A High Throughput FPGA-Based Implementation of the Lanczos Method for the Symmetric Extremal Eigenvalue Problem

Abid Rafique, Nachiket Kapre, and George A. Constantinides

Electrical and Electronic Engineering,
Imperial College London,
London SW7 2BT, UK
{a.rafique09,n.kapre,g.constantinides}@ic.ac.uk

Abstract. Iterative numerical algorithms with high memory bandwidth requirements but medium-size data sets (matrix size $\sim$ a few 100s) are highly appropriate for FPGA acceleration. This paper presents a streaming architecture comprising floating-point operators coupled with high-bandwidth on-chip memories for the Lanczos method, an iterative algorithm for symmetric eigenvalues computation. We show the Lanczos method can be specialized only for extremal eigenvalues computation and present an architecture which can achieve a sustained single precision floating-point performance of 175 GFLOPs on Virtex6-SX475T for a dense matrix of size 335×335. We perform a quantitative comparison with the parallel implementations of the Lanczos method using optimized Intel MKL and CUBLAS libraries for multi-core and GPU respectively. We find that for a range of matrices the FPGA implementation outperforms both multi-core and GPU; a speed up of 8.2-27.3× (13.4× geo. mean) over an Intel Xeon X5650 and 26.2-116× (52.8× geo. mean) over an Nvidia C2050 when FPGA is solving a single eigenvalue problem whereas a speed up of 41-520× (103× geo.mean) and 131-2220× (408× geo.mean) respectively when it is solving multiple eigenvalue problems.

1 Introduction

Recent trends have shown that the peak floating-point performance of the Field Programmable Gate Arrays (FPGAs) will significantly exceed that of the traditional processors [1]. We can accelerate scientific computations like the solution to systems of linear equations [2], [3] and [4] due to ever-increasing capacity of modern FPGAs in terms of floating point units and on-chip memory. The symmetric extremal eigenvalue problem is an important scientific computation involving dense linear algebra where one is interested in finding only the extremal eigenvalues of a $n \times n$ symmetric matrix. Unlike general eigenvalues computation which has a computational complexity of $\Theta(n^3)$, the extremal eigenvalues can be computed with much less computational complexity ($\Theta(n^2)$) [8]. Solving multiple independent extremal eigenvalue problems is common in semidefinite programming (SDP) solvers [6] where one has to find the extremal eigenvalues of more than one symmetric matrix in each iteration and also in real-time eigenvalue based channel sensing for cognitive radios (IEEE 802.22) [7] where one has to sense multiple channels simultaneously. Accelerating symmetric extremal eigenvalues computation is of increasing importance in both cases. While, in the former case, the goal is to reduce the overall runtime, in the latter case it is desired to meet strict timing requirements which are in the range of a few microseconds.

Existing FPGA-based eigensolvers [9], [10] and [11] compute all the eigenvalues of very small matrices by using the direct method of Jacobi which has a computational complexity of $\Theta(n^3)$. We investigate a 2-stage iterative framework comprising the Lanczos method [5] followed by the bisection method [5] which has an overall computational complexity of $\Theta(n^2)$.

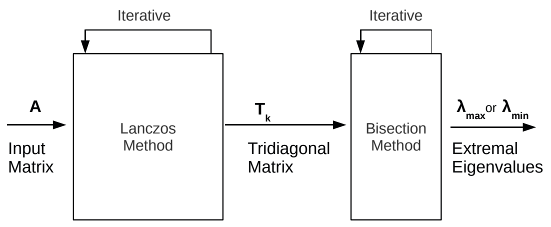


Fig. 1. Symmetric Extremal Eigenvalue Computation

The Lanczos method, like any other iterative numerical algorithm, is dominated by repeated matrix-vector multiplication which can be easily parallelized using the FPGAs. We have noticed that with minimum modifications existing efficient architectures for matrix-vector multiplication in minimum residual (MINRES) method [3] can be reused for the Lanczos method. We present a streaming architecture comprising pipelined single-precision floating-point operators coupled with high-bandwidth on-chip memories leading to an architecture with high throughput but also considerable latency. We exploit pipelining to solve multiple independent extremal eigenvalue problems on the same architecture achieving an efficiency of almost 100% for higher order matrices. The key contributions of this paper are thus:

– A specialized iterative framework for computing only the extremal eigenvalues of a dense symmetric matrix with a computational complexity of $\Theta(n^2)$.
– An architecture modified from [3] for accelerating multiple small to medium size symmetric eigenvalue problems.
– An FPGA implementation of the proposed architecture capable of a sustained performance of 175 GFLOPs on a 260 MHz Virtex6-SX475T for a maximum matrix of size 335 $\times$ 335.
– A quantitative comparison with an Intel Xeon X5650 and an Nvidia C2050 GPU showing a speed up of 8.2-27.3$\times$ (13.4$\times$ geo. mean) and 26.2-116$\times$ (52.8$\times$ geo. mean) respectively when FPGA is solving a single eigenvalue problem whereas a speed up of 41-520$\times$ (103$\times$ geo.mean) and 131-2220$\times$ (408$\times$ geo.mean) when it is solving multiple eigenvalue problems.

