Accelerating Double Precision Floating-point Hessenberg Reduction on FPGA and Multicore Architectures

Miaoqing Huang
CSCE Department, University of Arkansas
mqhuang@uark.edu

Lingyuan Wang, Tarek El-Ghazawi
ECE Department, The George Washington University
{lwanghpc,tarek}@gwu.edu

Abstract—**Double precision floating-point performance is critical for hardware acceleration technologies to be adopted by domain scientists. In this work we use the Hessenberg reduction to demonstrate the potential of FPGAs and GPUs for obtaining satisfactory double precision floating-point performance. Currently a Xeon (Nehalem) 2.26 GHz CPU can outperform Xilinx Virtex4LX200 by 3.6 folds. However, given higher frequency, more hardware resources and local memory banks, FPGAs have the potential to outperform multicore CPUs in the near future. On the GPU side, a GTX 480 (Fermi) achieves 19.4× speedup against the Xeon CPU. Based on the current trend, GPUs will keep widening the advantages against both FPGAs and CPUs on double precision floating-point performance.**

I. Introduction

The performance of double precision (DP) floating-point operations is critical for hardware acceleration technologies (e.g., FPGAs and GPUs) to be widely accepted by domain scientists. Traditionally the floating-point performance of FPGAs was comparatively poor due to the lack of built-in floating-point units. Since the transistor count in a chip still follows Moore's law, it becomes possible to include high-performance floating-point operators in large FPGA devices. In this work, we use the Hessenberg reduction as a case study to demonstrate the potential of floating-point performance of FPGA devices. By building fully-pipelined complex processing units and efficiently utilizing the local memory banks, the FPGA device is capable of achieving comparable performance to modern multicore microprocessors when running at 100 MHz, which is one order of magnitude lower than microprocessors.

The other leading hardware acceleration technology, Graphics Processing Units (GPUs), tries to improve the performance by adopting streaming processing paradigm. Hundreds of streaming processors provide a massive parallel processing power, which already demonstrates significant advantage over microprocessors. With the new addition of powerful double precision floating-point units in recently released GPUs (i.e., Fermi), it is shown in this work that a single GPU is capable of outperforming a high end Xeon (Nehalem) quad-core processor by 19.4 folds.

The selection of Hessenberg reduction in this work is motivated by two factors. (i) The matrix operations and the loop dependence in Hessenberg reduction are ubiquitous in scientific applications. (ii) Hessenberg reduction is the first phase in solving the eigenvalue problem, which is a very important problem in scientific domain. Given a complex square matrix $A \in \mathbb{C}^{n\times n}$, an eigenvalue λ and its associated eigenvector $\mathbf{v}$ are a pair obeying the relation $A\mathbf{v} = \lambda\mathbf{v}$. The QR algorithm [1] is accepted as a practical solution to deal with general unsymmetric square matrices. The first phase of the QR algorithm is to reduce the original matrix A to its upper Hessenberg form H, as shown in (1). This phase is called Hessenberg reduction. Hessenberg reduction is carried out by applying the Householder reflection for $n-2$ iterations (see Alg. 1), where n is the rank of the original matrix A.

$$\underset{A}{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}} \Longrightarrow \underset{\text{Hessenberg } H}{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix}} \tag{1}$$

Algorithm 1: Hessenberg reduction (vector-based)

```
    Input: A square complex matrix $A$ with rank $n$
    Output: The Hessenberg matrix $H$
1.1 for $k=0$ to $n-3$ do
1.2     $v_k = \mathbf{House}(A_{k+1:n-1,k})$;          /*Refer to [1]*/
1.3     $A_{k+1:n-1,k:n-1} = A_{k+1:n-1,k:n-1} - 2v_k(v_k^* A_{k+1:n-1,k:n-1})$;
1.4     $A_{0:n-1,k+1:n-1} = A_{0:n-1,k+1:n-1} - 2(A_{0:n-1,k+1:n-1}v_k)v_k^*$;
```

II. Hessenberg Reduction on Three Platforms

A. FPGA Implementation

SGI's Altix RASC RC100 reconfigurable computer is selected as the platform for FPGA implementation. The FPGA device, Xilinx Virtex-4LX200, is equipped with 5 banks of SRAM for local data storage, each of which is 8 MB and has separate 64-bit read port and write port.

The total number of clock cycles required to reduce a matrix of rank n to its Hessenberg form is $\frac{5}{2}n^3 - \frac{9}{2}n - 11$ clock cycles by ignoring all latencies and delays. The detail of the hardware implementation has been reported in [2]. The two basic techniques are (i) building fully pipelined complex processing units using basic floating-point units, and (ii) distributing data into multiple local memory banks and carefully designing data access pattern to maximize processing parallelism. In the hardware implementation of the Hessenberg reduction, two physical local memory banks are combined to form a 128-bit wide logical memory bank so that each memory entry can store one complete matrix entry. In our implementation, two 128-bit wide local memory banks are

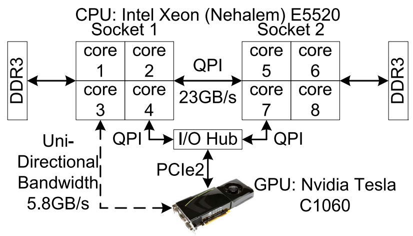


Fig. 1. The heterogeneous CPU-GPU board

used, each of which has a size of 16 MB. Each memory bank is divided evenly into 4 regions. We store a whole matrix into one region, which limits the maximize size of matrix to 480×480. In order to maximize the data processing parallelism, we distributed the data (including the original data and the intermediate data) accordingly between these two local memory banks so that there is no competition regarding the memory access ports within each step.

