PAPER *Special Issue on Parallel/Distributed Computing and Networking*

A Coarse-Grain Hierarchical Technique for 2-Dimensional FFT on Configurable Parallel Computers*

ziavras@adm.njit.edu

Xizhen XU† *and* **Sotirios G. ZIAVRAS**†, ***Nonmembers***

SUMMARY FPGAs (Field-Programmable Gate Arrays) have been widely used as coprocessors to boost the performance of data-intensive applications [1][2]. However, there are several challenges to further boost FPGA performance: the communication overhead between the host workstation and the FPGAs can be substantial; large-scale applications cannot fit in a single FPGA because of its limited capacity; mapping an application algorithm to FPGAs still remains a daunting job in configurable system design. To circumvent these problems, we propose in this paper the FPGA-based Hierarchical-SIMD (H-SIMD) machine with its codesign of the Pyramidal Instruction Set Architecture (PISA). PISA comprises high-level instructions implemented as FPGA functions of coarse-grain SIMD (Single-Instruction, Multiple-Data) tasks to facilitate ease of program development, code portability across different H-SIMD implementations and high performance. We assume a multi-FPGA board where each FPGA is configured as a separate SIMD machine. Multiple FPGA chips can work in unison at a higher SIMD level, if needed, controlled by the host. Additionally, by using a memory switching scheme and the high-level PISA to partition applications into coarse-grain tasks, host-FPGA communication overheads can be hidden. We enlist the two-dimensional Fast Fourier Transform (2D FFT) to test the effectiveness of H-SIMD. The test results show sustained high performance for this problem. The H-SIMD machine even outperforms a Xeon processor for this problem.

key words: *Configurable Computing, FPGA, SIMD, Parallel Processing, Memory Switching, FFT, Hardware-Software Codesign.*

1. Introduction

FPGAs have emerged as a new powerful computing paradigm deemed appropriate for datastream processing. They are usually customized to implement computationally costly datapaths when coupled with a host. The host may be a microprocessor or a workstation in charge of program scheduling, dataflow control and data partitioning. There are several ways to connect a host to FPGAs. First, like in the Altera Nios SOPC [3] embedded system, the host is a microprocessor embedded into the FPGA(s) and the rest of the FPGA resources can be used to implement custom instructions as an extension to the microprocessor's ISA. Second, the host may be a workstation while the FPGA resources form a coprocessor that communicates with the host via an I/O interface [5][6]. Each connection mechanism has its own objectives, pros and cons. The microprocessor-FPGA architecture can take advantage of a friendly programming environment and more efficient data exchanges. However, frequent host interventions are needed and the logic capacity of an FPGA chip can be easily consumed up. The workstation-FPGA architecture yields greater parallelism but at the expense of higher communication overheads and a daunting algorithm mapping process. Our H-SIMD design tries to alleviate the drawbacks of the latter mechanism by applying a coarse-grain hierarchical technique to facilitate ease of program development and to overlap as much as possible communications with computations. This is possible since the FPGA board is customized to the application by macro instructions issued by the host machine.

The current trend in high-performance computing emphasizes low cost in the form of PC clusters [15]. Targeting low cost systems of small volume, the implementation of parallel computer architectures on FPGAs has recently become an interesting research topic [14][21]. Many research projects have studied design environments for configurable systems, such as the SPLASH-2 [8], PipeRech [9] and Garp [10] compilers. They can be used to port generic applications to various FPGA platforms. Thus, it is difficult to sufficiently explore data parallelism inherent in application algorithms. [18] employs an FPGA resource manager to allocate and deallocate FPGA resources for concurrent applications. [13] proposes a three-step approach to map generalized template matching applications onto reconfigurable computers. [13][18] implement their designs as software running on the host to specifically explore data parallelism in applications.

The SIMD mode of computation demonstrates high data parallelism. For a multi-FPGA target system, we configure in our work each FPGA chip as a separate SIMD architecture. Multiple FPGA chips can be synchronized to work in SIMD at a higher level controlled by the host. Our approach facilitates ease of program coding since programs are developed in the form of function calls for which the code implementation already resides in the FPGAs. The host sends function IDs as well as corresponding parameter values, if needed, to the FPGAs in a manner that attempts to balance the workload across FPGAs. Based on PISA,

Manuscript received March 22, 2005.

Manuscript revised August 7, 2005.

†The authors are with the ECE Dept., New Jersey Institute of Technology, Newark, NJ 07102, USA

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

application tasks are partitioned into the host, FPGA and nano-processor layers, in decreasing order of task granularity. The host layer, at the top of the H-SIMD machine, is implemented in the workstation by a high-level language. The program developed for the host could be easily ported to different FPGA platforms if the entire software hierarchy was implemented on the FPGA platform and the H-SIMD configuration was embedded in them. This portability resembles that of the JAVA virtual machine [19]. More specifically, similar to the approach for PC clusters in [12][15][20] we suggest that an effective ISA be developed at the host layer for each application domain. Frequently used instructions in that domain should belong to this ISA. To this extent, the hardware design is customized to the application. We have created libraries of effective ISAs for matrix multiplication and discrete cosine transformation [26][27]. The invocation of these instructions is actually implemented as function calls at the host layer. These functions are to be coded efficiently for individual FPGAs. Due to different communication bandwidths across the host-FPGA system, PISA is designed in a workload-oriented manner that attempts to balance the pipeline flow crossing the three layers. Based on the PISA implementation, the workload becomes finer as we go from the host to the nano-processor layer. Another major advantage of the H-SIMD machine is the employment of a memory switching scheme for data loads/stores involving the host and the FPGAs. The switching between pairs of data memory banks overlaps operand communications with computations, thus hiding the communication overheads to improve performance.