2 Background

2.1 Symmetric Extremal Eigenvalue Problem

Symmetric eigenvalue computation is essential in many fields of science and engineering where one has to solve Ax = λx (A is an $n \times n$ symmetric matrix, λ is an eigenvalue and x is the corresponding $n \times 1$ eigenvector). In the particular case of the symmetric extremal eigenvalue problem, we are only interested in finding either $\lambda_{\max}$, $\lambda_{\min}$ or both eigenvalues. There are two main families of methods for solving the extremal eigenvalue problem: direct methods and iterative methods [8]. Direct methods compute eigenvalues in one shot, however, they incur a computation cost of $\Theta(n^3)$ and are more applicable when all the eigenvalues and the corresponding eigenvectors are required. While iterative methods only approximate the eigenvalues, their computational complexity is $\Theta(n^2)$ when only a few eigenvalues are desired and they are thus suitable for extremal eigenvalues computation. A flow is shown in Fig. 1 for the extremal eigenvalues computation comprising iterative Lanczos method followed by the bisection method.

Algorithm 1 . Lanczos Method in Exact Arithmetic [5]

```
Require: Symmetric matrix A$\in \mathbb{R}^{n \times n}$, initial
  orthonormal Lanczos vector $q_0 \in \mathbb{R}^{n \times 1}$ and
  number of iterations $k$, $\beta_0 = 0$.
  for i = 1 to $k$ do

      $\overline{q_i} := Aq_{i-1}$                  ($lz1$)
      $c_i := \overline{q_i} - \beta_{i-1}q_{i-2}$   ($lz2$)
      $\alpha_i := \overline{q_i}^T q_{i-1}$         ($lz3$)
      $d_i := c_i - \alpha_i q_{i-1}$                ($lz4$)
      $b := d_i{}^T d_i$                             ($lz5$)
      $\beta_i := \sqrt{b}$                          ($lz6$)
      $f := 1/\beta_i$                               ($lz7$)
      $q_i := f d_i$                                 ($lz8$)

  end for
  return Tridiagonal matrix $T_k$ containing $\beta_i$
  and $\alpha_i$ i = 1, 2,..., k
```

Algorithm 2. Bisection Method [5]

```
Require: $\alpha_i$ and $\beta_i$ for i = 1,2, .... k
  a:= 0, b:= 0, $eps$ := 5.96x10$^{-8}$
  for i = 1 to k do
      a := max(a , $\alpha_i$ - ( | $\beta_i$| + |$\beta_{i-1}$|))
      b := max(b , $\alpha_i$ + ( | $\beta_i$| + |$\beta_{i-1}$|))
  end for
  i := 1, q:= 1
  while (|b-a| < $eps$(|a| + |b|)) do
      $\lambda := (a + b)/2$
      p := $\alpha_i$ - $\lambda$ - $\beta_i^2$/q
      if (p > 0) then
          a := $\lambda$, i := 1, q := 1
      else if (i >= k) then
          b := $\lambda$, i := 1, q := 1
      else
          q := p, i := i + 1
      end if
  end while
  return $\lambda$
```

The Lanczos Method

The Lanczos method is an iterative Krylov Subspace method [5] which, starting from an initial $n \times 1$ q_0 vector, builds the subspace spanned by K_k(A, q_0) = [q_0, Aq_0, A$^2q_0$, ... , A$^{k-1}q_0$]. It utilizes this subspace to approximate the eigenvectors and the corresponding eigenvalues of A. In order to find all the eigenvalues, A is generally reduced to a $n \times n$ tridiagonal matrix T [5] because there are efficient algorithms available for finding the eigenvalues of T. The main intuition behind the Lanczos method is that it generates a partial tridiagonal matrix T_k (see Algorithm 1) from A where the extremal eigenvalues of T_k are the optimal approximations of the extremal eigenvalues of A [8] from the subspace K_k.

