Systems & Computers*, Vol. 2, Nov. 2003, pp.1421-1425.

22 Rajopadhye, S. V.: ‘Systolic arrays for LU decomposition’, *IEEE International Symposium on Circuits and Systems*, Vol.3, June 1988, pp. 2513-2516.

23 Govindu, G., Choi, S., Prasanna, V. K., Daga, V., Gangadharpalli, S., and Sridhar, V.: ‘A high-performance and energy-efficient architecture for floating-point based LU decomposition on FPGAs’, *12th Reconfigurable Architectures Workshop,* April 2004.

24 Singh, H., Lee, M.-H., Lu, G., Kurdahi, F. J., Bagherzadeh, N., and Filho, E. M. C.: ‘MorphoSys: an integrated reconfigurable system for data-parallel and computation-Intensive Applications’, *IEEE Transactions on Computers,* May 2000, **49**, (5), pp. 465-481.

25 Taylor, M. Kim, B., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, Hoffmann, B., Johnson, H., P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Frank, V. S. M., Amarasinghe, S., and Agarwal, A.: ‘The RAW microprocessor: a computational fabric for software circuits and general purpose programs’, *IEEE Micro*, Mar/Apr 2002, pp. 25-35.

26 Mirsky, E., and DeHon, A.: ‘MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources’, *1996 IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996, pp. 157-166.

27 Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., and Chang, K.: ‘The case for a single-chip multiprocessor’, *Seventh International Symp. Architectural Support for Programming Languages and Operating Systems*, Oct. 1996, pp. 2-11.

28 Baker, J. M. Jr., Bennett, S., Bucciero, M., Gold, B., and Mahajan, R.: 'SCMP: a single-chip message-passing parallel computer', *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 2002, pp.1485-1491.

29 Annapolis Micro Systems, Inc., http://www.annapmicro.com/.

30 Sangiovanni-Vincentelli, A., Chen, L. K., and Chua, L. O.: 'An efficient heuristic cluster algorithm for tearing large-scale networks', *IEEE Transactions on Circuits and Systems*, December 1977, **24**, (12), pp. 709-717.

31 Matrix Market, *http://math.nist.gov/MatrixMarket/*.

32 TMS320C6711/11B/11C/11D Floating-Point Digital Signal Processors, http://focus.ti.com/docs/prod/folders/print/tms320c6711.html.

33 Codito Technologies Pvt. Ltd., http://www.codito.com/prodtech_framework.html.

34 OpenMP, http://www.openmp.org.

35 Wang, X., and Ziavras, S. G.: 'A framework for dynamic resource management and scheduling on reconfigurable mixed-mode multiprocessors', *IEEE International Conference on Field-Programmable Technology*, Singapore, December 2005, pp. 51-58.

36 Wang, X., and Ziavras, S. G.: 'A multiprocessor-on-a-programmable-chip reconfigurable system for matrix operations with power-grid case studies', *International Journal of Computational Science and Engineering, Special Issue on Parallel and Distributed Scientific and Engineering Computing*, accepted for publication.

37 Ziavras, S. G., Gerbessiotis, A., and Bafna, R.: 'Coprocessor design to support MPI primitives in configurable multiprocessors', *Integration, the VLSI Journal*, accepted for publication.

38 Barat, F., Rudy, L., and Geert, D.: ‘Reconfigurable instruction set processors from a hardware/software perspective’, *IEEE Transactions on Software Engineering*, 2002, **28**, (9), pp. 647-662.

39 Tensilica, http://tensilica.com.

40 Wunderlich, R., Püschel, M., and Hoe, J.: ‘Accelerating blocked matrix-matrix multiplication using a software-managed memory hierarchy with DMA’, *High Performance Embedded Computing Workshop*, MIT, 2005.

41 Dou, Y., Vassiliadis, S., Kuzmanov, G. K., and Gaydadjiev, G. N.: ‘64-bit floating-point FPGA matrix multiplication’, *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2005, pp. 86-95.

42 J. Demmel, and Yelick, K.: ‘Automatic performance tuning of linear algebra kernels’, *TOPS-SciDAC (http://www.tops-scidac.org)*, January 2002, http://bebop.cs.berkeley.edu/pubs/SciDAC_250102.pdf.