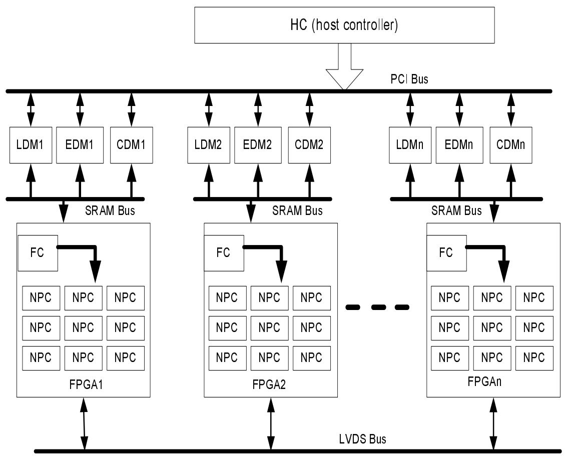


Fig. 1 H-SIMD machine architecture.

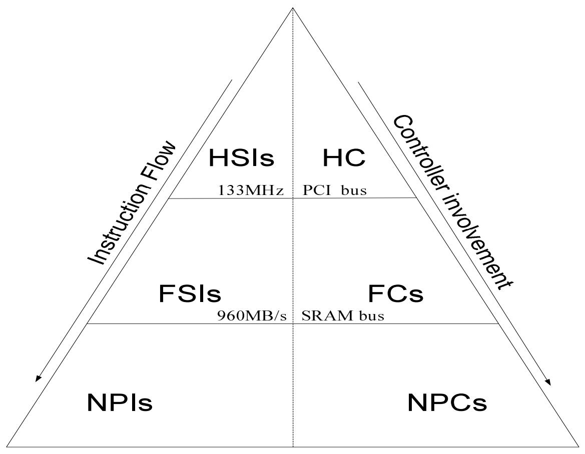


Fig. 2 The PISA ISA for H-SIMD (current implementation).

2. Multi-Layered H-SIMD Machine

2.1 H-SIMD Architecture

The H-SIMD control hierarchy is composed of three layers, as shown in Fig. 1. It comprises the host controller (HC), the FPGA controllers (FCs), and the nano-processor controllers (NPCs). HC lies in the host machine and controls all the FPGA chips in the SIMD mode. Inside each FPGA, an FC is designed to run all the on-chip NPCs in the SIMD mode as well. The NPCs control the execution of nano-processor-level code. PISA facilitates the logical interconnection of the three layers by allowing to logically partition the application into the HC, FC, and NPC layers. These layers can be handled efficiently at run time to balance the pipeline flow from the host down to the on-chip NPCs. Task scheduling and the coarse-grain dataflow control of applications are left to the HC. As shown in Fig. 2, the FCs convert application program functions represented by coarse-grain host SIMD instructions (HSIs) into sequences of medium-grain FPGA SIMD instructions (FSIs). Then, the NPCs execute sequences of machine-level nano-processor instructions (NPIs) for final code execution. Additionally, the HC, the FCs, and the NPCs run at different frequencies. HC is the slowest component while the NPCs are the fastest. It should be noted here that the term nano-processor in the context of reconfigurable systems was coined in [2]. The H-SIMD machine configures one of the FPGA chips as the master FPGA which is responsible for sending an interrupt signal back to the HC once the previously executed HSI has been completed at the nano-processor level. Similarly, one nano-processor within each FPGA is configured as the master nano-processor that sends an interrupt signal back to its FC such that a new FSI can be initiated. The combination of HC and the FCs, however, is different from the conventional SIMD controller scheme because the former is based on decentralized control of the H-SIMD machine while the latter depends on centralized control.

On the programming side, PISA plays a critical role in the H-SIMD machine for bridging the performance gap between sequentially issued HSIs and their parallel execution on the FPGAs. The pyramidal nature of PISA with its three layers implies that larger numbers of instructions can be executed by lower layers in any given amount of time. The HSIs are task-level host instructions issued by HC and decoded by the FCs. The host should be a PC or workstation good at high-level languages. At the second layer, the FSIs sequenced by the FCs aim at function-level processing. The FCs further convert the FSIs into blocks of NPIs to be executed by the nano-processors in the SIMD mode. In this way, the task-level HSIs are executed in parallel by the FPGA-level SIMD machine and the applications fully exploit the FPGA parallel datapath resources. The PISA instruction set is tailored to the specific application in an effort to yield high performance. The developed PISA ISA for 2D FFT is presented in Section 3.