Loss of orthogonality among the vectors is an inherent problem with the Lanczos method and often a costly step of re-orthogonalization is introduced [5]. However, investigation of the loss of orthogonality reveals it does not affect extremal eigenvalues [8] and therefore we do not need to perform re-orthogonalization. This significantly reduces the complexity of the Lanczos method and helps us designing an architecture highly specialized for extremal eigenvalues computation.

Bisection Method

The bisection method is an efficient method for finding the eigenvalues of a symmetric tridiagonal matrix T_k and it has a computational complexity of $\Theta(k)$ for extremal eigenvalue computation [8]. An extremal eigenvalue of the matrix T_k is computed by finding an extremal root of the polynomial.

$$p_k(\lambda) = \det(\mathrm{T}_k - \lambda \mathrm{I}). \tag{1}$$

The extremal root can be computed recursively from the roots of the polynomials $p_r(\lambda)$ where $0 \leq r \leq k$ (see Algorithm 2).

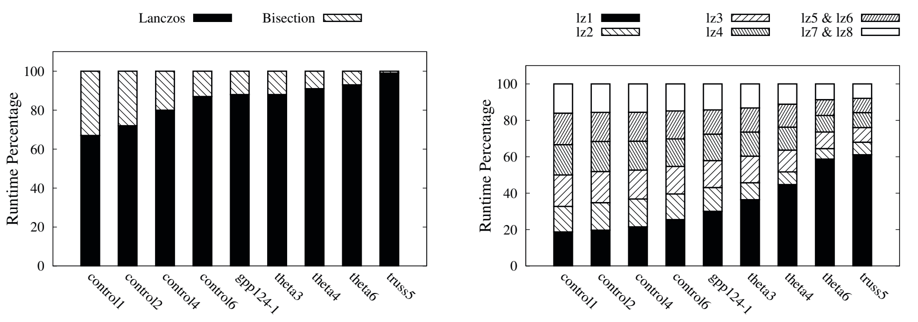
2.2 Sequential Runtime Analysis

We use sequential runtime analysis to find out the computationally intensive part of the flow shown in Fig. 1. We pick small to medium size matrices from SDPLIB [12], a collection of benchmarks for solving SDPs, and use Intel MKL library for sequential implementation on an Intel Xeon X5650. We show the runtime distribution in Fig. 2(a) and observe that it is the Lanczos method which takes most of the time reaching 99% for higher order matrices. On the other hand, the time taken by the bisection method is independent of the problem size as it only varies with the number of desired eigenvalues and their distribution [5]. We, therefore, focus our attention on parallelizing the Lanczos method.

The runtime distribution of the Lanczos method is plotted in Fig. 2(b) demonstrating that $lz1$ (matrix-vector multiplication) is the dominant operation which has a computational complexity of $\Theta(n^2)$. Hence our streaming architecture (see Section 3) shows how to use a parallel dot product circuit for accelerating this phase in a pipelined fashion.

2.3 Related Work

We briefly survey the existing FPGA-based eigensolvers. In Ahmedsaid *et al.* [9], Liu *et al.* [10] and Bravo *et al.* [11], the target applications involve Principal Component Analysis (PCA) where the matrix size does not usually go over 20 × 20. The direct method of Jacobi [5] and its variants are used for these small eigenvalue problems. The main reason behind using the Jacobi method is its inherent parallelism which can be exploited by systolic architectures requiring $\frac{n}{2}$ diagonal processors and $\frac{n(n-1)}{4}$ off-diagonal processors for a $n \times n$ matrix. Existing FPGA-based eigensolvers are not suitable for the extremal eigenvalues



(a) Extremal Eigenvalues Computation (b) The Lanczos Method

Fig. 2. Sequential Runtime Analysis

computation of matrices of our interest for two reasons. Firstly, they target very small matrices and are not resource efficient as the number of processors grows with $\Theta(n^2)$. Additionally, since the Jacobi method inherently computes all eigenvalues and eigenvectors, the approach utilized in these architectures is wasteful for computing only the extremal eigenvalues.

We propose an architecture based on the framework shown in Fig. 1 which is specialized for the extremal eigenvalues computation. Unlike previous approaches, the proposed architecture can address medium-size ($n \sim$ a few 100s) problems and is resource efficient as the number of floating-point units grows with $\Theta(n)$. Table 1 summarizes the previous work on eigenvalue computations on FPGA with the year, device, method, precision and the performance results.