43 Intel Math Kernel Library (MKL) 8.0, *http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/219823.htm*.

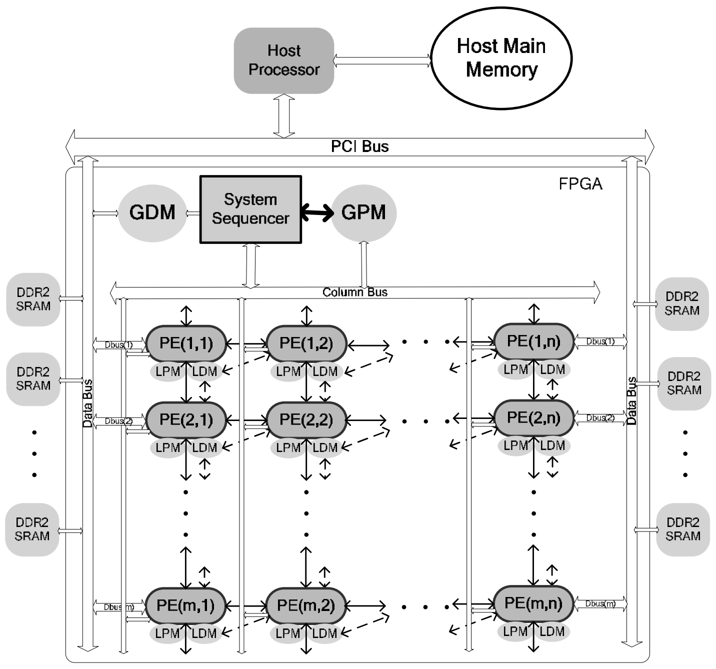

Fig. 1 *HERA system architecture*

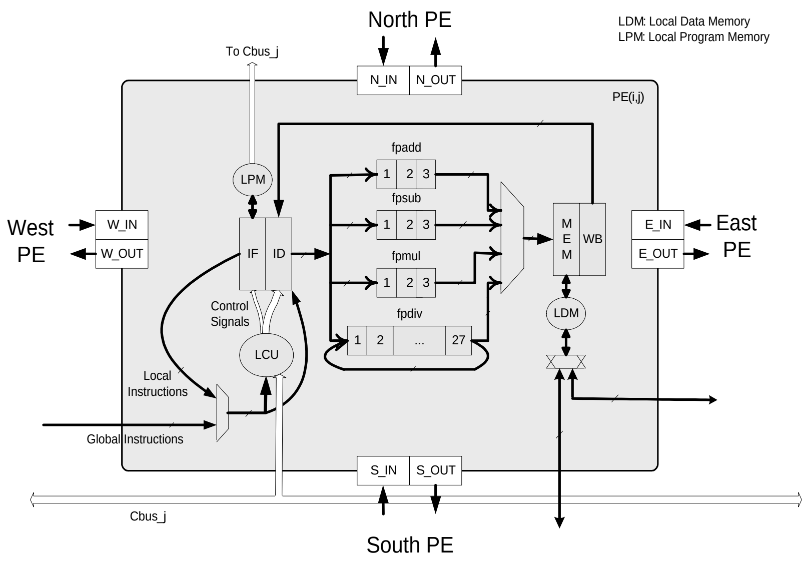

Fig. 2 *A HERA PE*

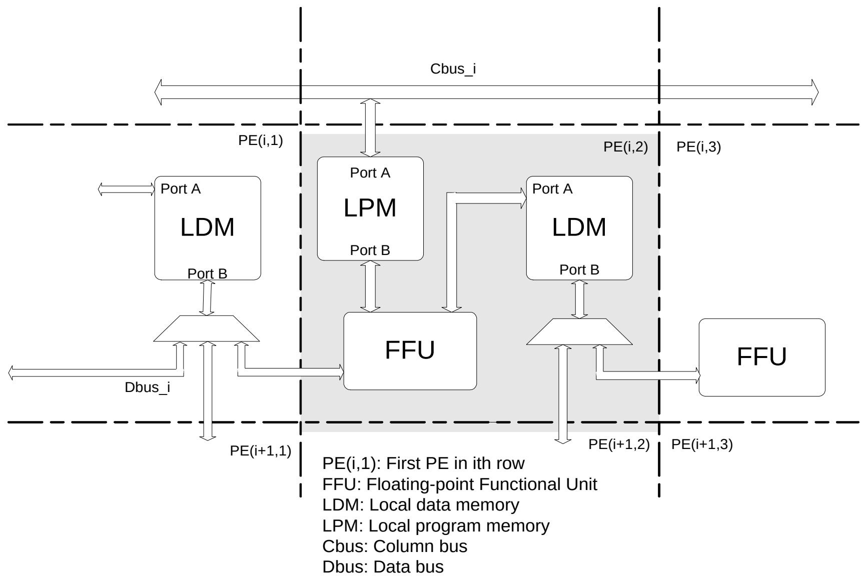

Fig. 3 *Memory interface*

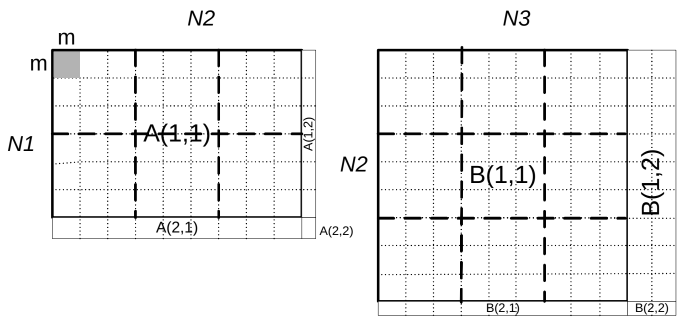

Fig. 4 *A partitioning example for matrices A and B* $(q = 3,\ p1 = 2,\ p2 = 3,\ p3 = 3)$