2.2 Memory Switching Schemes

The communication overhead between the host and the FPGA chips is very high primarily due to the non-preemptive nature of the host's operating system. Based on tests in our laboratory, the one-time interrupt latency is about 3.5 ms for a Windows-XP installed Dell Precision 650 workstation with the 133MHz PCI bus. This penalty is intolerable in high-performance computing because, for example, the 1024-point IEEE754-based complex FFT takes about 640 us on a single floating-point unit running at 80 MHz (which is within the range of the current FPGA technology). If the host frequently intervenes in FPGA operations, the speedup benefits gained from the parallel FPGA implementation can be significantly reduced or even removed. Yet, given the system configuration of the host machine, the latency is fixed and cannot be reduced unless a preemptive operating system or special hardware interface is enlisted. In order to overcome the interrupt latency problem, a data prefetching scheme involving memory switching is designed for H-SIMD to overlap host communications with FPGA computations. We make sure that the computation time on the HSIs is longer than the total host communication time, which is the sum of the data reference and host interrupt response times.

Data flowing from HC is directed into the high-speed SRAM banks on the FPGA board. The HC-level memory switching scheme is shown in Fig. 3. The SRAM banks are organized into three functional memory units: the execution data memory (EDM), the loaded data memory (LDM) and the communication data memory (CDM). The EDMs and LDMs are functionally interchangeable. They alternate between HSI execution and HSI data load/retrieve. At one time, EDM is connected to the FCs to supply their operands. On the contrary, LDM is connected to the host, and is ready to receive the next HSI and its new operands. When the FCs finish their current HSI, they will switch

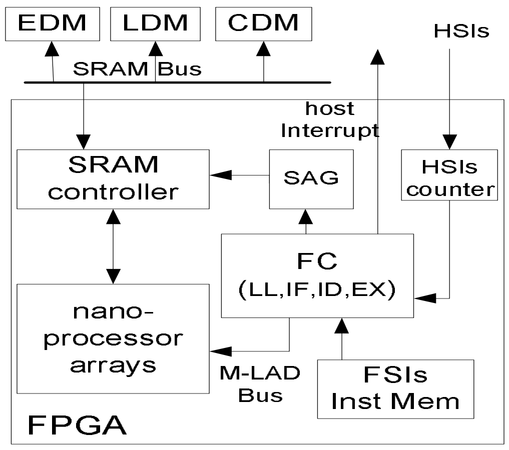


Fig. 3 HC-level memory switching in H-SIMD.

EDM and LDM to begin a new iteration. An FC is a finite-state machine responsible for the execution of a computation HSI. The FCs have access to the NP array over a modified LAD (M-LAD) bus. The LAD bus was originally developed by Annapolis Micro Systems and used for on-chip access [4]. The M-LAD bus controller is changed from the PCI controller to the FCs. The HSI counter is used to count the elapsed cycles for computation HSIs. The SRAM address generator (SAG) is used to calculate the SRAM load/store addresses for the EDM banks. CDM stores the nano-processor computation results which can be transmitted via the 8Gbytes/s LVDS (low-voltage differential signaling) connection to other FPGA chips. Each SRAM bank has its own 960Mbytes/s bus and the aggregate SRAM bandwidth can sustain high-performance applications. LVDS can transfer large amounts of data between chips while keeping low the power consumption and wire count. It employs a differential mode of transmission where every channel uses a pair of lines.

The nano-processors form the execution units in the H-SIMD datapath. Their functionality can be customized according to the application. In our implementation, the communications among the nano-processors are based on a nearest neighbors NEWS (North-East-West-South) grid interconnection. Data needed by a nano-processor can be routed from its north/east/west/south neighbor by using an NR/ER/WR/SR NPI. The nano-processors reside at the lowest layer of the H-SIMD machine hierarchy. Each nano-processor consists of two parts: a fine-grain custom datapath and large-sized register files. The former can be suitable for FFT, matrix multiplication (MM) or any other customized NPIs. The latter includes the load register file (LRF) and the execution register file (ERF) shown in Fig. 4. Both register files work in a "memory" switching capacity, similarly to the LDMs and EDMs. They alternate the execution of FSIs with data loading from the FCs. After the

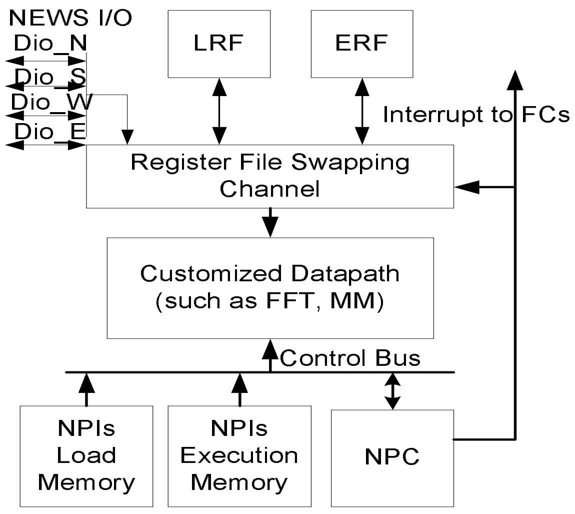


Fig. 4 Nano-processor datapath and control unit.

nano-processor finishes data processing with ERF, its datapath will configure another LRF as an ERF and will then begin a new program flow. At the same time, the just switched-out ERFs will be configured as LRFs to be loaded with new operands from the EDMs controlled by the FCs. These operands will then be ready to be processed. The interrupt request/response latency between the FCs and their own nano-processors is one cycle only as opposed to the tens of thousands of cycles between the host and the FPGAs, thus gaining a performance boost.