Table 1. Comparison of FPGA-based Symmetric Eigenvalues Computation

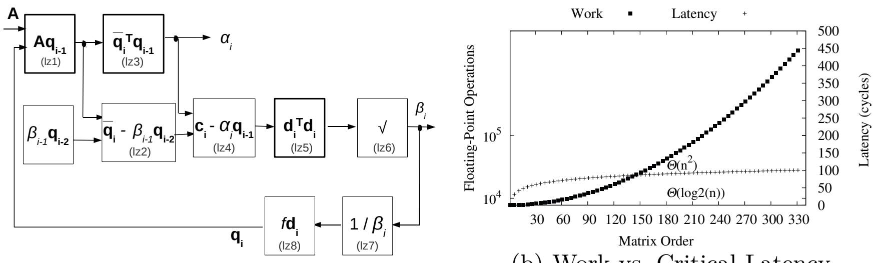
Ref.	**Year**	**Method**	**n**	**Device**	**Eigen-values**	**Freq.** MHz	**GFLOPs**	**Precision**	**Resources** (asymptotic)
[9]	2003	Direct (Exact Jacobi)	8	Virtex-E	All	84.44	Not Reported	fixed point (16-bit)	$\Theta(n^2)$
[10]	2006	Direct (Approx. Jacobi)	8	Virtex-II	All	70	Not Reported	fixed point (16-bit)	$\Theta(n^2)$
[11]	2006	Direct (Exact Jacobi)	16	Virtex-II Pro	All	110	0.243	fixed point (18-bit)	$\Theta(n^2)$
This Paper	2011	Iterative (Lanczos Method)	335	Virtex-6	Extremal	260	175	floating point (32-bit)	$\Theta(n)$

3 Parallelizing the Lanczos Method on FPGAs

3.1 Parallelism Potential

We can identify the parallel potential in the Lanczos method from its dataflow graph shown in Fig. 3(a). We plot the number of floating-point operations per Lanczos iteration as well as the critical latency assuming ideal parallel hardware as a function of the matrix size in Fig. 3(b). We find out that the work grows with $\Theta(n^2)$ due to dominant matrix-vector multiplication ($lz1$) whereas the latency grows with $\Theta(\log(n))$ *i.e.* the latency of a single dot product circuit [3] (assuming there are n such dot product circuits working in parallel to perform matrix-vector multiplication). Thus the Lanczos method has a high degree of parallel potential ($\sim 10^4$ for large n).

We use **associative reformulation of computation** to implement single dot product circuit as a reduction tree [3] and perform matrix-vector multiplication in a pipelined fashion as n dot products where a new dot product is launched every clock cycle.



(a) Dataflow (b) Work vs. Critical Latency

Fig. 3. Identifying Parallelism in the Lanczos Method

From the dataflow in Fig. 3(a), we can also observe **thread-level parallelism** as some of the operations may be performed in parallel *e.g.* ($lz2$) and ($lz3$). The operations ($lz2$), ($lz4$) and ($lz8$) can be unrolled completely with a latency of $\Theta(1)$ but since they contribute very little to the overall runtime of the system (see Section 2.2), we implement these sequentially.

We also use **pipeline parallelism** available in the architecture to solve multiple independent extremal eigenvalue problems arising in SDPs and eigenvalue based channel sensing while having the latency of solving a single extremal eigenvalue problem (see Section 3.3).

3.2 System Architecture

We present an architecture consisting of a parallelized Lanczos method coupled to a sequential architecture for the bisection method due to the latter's very low runtime contribution (see Section 2.2). We capture the high-level stream organization of the FPGA architecture in Fig 4.

The matrix A is loaded **only once** in the on-chip memory of the FPGA in a banked column fashion, and then the rows of A are streamed along with the Lanczos vector q_{i-1} to launch a new dot product in each clock cycle. We use deeply pipelined floating-point (FP) operators and aim to keep them busy all the time thus getting maximum throughput. The number of FP units is given by

$$\text{Total FP Units}(n) = 2n + 6. \tag{2}$$

Referring to (2), $2n$ - 1 units are used for the dot product circuit whereas 7 FP units are used for other operations.