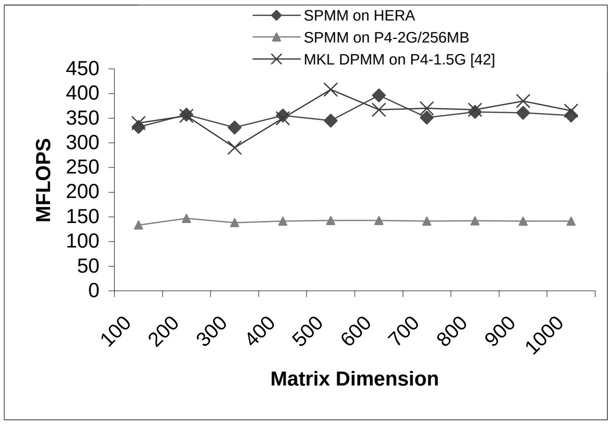

Fig. 5 *Performance comparison of our MMM implementation on HERA and a Dell PC, and an MKL implementation [42] (SPMM: Single-Precision Matrix Multiplication; DPMM: Double-Precision Matrix Multiplication)*

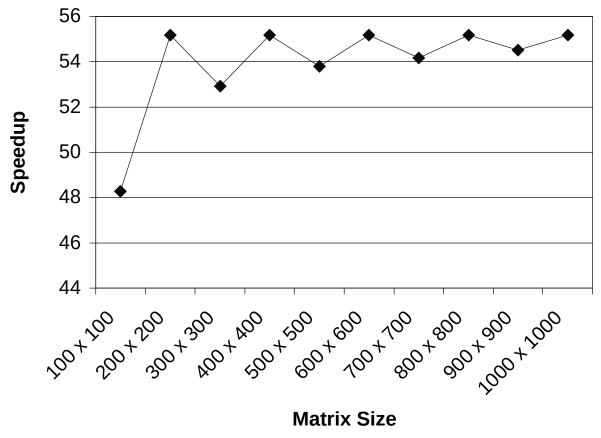

Fig. 6 *HERA speedup of parallel over uni-PE execution*

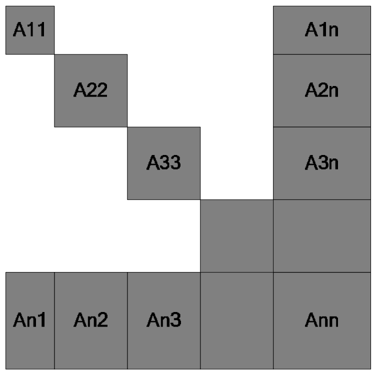

Fig. 7 *Sparse DBBD matrix*

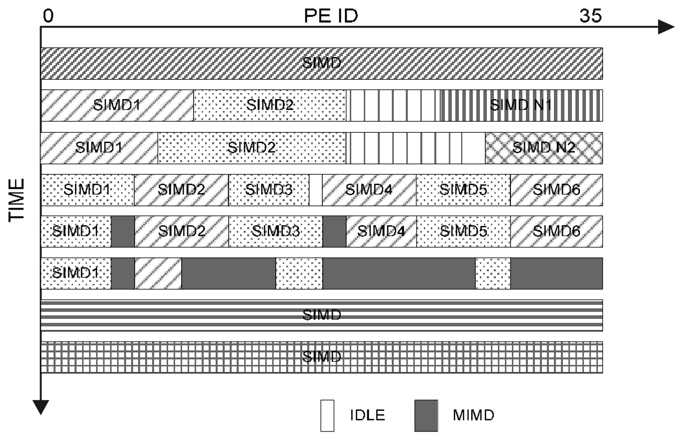

Fig. 8 *Typical PE mode assignment for large DBBD matrices*

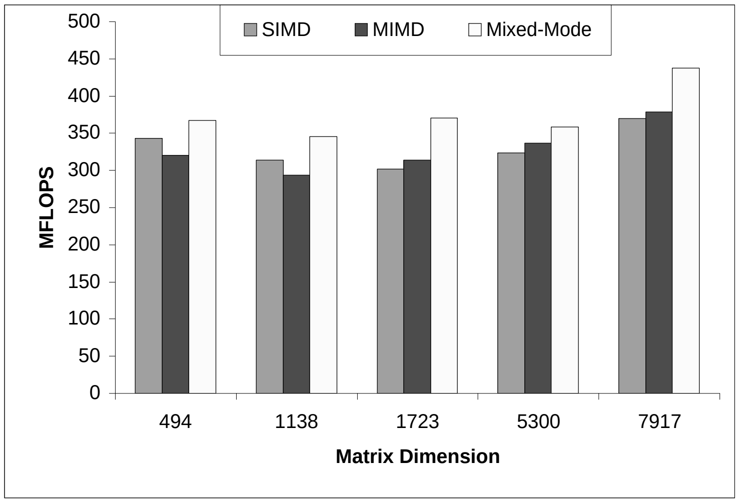

Fig. 9 *Performance of parallel LU factorization on HERA under the SIMD, MIMD and mixed modes (HERA system frequency: 125MHz)*