3. PISA Design and Task Partitioning for 2D FFT

3.1 PISA ISA for 2D FFT

FFT is an efficient algorithm to compute the discrete Fourier transform (DFT). FFT can reduce the computation complexity of the N-point DFT from $O(N^2)$ to $O(NlogN)$. For a discrete-time sequence x(n), for n=0, , N-1, its 1D DFT is defined as [7]: $X(k) = \sum_{n=0}^{N-1} x(n)e^{-jnk2\pi/N} \quad k = 0, ..., N-1$, where the twiddle factors $e^{-jnk2\pi/N}$ can be pre-computed and stored in the on-chip memory of the FPGA. The 2D FFT can be carried out by applying 1D FFT in parallel to all rows and then in parallel to all columns, or vice versa. We exemplify here the tailoring of PISA for the H-SIMD design methodology with 2D FFT. As already mentioned, the H-SIMD machine can fit a wide variety of applications characterized by data parallelism. Three HSIs are needed for 2D FFT and sit at the topmost layer. Their object methods are: 1) $host\_fft2\_load(LDM_i, N_h)$: This HSI will load each FPGA's LDM_i memory sequentially via the host PCI bus (host-based DMA control is used); 2) $host\_fft2(H_A, H_B, N_h)$: H_A is the input matrix of size N_hxN_h and $H_B = fft2(H_A)$; 3) $host\_fft2\_retrieve(LDM_i, N_h)$: The final results are stored in the switched-out LDM_i and can be retrieved by the host sequentially via DMA through the PCI bus when the last issued HSI is done.

The FSIs in the middle of the PISA hierarchy run on the FPGA hardware. Assume that there are q FPGAs with p nano-processors each. There are four FSIs: 1) $FPGA\_fft\_load(F_a, LRF_i, N_h)$: There are two register files in nano-processor i, LRF_i and ERF_i. FC will execute this instruction by loading each nano-processor's LRF with a row vector of size N_h from matrix F_a of size N_h/q x N_h. F_a is the starting addresses of the input; 2) $FPGA\_fft(F_a, F_b, N_h)$: matrix F_b of size N_h/q x N_h is produced by applying 1D FFT to each row of the input matrix F_a; 3) $FPGA\_fft\_transpose(F_a, F_b, N_h)$: matrix F_a of size N_h/q x N_h is transposed and stored into the FPGA's CDM via the high-speed SRAM interface with source starting address F_a and destination starting address F_b; 4) $FPGA\_lvds\_comm(F_a, N_h)$: matrix F_a of size N_h/q x N_h within one FPGA is transmitted by a LVDS connection to neighbor FPGAs.

The NPIs constitute three instructions:$NP\_load(R_a, N_h)$, $NP\_retrieve(R_a, N_h)$ and 1D N_h-point FFT, $NP\_fft(R_a, N_h)$. The last instruction is produced by the Xilinx CoreGenerator [22] or Annapolis CoreFire libraries [11]. The customized FFT nano-processor is shown in Fig. 5. The input data in our experiments is a vector R_a of N_h = 64, 256 or 1024-point values represented as 16-bit complex or IEEE754 single-precision floating-point numbers. R_a is the starting address of the vector residing in the ERF memory of each nano-processor.

The input matrices for 2D FFT have size N_hxN_h, where N_h= 1024 in the current implementation due to the recent CoreFire6.2.03 release and the capacity constraints of the on-board SRAMs. Each FPGA is assigned N_h/q vectors of size N_h, where q is the number of FPGAs in the H-SIMD machine. For each FPGA chip, the FC issues $FPGA\_fft(F_a, F_b, N_h)$ to carry out N_h/q N_h-point 1D FFTs which can be run on the nano-processor arrays in parallel. After finishing the FFTs of the first dimension vectors, the FCs execute $FPGA\_fft\_transpose(F_a, F_b, N_h)$ to transpose the results and write back into the CDMs. Then, $FPGA\_lvds\_comm(F_b, N_h)$ is invoked to broadcast the results from CDM to the neighbor FPGAs' EDMs in the ring network. The inter-FPGA communication cost is negligible due to the high-speed 8Gbytes/s LVDS connections between FPGAs. After that, FC issues another $FPGA\_fft(F_a, F_b, N_h)$ to transform the second dimension vectors. The FCs also respond to nano-processor interrupt requests by first initiating a new FFT execution for the nano-processors and then loading new data into the LRFs. All these steps are done

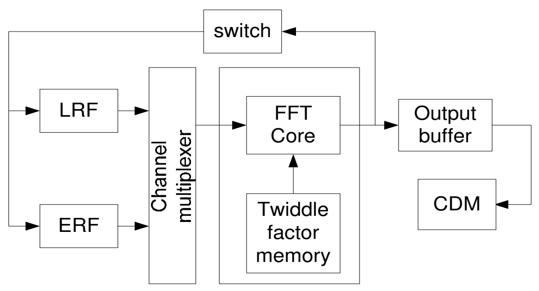


Fig. 5 Nano-processor FFT datapath.

within the FPGAs without host intervention.