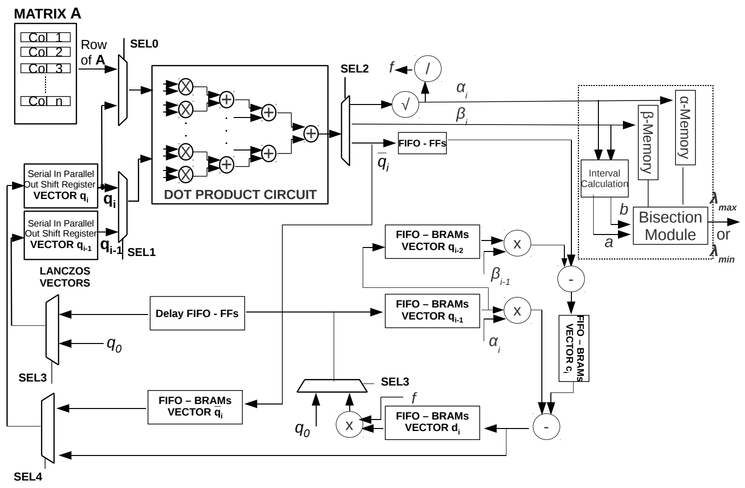

MATRIX A
Col 1
Col 2
Col 3
Col n
Row of A
SEL0
SEL1
SEL2
SEL3
SEL4
DOT PRODUCT CIRCUIT
Serial In Parallel Out Shift Register VECTOR q_i
Serial In Parallel Out Shift Register VECTOR q_{i-1}
q_i
q_{i-1}
LANCZOS VECTORS
FIFO - FFs
Delay FIFO - FFs
FIFO – BRAMs VECTOR q_{i-2}
FIFO – BRAMs VECTOR q_{i-1}
FIFO – BRAMs VECTOR q_i
FIFO – BRAMs VECTOR d_i
FIFO – BRAMs VECTOR c_i
q_0
f
α_i
β_i
β_{i-1}
$\bar{q}_i$
Interval Calculation
β-Memory
α-Memory
Bisection Module
a
b
λ_{max} or λ_{min}

Fig. 4. Partial schematic for the implementation of the Lanczos method and the bisection method displaying main components including a dot product circuit module, FIFOs for storing Lanczos vectors, banked memory arrangement for matrix A, two memories for storing α_i and β_i and a Bisection Module

The Interval Calculation module computes the initial interval for the extremal eigenvalue using the Gershgorin circle theorem [5] and the Bisection Module computes the extremal eigenvalue in that interval in a sequential fashion as shown in Algorithm 2.

3.3 Solving Multiple Extremal Eigenvalue Problems

For solving a single extremal eigenvalue problem, the deeply pipelined nature of the dot product circuit in Fig. 4 leads to high throughput but also considerable latency. As a result, the pipeline will be underutilized if only single problem is solved, therefore, mismatch between throughput and latency is exploited to solve multiple independent extremal eigenvalue problems. The initiation interval of this circuit is $n + 2$ clock cycles (for $lz1$, $lz3$ and $lz5$) after which a new problem can be streamed into this circuit. The pipeline depth (P) of the circuit is given by (4) which indicates how many problems can be active in the pipeline at one time.

$$\text{Latency per iteration}(n) = 3n + k_1\lceil \log_2 n \rceil + k_2. \tag{3}$$

$$\text{Pipeline Depth P}(n) = \left\lceil \frac{3n + k_1\lceil \log_2 n \rceil + k_2}{n+2} \right\rceil. \tag{4}$$

Referring to (3), the $3n$ comes from n cycles for ($lz1$) and $2n$ cycles for ($lz4$) and ($lz8$). The $\log_2 n$ term comes from the adder reduction tree and $k_1 = 36$ and $k_2 = 137$ derive from the latencies of the floating-point operators. We can see from (4) that the number of independent eigenvalue problems approaches to a constant value for large matrices (P $\rightarrow$ 5 as $n \rightarrow \infty$).

4 Methodology

The experimental setup for performance evaluation is summarized in Table 2 and dense matrices are extracted from SDPLIB [12] benchmarks shown in Table 3.

Table 2. Experimental Setup

Platform	**Peak GFLOPs** Single Precision	**Compiler**	**Libraries**	**Timing**
Intel Xeon X5650	127.8 [15]	gcc (4.4.3(-O3))	Intel MKL (10.2.4.032)	PAPI (4.1.1.0)
Nvidia GPU C2050	1050 [16]	nvcc	CUBLAS (3.2)	cudaEvent-Record()
Xilinx Virtex6-SX475T	450 [17]	Xilinx ISE (10.1)	Xilinx Coregen	ModelSim

Table 3. Benchmarks

Benchmark	**n**
control1	15
control2	30
control4	60
control6	90
gpp124-1	124
theta3	150
theta4	200
theta6	300
truss5	335