The hardware implementation of Hessenberg reduction occupies 56,520 (63%) slices on the target FPGA device and runs at 100 MHz, which is mainly due to the delay of the critical path in the control logic.

B. GPU Implementation

We have implemented Alg. 1 on both Nvidia Tesla C1060 and GeForce GTX 480 (Fermi) GPUs. Tesla C1060 (architecture code-named GT200) features 30 cores (namely Streaming Multiprocessors), each of which is further composed of eight single precision (SP) floating-point CUDA processors and one double precision (DP) floating-point processor, with 16KB on-chip storage called shared memory and 64KB of register windows for massive threading. The total 240 (SP) + 30 (DP) floating point processors have an observed peak performance of 78 GFLOPS for double precision. The Tesla GPU is equipped with 4 GB GDDR3 memory on board with the theoretical memory bandwidth of 102 GB/s.

The latest GPU offered by Nvidia is code-named as Fermi, which takes a significant leap forward in architecture highlighted by features such as improved double precision performance and configurable cache hierarchy. The model GTX 480 used in our experiments is composed of 15 newly designed Streaming Multiprocessors. Each SM features 32 CUDA cores and is capable of 16 double precision fused multiply-add operations per clock, which is an 8× improvement over the GT200 architecture. Another key architectural difference is that Fermi has two instruction dispatch units and most instructions can be dual-issued, which is different from the HyperThreads used in the Intel Nehalem processors. Two HyperThreads within a single core of Nehalem processors share a single instruction fetch and decoding unit.

Both GPUs communicate with a host CPU via PCI Express 2 ×16 bus (as shown in Fig. 1), with observed uni-directional bandwidth at 5.8 GB/s.

The GPU implementations are developed using CUDA [3]. The vector-based diagonal factorization is composed of a major outer loop that factorizes one column/row per step. Unfortunately, advanced features offered on the GPU such as asynchronized communication/computation and concurrently kernel execution cannot be used for such an algorithm, as dependency exists among the outer loops and all inner steps. Therefore the GPU implementation suffers from low occupancy for small problem sizes. In order to optimize the GPU implementation, firstly we managed to squeeze every inner computation step except the Householder generator into the GPU to keep the entire matrix remaining in the GPU memory throughout the computation. Thereby we managed to minimize the round trip communication overhead to approximately 5% of overall execution time. All kernels are further incrementally optimized through memory coalescing, using of shared memory and assigning more work per thread. The configurable L1 cache on the Fermi GPU introduces more design tradeoffs for users. In our experiments, for kernels with limited or no usage of shared memory, configuring the L1 to be 48KB can yield an approximately 10% improvement. Moreover, we found that the multi-dimensional threads and blocks configuration can also affect the cache performance, especially when the performance differences are examined on both GT200 and Fermi. We achieved the best performance mostly at the thread configuration of 32×8 for the Fermi GPU.

TABLE I
PLATFORM CHARACTERISTICS

Criteria	Xeon (Nehalem)	Tesla C1060	GTX 480
Cores	4	240/30	480
Frequency (GHz)	2.26	1.3	1.4
DP GFLOPs	36	78	672
Memory Bandwidth (GB/s)	25.6	102	177.4

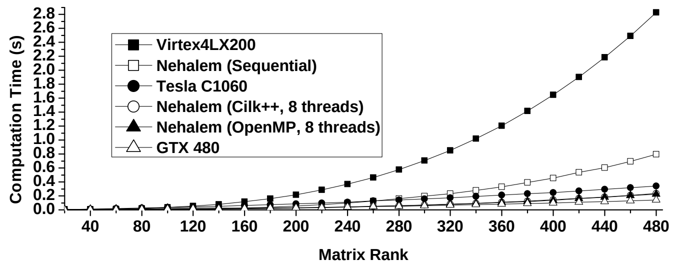
C. x86 Implementation

The software platform is a dual-socket Intel Xeon (Nehalem) system. The CPU is clocked at 2.26GHz with 8MB shared L3 cache and 12GB DDR3 memory (total 24GB for the entire system). The theoretical peak double precision floating point performance is 36 GFLOP/S for each CPU.