3.2 Analysis of Task Partitioning in 2D FFT

The bandwidth of the communication channels in the H-SIMD machine varies greatly. Basically, there are three interfaces in H-SIMD: the PCI bus with a bandwidth B_{pci} of 133MHz x 64 bits, the SRAM bus with a bandwidth B_{sram} of 960Mbytes/s, and the LVDS inter-FPGA connections with a bandwidth B_{lvds} of 8Gbytes/s. In the case of 2D FFT on an N_hxN_h matrix, let us assume that N_h/q x N_h sub-matrices of an N_hxN_h matrix are uniformly distributed among the q FPGAs, with p nano-processors each. The $host\_fft2(H_A, H_B, N_h)$ HSI consists of three operations on the input matrix H_A, i.e., the F_a sub-matrix of size N_h/q x N_h from matrix H_A will go through 1D FFTs on its rows and transposed columns, sub-matrix transpose and inter-FPGA communications. We denote the execution time of the three operations as T_{1D_fft}, T_{trans}, and $T_{fpga-fpga}$, respectively. The total computation time for $host\_fft2(H_A, H_B, N_h)$ is:
$T_{compute}(host\_fft2) = 2*N_h/(q*p)*T_{1D\_fft}+T_{trans}+T_{fpga-fpga}+T_{initialize\_fft}$.
On the other hand, the communication time T_{i/o_pci} of $host\_fft2\_load/retrieve$ depends on the available PCI I/O bandwidth and the interrupt latency T_{host_int}:
$T_{i/o\_pci} = 2*b*N_h^2/B_{pci}+T_{host\_int}$,
where b is the width in bits of each transaction. Pipeline balancing is guaranteed if $T_{compute}$ is greater than or equal to T_{i/o_pci}, and the computations fully overlap I/O communications. Specifically, the following approximations should hold for 2D FFT:
$T_{initialize\_fft} > 3*N_h*t$,
$T_{1D\_fft} > N_h*t$,
$T_{trans} > 2*N_h*N_h*t/q$,
$T_{fpga-fpga} > N_h*N_h*(q-1)*b/(q*B_{lvds})$,
Thus,
$T_{i/o\_pci} = 2*b*N_h^2/B_{pci}+T_{host\_int}$,
$T_{compute}(host\_fft2) > 2*N_h*T_{1D\_fft}/(q*p)+T_{trans}+T_{fpga-fpga}+T_{initialize\_fft} = (2*N_h*N_h/(p*q)+2N_h*N_h/q+3*N_h)*t+N_h*N_h*(q-1)*b/(q*B_{lvds})$ where t is the nano-processor cycle time. For the configuration p=6, B_{pci}=133MHz x 64bits, $T_{host\_int} = 3.5ms$, b=32 and t=11ns on 16-bit complex numbers, the simulation results in Fig. 6a show that the computation time varies with the size N_h of the matrix and the number q of FPGAs. The communication time is independent of the number of FPGAs because the data traffic on the PCI bus, given a problem size N_h, is fixed and the input vectors from the host are uniformly distributed among all the FPGAs. With increases in the matrix size for 2D FFT, the computation time grows faster than the I/O communication time, which is exploited by H-SIMD to implement host-level memory switching (the FPGAs waste no time to wait for data loads/retrievals). This condition is easier met for applications with high computation load (e.g., floating-point matrix multiplication) rather than with low computation load (e.g., integer FFT). In fact, a full overlap is achieved when the input matrix has size greater than 896 and there are two FPGAs. If more FPGAs are enlisted, H-SIMD is difficult to fully overlap communications with computations for 2D FFT on 16-bit complex numbers.

At the FC level, $FPGA\_fft(F_a, F_b, N_h)$ is carried out on all the nano-processors while vectors of size N_h are loaded into LRF at the same time. For effective nano-processor-level memory switching, the execution time T_{NPI_fft} of the 1D N_h-point FFT $NPI\_fft(R_a, N_h)$ must be greater than p times the register file reference time T_{NP_load}, i.e., $T_{NPI\_fft} > p*T_{NP\_load}$, where $T_{NPI\_fft} = N_h*t$ and $T_{NP\_load} = N_h/(B_{sram}*N_{bank})$. N_{bank} is the number of the SRAM banks available to each FPGA. In fact, this condition can be easily met when B_{sram}=960Mbytes/s, p=6, t=11ns and $N_{bank}=6$, as shown in Fig. 6b. If computations do not overlap fully communications at the FC level, more SRAM banks can be employed to provide higher aggregate bandwidth.