We implement the proposed architecture in VHDL and synthesize the circuit for Xilinx Virtex6-SX475T FPGA, a device with the largest number of DSP48Es and a large on-chip capacity. The placed and routed design has an operating frequency of 260 MHz. We find out that for a matrix of size 335×335, P is equal to 5 and we occupy nearly all the BRAMs available in the device. There is 50% utilization for the DSP48Es and 70% for the Slice LUTs and they show a linear increase as the number of floating-point units grows with $\Theta(n)$. Optimized Basic Linear Algebra Subroutine (BLAS) libraries are used for the multi-core and GPU implementations, an Intel MKL library for the multi-core and CUBLAS for the GPU. In the multi-core, the number of threads are set equal to the physical cores whereas in case of a GPU, the grid configuration is picked by CUBLAS automatically. We do not use multi-threading in the multi-core and GPU for solving multiple eigenvalues problem (see Section 6).

5 Results

We now present the performance achieved by our FPGA design, compare it with the multi-core and GPU and then discuss the underlying factors that explain our results.

5.1 FPGA Performance Evaluation

The peak and sustained single-precision floating-point performance of the FPGA is given by (5) and (6) respectively. For a matrix of size 335×335 and an operating frequency of 260 MHz, the peak performance of our design is 175 GFLOPs and sustained performance is 35 GFLOPs for a single problem and that the sustained performance approaches the peak performance when P problems are solved simultaneously as shown in Fig. 6(a). There is a linear increase in the performance with the problem size as the floating-point units grow with $\Theta(n)$.

$$\text{Peak Throughput} = \text{Total FP Units} = 2n + 6 \qquad \text{FLOPs/cycle.} \tag{5}$$

$$\text{Sustained Throughput} = \frac{\mathrm{P}\ (2n^2 + 8n)}{\mathrm{P}(n+2) + \mathrm{P} - 1}\ \text{FLOPs/cycle.} \tag{6}$$

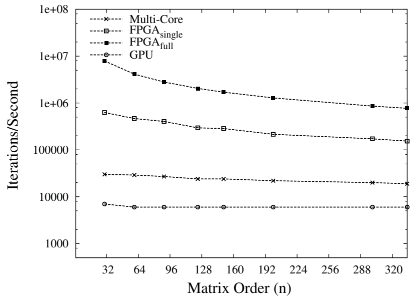
$$\text{Efficiency}(n) = \frac{\mathrm{P}(2n^2 + 8n)}{\text{Total FP Units} \times (\mathrm{P}(n+2) + \mathrm{P} - 1)}. \tag{7}$$

where $2n^2 + 8n$ represent the number of floating-point operations per Lanczos iteration. It is observed that even for low order matrices a high efficiency (70%) is achieved and the efficiency tends to 100% for large matrices. This is because the number of floating-point operators in dot product circuit grow linearly with n and by design the the dot product circuit remains busy.

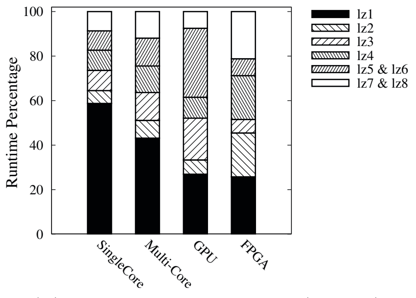
5.2 Comparison with Multi-core and GPU

We choose Iterations/second of the Lanczos method as the criterion for performance comparison plotted in Fig 5(a). We can observe that the FPGA outperforms both multi-core and the GPU. The runtime percentage of each part of the Lanczos method is plotted in Fig. 5(b) for different architectures. We can see a decrease in the runtime of ($lz1$) for GPU and the FPGA because of the high parallelism available in this operation. The Lanczos method is memory-bound due to dominant matrix-vector multiplication ($lz1$) with an arithmetic intensity of 0.5 FLOPs per byte (n^2 x 4 bytes of data (single-precision), $2\mathrm{n}^2$ FLOPs). It is for this reason the multi-core does not perform any better than single core for small matrices but show a speedup of 1.5× for large matrices as shown in 5(c). In the case of an FPGA, ($lz1$) is computed as n dot products where each new dot product is launched every clock cycle. Due to high on-chip memory bandwidth and streaming nature of the architecture, the overall design has much lower latency and high throughput. We get a speed up of 7.6-25.9× (12.5× geo. mean) using **associative reformulation of computation** and 1.04-1.22× (1.14× geo. mean) using **thread-level parallelism** with a combined speed up of 8.8-27.3× (13.4× geo. mean) for solving a single eigenvalue problem shown in Fig. 5(c). Using **pipeline parallelism** we additionally get a speed up of 4.3-19.1× (7.19× geo. mean) and, therefore deliver an overall speed up of 41-520× (103× geo. mean) when solving P independent eigenvalue problems shown in Fig. 5(c).