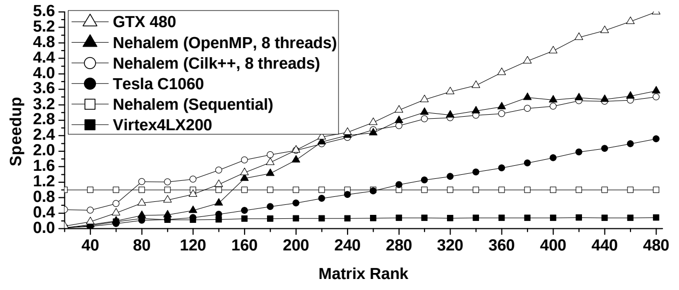
Our CPU versions are parallelized using OpenMP [4] and Intel Cilk++ [5], respectively. The critical computing intensive paths are parallelized by multiple threads first then further vectorized by the compiler utilizing the SSE units per core. Specifically, in order to achieve better scalability on all eight cores of both CPUs, we manually optimized our OpenMP and Cilk++ codes for better data locality control and further applied *numactl* to bind threads to physical CPU cores to avoid the NUMA penalty. Such an optimization significantly improves the overall performance on two CPUs for up to 60%.

III. PERFORMANCE COMPARISON

In this section, we first compare the performance of the vector-based implementation on FPGA, GPU and microprocessor. In order to demonstrate the complete potential of double precision floating-point performance on GPU, we compare it with both single-processor (8 threads) and dual-processor



(a) Computation time



(b) Speedup against sequential implementation on Nehalem

Fig. 2. Performance comparison of the vector-based Hessenberg reduction

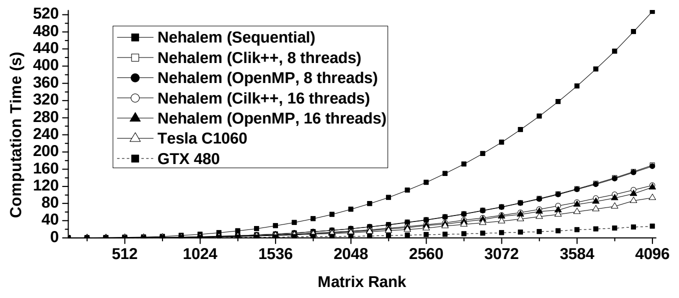


Fig. 3. Performance scalability of vector-based implementations

(16 threads) x86 implementations and test the rank of the matrix up to 4,096×4,096.

A. Vector-based Hessenberg Reduction

The vector-based Hessenberg reduction has been realized in 6 different implementations as follows.

- The FPGA implementation;
- The Tesla C1060 GPU implementation;
- The GTX 480 GPU implementation;
- The parallel software implementation of Cilk++;
- The parallel software implementation of OpenMP;
- The sequential software implementation on Xeon E5520.

The comparison among these 6 implementations is illustrated in Fig. 2. It can be found that the FPGA implementation is outperformed by other two technologies at almost all cases. The inferior performance of FPGA is mainly due to three factors. (i) The FPGA device is running at a very low frequency, i.e., 100 MHz. (ii) The direct implementation of Alg. 1 is a sequential process due to the data dependency. Although we have tried to parallelize the hardware implementation to the extreme, its performance is easily surpassed by modern multicore processors with improved design on cache and SSE when dealing with sequential applications such as Hessenberg reduction. (iii) The 5 local memory banks on the current platform become the limiting factor to increase the parallelism in the hardware implementation. More memory banks are desired to achieve higher parallelism on FPGA device.

From Fig. 2, it is evident that it will be beneficial to implement the application on GPU as the matrix rank increases. The Tesla implementation surpasses the sequential software implementation at rank 260 and then approaches the parallel software implementation afterwards. Fermi consistently outperforms GT200 for approximately 4×. Clearly the GT200 performance is limited by the lack of DP capability, as shown in Table I.

B. Performance Scalability

In the previous test, we limit the matrix rank at 480 because it is the biggest size the FPGA design can accommodate. In the meantime, it is clearly demonstrated that GPUs are capable of outperforming multicore CPUs as the matrix rank increases. In order to completely show the performance potential of GPUs, we compare them with 8-thread and 16-thread x86 implementations on the platform (shown on Fig. 1) with the matrix rank up to 4,096. By observing Fig. 3, the GT200 performance is generally close to or slightly better than the dual socket Nehalem (16-thread case). The GTX 480 GPU outperforms all other versions consistently with a big margin, clearly demonstrating the advantage of the newly designed Fermi architecture on DP performance.

IV. Conclusion

In this work, we use a Hessenberg reduction application on FPGA and GPU to demonstrate their capabilities of double precision floating-point performance by comparing with parallel implementations on modern x86 microprocessors. Although the FPGA is outperformed by microprocessor, higher frequency and more local memory banks provide the opportunity for the FPGA device to regain the advantages. On the other hand, GPU has clearly established its advantages by bringing abundant double precision floating-point units into the device and enjoying high bandwidth with global shared on-board memory.

References

[1] J. G. F. Francis, "The QR transformation, I," *The Computer Journal*, vol. 4, no. 3, pp. 265–271, 1961.

[2] M. Huang and O. Kilic, "Reaping the processing potential of FPGA on double-precision floating-point operations: an eigenvalue solver case study," in *Proc. the 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2010)*, May 2010, pp. 95–102.

[3] *Nvidia CUDA Programming Guide 2.3.1*, Nvidia Corporation, Aug. 2009.

[4] *http://openmp.org*.

[5] *Intel Cilk++ Software Development Kit*, Intel Corporation, Feb. 2010.