4. Implementation and Test Results for 2D FFT

4.1 H-SIMD Machine Programming

We implemented the H-SIMD machine on a host PC workstation and an Annapolis FPGA board [4] containing two Xilinx Virtex-II 6000 FPGAs. There are six Samsung 512k x 36bits SRAM banks and one Micron 16M x 32bits DRAM bank around each FPGA. The host is given a program of 2D transformation $H_{fft2}(N_h, N_h) = fft2(H_A)$, where H_A and H_{fft2} are matrices of size N_hxN_h. N_h/q vectors of size N_h are assigned to each FPGA for 1D N_h-point FFT, where N_h is the size of the HC-level matrix. The host pseudo-program with its HSIs is:

```
$\boldsymbol{host\_fft2\_load(EDM_m, N_h)}$;/*load EDMs*/
do in parallel for all the FPGAs{
$\boldsymbol{host\_fft2\_load(LDM_m, N_h)}$; /*m=0,..., q-1*/
$\boldsymbol{host\_fft2(H_A, H_B, N_h)}$;
$\boldsymbol{wait\_for\_Interrupt(masterFPGA)}$;
```

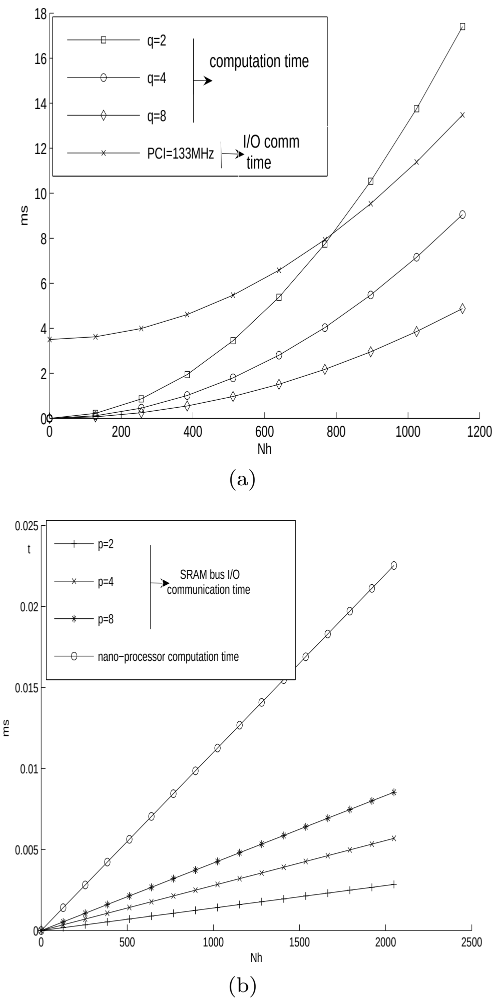


Fig. 6 Computation and I/O communication times with (a) host PCI bandwidth and (b) SRAM bandwidth.

```
$host\_fft2\_retrieve(LDM_m, N_h);$}
```

On the other hand, the FCs execute each computation HSI by first decoding it and then initiating a sequence of nano-processor operations. The program flow is run by the FCs. The "for loop" is implemented by the SRAM Address Generator (SAG) and the FSI instruction queues. The FC-layer pseudo-code goes as follows:

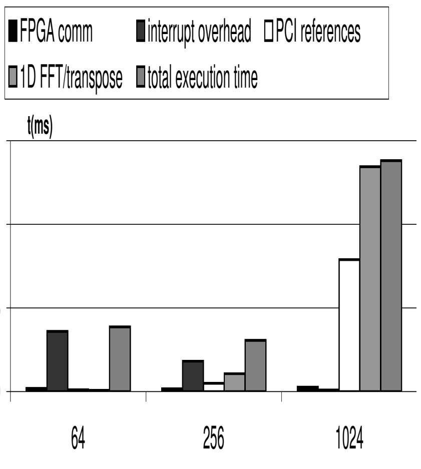
```
$for(m = 0; m < p; m++)$
${FPGA\_fft\_load(F_a, LRF_m, N_h);}$
do in parallel for all the nano_processors{
$FPGA\_fft\_load(F_a, LRF_m, N_h);$/*m=0,..., p-1*/
$FPGA\_fft(F_a, F_b, N_h);$
$WaitForInterrupt$(masterNP);
}
$FPGA\_fft\_transpose(F_a, F_b, N_h);$
$FPGA\_fft\_comm(F_b, N_h);$
do in parallel for all the nano_processors{
$FPGA\_fft\_load(F_a, LRF_m, N_h);$/*m=0, ..., p-1*/
$FPGA\_fft(F_a, F_b, N_h);$
$WaitForInterrupt$(masterNP); }
```

4.2 Implementation Results

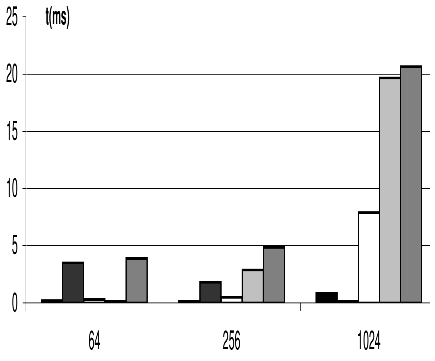
We implemented H-SIMD on the Annapolis board with two kinds of nano-processors for 16-bit complex numbers and IEEE754 single-precision floating-point numbers, respectively. The two 1D FFT cores were generated by Xilinx CoreGenerator6.2 [22] and Annapolis CoreFire3.8 [11]. The CoreFire Design Suite3.8 is a dataflow-based application package that provides end-users with a large collection of cores, including host access cores. We implemented a 32-bit counter in the FPGAs to count the elapsed cycles between the initiation of the first HSI and the completion of the last HSI. After Xilinx ISE6.2 Place& Route, six 89MHz nano-processors can fit in one FPGA for 16-bit complex numbers while only one 80MHz nano-processor for single-precision floating-point numbers. 2D FFTs on 64x64, 256x256 and 1024x1024 matrices were tested. The results were compared with results produced by Matlab. Our H-SIMD machine has precision 10^{-5} if the Matlab results are assumed as the benchmark. The timing results break down into inter-FPGA communication, interrupt overhead, PCI reference time and 1D FFT/transpose computation time, as shown in Fig. 7. The performance depends heavily on the interrupt overhead, 1D FFT/transpose computation time and host-PCI reference time. All the other factors contribute little to the total execution time, which is desirable for the H-SIMD design. When N_h is set to 64, the frequent interrupt requests to the host contribute excessively to the performance penalty. When N_h is set to 256, the computation time does not increase long enough to overlap fully the sum of the costs for host interrupts and PCI-SRAM memory sequential references. If N_h is set to 1024, the designed overlap is so good that the interrupt and communication overheads are hidden, and all the nano-processors work in parallel.