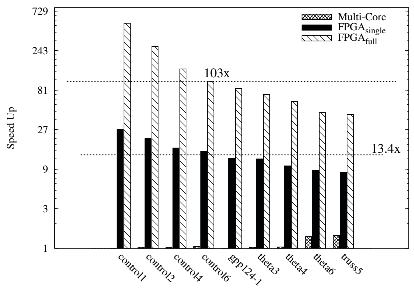
Although GPUs are highly efficient for dense linear algebra, we observe in Fig. 5(b) that the performance of GPU is even worse than the multi-core. This is due to medium-size data sets ($n \sim$ a few 100s) for which the GPU exhibits a performance less than 1 GFLOPs for BLAS 1 ($lz2$ to $lz8$) and BLAS 2 ($lz1$) operations [18]. Both BLAS 1 and BLAS 2 operations have very low arithmetic intensity to be exploited by multiple cores in the GPU. Additionally, we are using CUBLAS routines for matrix-vector multiplication which does not cache the matrix in the shared memory of the GPU and therefore the matrix is fetched repeatedly from off-chip memory in each iteration. When comparing



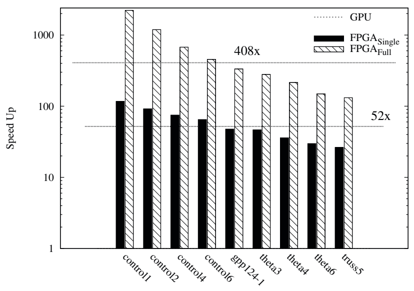
(a) Iterations/second



(b) Runtime Breakdown (truss5)

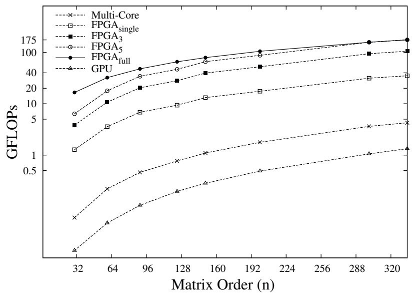


(c) FPGA vs. Multi-Core (Speed Up)

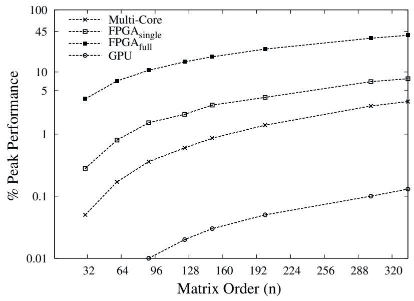


(d) FPGA vs. GPU (Speed Up)

Fig. 5. Performance Comparison ('single' is for 1 problem, 'full' is for P problems)



(a) GFLOPs



(b) % Peak Performance

Fig. 6. Raw Performance and Efficiency Comparison ('single' is for 1, 'full' is for P problems)

with our FPGA design, we get a speed up of 24.5-110.7× (49.4× geo. mean) using **associative reformulation of computation** and 1.04-1.22× (1.14× geo. mean) using **thread-level parallelism** with a combined speed up of 26.2-116x× (52.8× geo. mean) for solving a single eigenvalue problem shown in Fig. 5(d). Using **pipeline parallelism** we additionally get a speed up of 4.3-19.1× (7.19× geo. mean) and, therefore deliver an overall speed up of 131-2220× (408× geo. mean) when solving P independent eigenvalue problems shown in Fig. 5(d).

The raw performance of our FPGA design is compared with that of multi-core and GPU implementations in Fig. 6(a), and also their efficiency as a proportion of peak performance (see Table 2) is compared in Fig. 6(b). We can see for the maximum matrix size, the efficiency of the FPGA when solving single problem is approximately 7.79% whereas with P problems it increases to 38.9% of the peak performance. The efficiency of multi-core is around 3.34% and that of GPU is 0.13% and their efficiency increases with the problem size.

6 Future Work

We identify the following areas of research in this direction to further improve our results.

– We intend to further improve multi-core and GPU performance by solving multiple independent eigenvalue problems to hide memory latency.
– For small to medium matrices, we intend to have a customized GPU implementation of the Lanczos method where, like FPGA, the input matrix is explicitly loaded once in the shared memory and is reused for subsequent iterations. Similarly, we intend to consider I/O limitations in FPGA design.
– We are independently developing a BLAS library using the high level SCORE hardware compilation framework for stream computing [19]. We will exploit auto-tuning of implementation parameters to explore new architectures for the iterative numerical algorithms *e.g.* the Lanczos method.
– We would like to explore the algorithmic modifications in the Lanczos method for large-scale eigenvalue problems from different application areas where the matrix is fetched from off-chip memory in each iteration.