We count only arithmetic operations for the input 16-bit complex numbers and use the million-operations-per-second (MOPS) metric for the purpose of benchmarking. The operation count is consistent with the one in [24]. For complex-data and real-data FFTs, the number of operations is $5*N*log_2N$ and $2.5*N*log_2N$, respectively. Based on the same 2D FFT problem, we compare in Table 1 the H-SIMD MOPS for 16-bit complex numbers and its MFLOPS for IEEE754 single-precision numbers with the performance of a 2.8GHz Xeon processor as presented in [17]. The H-SIMD machine suffers a great deal of performance loss due to frequent host interventions when the application does not have enough parallelism to exploit. However, it can outperform the powerful Xeon processor when applications show enough parallelism. This proves that H-SIMD fits well the data-intensive applications.

A cost-performance analysis of the H-SIMD machine and a Xeon processor is in order now. The ten



(a)



(b)

Fig. 7 Execution time breakdown of 2D FFT on (a) 16-bit complex numbers and (b) IEEE754 single-precision floating-point numbers.

Table 1 Performance comparison of the H-SIMD machine and a 2.8GHz Xeon workstation for 2D FFT.

Matrix size	H-SIMD on 16-bit complex numbers (MOPS)	H-SIMD on IEEE754 single precision real numbers (MFLOPS)	2.8 GHz Xeon on IEEE754 single precisions (MFLOPS)
64	68.8	33	2700
256	1796	557	3100
1024	7643	2630	2300

million system gates in the Virtex II FPGA consume about 400 million transistors [22]. The H-SIMD machine built on the Annapolis board contains two Virtex II FPGAs. Our current implementation employs roughly 700 million transistors. On the other hand, a 2.8GHz Xeon processor contains about 286 million transistors [23]. For 1024x1024 FFT on IEEE-754 single-precision numbers, it takes 23 ms on a Xeon processor as opposed to 20 ms on the H-SIMD machine. According to a widely used VLSI complexity model, the cost C of implementing an algorithm is defined as $C = A * T^2$, where A is the chip area and T is the

Table 2 Cost-performance comparison of the H-SIMD machine and the Xeon processor.

System	Transistors (millions)	Execution Time(ms)	VLSI Cost (normalized)	Speedup (normalized)
2.8GHz Xeon	286	23	1	1
H-SIMD (Virtex-II)	700	20	1.85	1.15

execution time. The chip area is directly proportional to the number of transistors, so we substitute in the cost equation the latter for the former. The VLSI cost and speedup results in Table 2 are normalized with respect to the Xeon processor. The H-SIMD machine provides a speedup of 15% but its VLSI cost increase is 85%. This is due to the slow FPGA clock frequency (around 90MHz) of the implementation on Virtex II 6000. [16] reported 210MHz floating-point operations on Virtex2Pro. For recent FPGAs, like Virtex 4 and Stratix II, both Xilinx and Altera claim clock frequencies of 500MHz [3][25]. The slogan "silicon is free" is widely acceptable for system-on-chip designs. However, unlike the gigahertz microprocessors with power problems, state-of-the-art FPGAs still have much room to increase their clock speed. We should expect to see a dramatic cost reduction in the near future with steady advancements in FPGA technologies.

5. Conclusions

Our multi-layered H-SIMD machine paired with an appropriate multi-layered PISA software codesign is effective in data-parallel applications. To yield high performance, task partitioning is carried out at different granularity levels, involving the host, the FPGA controllers and the nano-processors inside the FPGAs. The H-SIMD's memory switching scheme is able to fully overlap I/O communications with FPGA computations to bridge the gap between sustained and peak performance. In our current implementation of 2D FFT, we have demonstrated the effectiveness of the H-SIMD machine on applications with enough parallelism. More recent FPGAs could improve performance even further. Our multi-layered design approach is not limited to 2D FFT. It can be applied to other applications as well which are rich in data parallelism [26][27]. In the future, we will continue efforts to extend current PISAs for more data-intensive applications.

References

[1] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: a survey", Journal VLSI Signal Processing, Vol. 28, pp. 7-27, 2001.

[2] M.J. Wirthlin, B.L. Hutchings and K.L. Gilson, "The nano processor: a low resource reconfigurable processor", IEEE Workshop on FPGAs for Custom Computing Machines, pp.23-30, April 1994.

[3] Altera, Inc., http://www.altera.com.

[4] Wildstar II hardware reference manual, Annapolis Microsystems, Inc, Annapolis, MD, 2004.

[5] R. Laufer, R.R. Taylor and H. Schmit, "PCI-PipeRench and the swordAPI: a system for stream-based reconfigurable computing", IEEE Symp. Field-Programmable Custom Comput. pp. 200-208, March. 1999.

[6] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati and P. Boucard, "Programmable active memories: reconfigurable systems come of age", IEEE Trans. VLSI Syst. Vol. 4, No.1, pp. 56-69, 1996.

[7] L. R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1975.

[8] D. Buell, W.J. Kleinfelder and J.M. Arnold, "Splash 2: FPGA's in a custom computing machine", IEEE Computer Society Press, 1996.

[9] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe and R.R. Taylor, "PipeRench: a reconfigurable architecture and computer", IEEE Computer, Vol. 33, No. 4, pp. 70-77, April 2000.

[10] T.J. Callahan, J.R. Hauser and J. Wawrzynek, "The Garp architecture and C compiler", IEEE Computer, Vol. 33, No. 4, pp. 62-69, April 2000.

[11] CoreFire Design Suite, Annapolis Microsystems, Inc, Annapolis, MD, 2004.

[12] D. Jin and S. G. Ziavras, "A super-programming technique for large sparse matrix multiplication on PC clusters", IEICE Trans. Information and Systems, Vol. E87-D, No. 7, pp. 1774-1781, July 2004.

[13] X. Liang and J.S.N. Jean, "Mapping of generalized template matching onto reconfigurable computers", IEEE Trans. VLSI Systems, Vol. 11, No. 3, pp. 485-498, June 2003.

[14] X. Wang and S.G. Ziavras, "Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines", Concurrency and Computation: Practice and Experience, Vol. 16, No. 4, pp. 319-343, April 2004.

[15] D. Jin and S. Ziavras, "A super-programming approach for mining association rules in parallel on PC clusters", IEEE Trans. Parallel and Distributed Systems, Vol. 15, No. 9, pp. 783-794, Sept. 2004.

[16] Y. Dou, S. Vassiliadis, G. K. Kuzmanov and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," ACM/SIGDA 13th Intern. Symposium on FPGAs, Feb. 2005.

[17] http://www.fftw.org.

[18] J.S.N. Jean, K. Tomko, V. Yavagal, J Shah and R. Cook, "Dynamic reconfiguration to support concurrent applications", IEEE Trans. Computers, Vol. 48, No. 6, pp. 591-602, June 1999.

[19] http://java.sun.com.

[20] D. Jin and S. G. Ziavras, "Modeling distributed data representation and its effect on parallel data accesses", Journal of Parallel and Distributed Computing, Vol. 65, No. 10, pp. 1281-1289, Oct. 2005.

[21] S.G. Ziavras, "Processor design based on dataflow concurrency", Microprocessors and Microsystems, Vol. 27, No. 4, pp. 199-220, May 2003.

[22] http://www.xilinx.com/ipcenter.

[23] http://www.intel.com/pressroom/kits/quickreffam.htm#Xeon.

[24] M. Frigo and S.G. Johnson, "The design and implementation of FFTW3," Proc. IEEE, vol 93, no 2, pp. 216-231, 2005.

[25] Xilinx, Inc., http://www.xilinx.com.

[26] X. Xu and S.G. Ziavras, "A hierarchically-controlled SIMD machine for 2D DCT on FPGAs," IEEE Intern. Systems-on-Chip Conf., Sept. 2005.

[27] X. Xu and S.G. Ziavras, "H-SIMD machine: configurable parallel computing for matrix multiplication," IEEE Intern. Conference on Computer Design, Oct. 2005.

Xizhen Xu received his B.S. and M.S. in electrical engineering from the Northwestern Polytechnical University (China) in 1993 and 1996, respectively. Currently, he is a Ph.D. candidate of electrical and computer engineering at New Jersey Institute of Technology (NJIT). His research interests include the design and application of reconfigurable computing systems, parallel computing architecture, and VLSI design.

Sotirios G. Ziavras received the Diploma in EE from the National Technical University of Athens, Greece, in 1984, the M.Sc. in Computer Engineering from Ohio University in 1985, and the Ph.D. degree in Computer Science from George Washington University in 1990. He was a Distinguished Graduate Teaching Assistant at GWU. He was also with the Center for Automation Research at the University of Maryland, College Park, from 1988 to 1989. He was a visiting Assistant Professor at George Mason University in Spring 1990. He joined in Fall 1990 the ECE Department at NJIT as an Assistant Professor. He was promoted to Associate Professor and then to Professor in 1995 and 2001, respectively. He is currently the Associate Chair for Graduate Studies in the ECE Department at NJIT. He also has a joint appointment in the Computer Science Department. He is an Associate Editor of the Pattern Recognition journal. He is an author/co-author of about 100 research and technical papers. He is listed, among others, in Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, and Who's Who in the East. His main research interests are conventional and unconventional processor designs, field-programmable gate arrays, embedded computing systems, parallel and distributed computer architectures and algorithms, network router design, and computer architecture design.