7 Conclusion

We present a pipelined streaming architecture coupled with high-bandwidth on-chip memories and demonstrate a sustained performance of 175 GFLOPs when solving multiple independent extremal eigenvalue problems. We show that the multi-core and GPU are underutilized for the medium-size data sets and achieve an efficiency of 3.34% and 0.13% respectively. Additionally, the dominant matrix-vector operation in the Lanczos method is memory-bound and has low arithmetic intensity (0.5 FLOPs/byte). On the other hand, the FPGA design exploits low latency high-bandwidth on-chip memories and the streaming computations to deliver a 41$\times$ speed up over an Intel Xeon X5650 and 131$\times$ over the Nvidia C2050 for a matrix of size 335$\times$335. We therefore highlight iterative numerical algorithms with high memory bandwidth requirements but medium-size data sets as highly appropriate for FPGA acceleration.

References

1. Underwood, K.: FPGAs vs. CPUs: trends in peak floating-point performance. In: Proc. ACM/SIGDA 12th International Symposium on Field programmable Gate Arrays, pp. 171–180 (2004)

2. Lopes, A.R., Constantinides, G.A.: A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation. In: Woods, R., Compton, K., Bouganis, C., Diniz, P.C. (eds.) ARC 2008. LNCS, vol. 4943, pp. 75–86. Springer, Heidelberg (2008)
3. Boland, D., Constantinides, G.: An FPGA-based implementation of the MINRES algorithm. In: Proc. Field Programmable Logic and Applications, pp. 379–384 (2008)
4. Kapre, N., DeHon, A.: Parallelizing sparse Matrix Solve for SPICE circuit simulation using FPGAs. In: Proc. Field-Programmable Technology, pp. 190–198 (2009)
5. Golub, G.H., Van Loan, C.F.: Matrix Computations, 3rd edn. The Johns Hopkins University Press, Baltimore (1996)
6. Toh, K.C.: A note on the calculation of step-lengths in interior-point methods for semidefinite programming. J. Computational Optimization and Applications 21(3), 301–310 (1999)
7. Zeng, Y., Koh, C.L., Liang, Y.C.: Maximum eigenvalue detection: theory and application. In: Proc. IEEE International Conference on Communications, pp. 4160–4164 (2008)
8. Demmel, J.W.: Applied numerical linear algebra. Society for Industrial and Applied Mathematics, Philadelphia (1997)
9. Ahmedsaid, A., Amira, A., Bouridane, A.: Improved SVD systolic array and implementation on FPGA. In: Proc. Field-Programmable Technology, pp. 35–42 (2003)
10. Liu, Y., Bouganis, C.S., Cheung, P.Y.K., Leong, P.H.W., Motley, S.J.: Hardware efficient architectures for eigenvalue computation. In: Proc. Design Automation & Test in Europe, p. 202 (2006)
11. Bravo, I., Jiménez, P., Mazo, M., Lázaro, J.L., Gardel, A.: Implementation in FPGAs of Jacobi method to solve the eigenvalue and eigenvector problem. In: Proc. Field Programmable Logic and Applications, pp. 1–4 (2006)
12. Brochers, B.: SDPLIB 1.2, a library of semidefinite programming test problems. Optimization Methods and Software 11(1-4), 683–690 (1999)
13. Intel Math Kernel Library 10.2.4.032 (2010), `http://software.intel.com/en-us/articles/intel-mkl/`
14. CUBLAS 3.2 (2010), `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf`
15. Intel microprocessor export compliance metrics (2010), `http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf`
16. Nvidia Tesla C2050 (2010), `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`
17. Sundararajan, P.: High Performance Computing using FPGAs (2010), `http://www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf`
18. Anzt, H., Hahn, T., Heuveline, V., Rocker, B.: GPU Accelerated Scientific Computing: Evaluation of the NVIDIA Fermi Architecture; Elementary Kernels and Linear Solvers, KIT (2010)
19. Caspi, E., Chu, M., Huang, R., Yeh, J., Wawrzynek, J., DeHon, A.: Stream computations organized for reconfigurable execution (SCORE). In: Proc. Field Programmable Logic and Applications, pp. 605–614 (2000)