GPU vs FPGA : A comparative analysis for non-standard precision

Umar Ibrahim Minhas, Samuel Bayliss, and George A. Constantinides

Department of Electrical and Electronic Engineering
Imperial College London
South Kensington Campus, London SW7 2AZ
umar.minhas12@imperial.ac.uk

Abstract. FPGAs and GPUs are increasingly used in a range of high performance computing applications. When implementing numerical algorithms on either platform, we can choose to represent operands with different levels of accuracy. A trade-off exists between the numerical accuracy of arithmetic operators and the resources needed to implement them. Where algorithmic requirements for numerical stability are captured in a design description, this trade-off can be exploited to optimize performance by using high-accuracy operators only where they are most required. Support for half and double-double floating point representations allows additional flexibility to achieve this. The aim of this work is to study the language and hardware support, and the achievable peak performance for non-standard precisions on a GPU and an FPGA. A compute intensive program, matrix-matrix multiply, is selected as a benchmark and implemented for various different matrix sizes. The results show that for large-enough matrices, GPUs out-perform FPGA-based implementations but for some smaller matrix sizes, specialized FPGA floating-point operators for half and double-double precision can deliver higher throughput than implementation on a GPU.

Keywords: GPU, FPGA, High Performance Computing (HPC), Non-standard Precision, Half Precision, Double-double Precision

1 Introduction

Over the past three decades, improvements in semiconductor process technology have delivered an exponential increase, over time, in the number of transistors that can be economically manufactured on a single silicon die. Until recently, each shrink in process technology was accompanied by a reduction of supply voltage in line with scaling rules established by Dennard et al.[1]. In recent process technologies, where leakage power dominates, higher transistor density has been accompanied by reductions in energy-efficiency. The industry response to this has been a shift to multi-core parallel processing.

In the world of High Performance Computing (HPC), this has meant an increase in the use of specialised parallel architectures such as Graphic Processing

Units (GPUs) and Field Programmable Gate Arrays (FPGAs) in heterogeneous supercomputers to achieve energy-efficient computation.

Numerical Analysts continually seek to improve the accuracy of numerical approximation under the constraint of finite arithmetic precision. For floating point computation on both GPUs and FPGAs, there is a trade-off between the resource utilization and precision of arithmetic primitives. GPUs commonly contain specialized units for computation on single and double precision floating-point operands and FPGAs contain hardened DSP primitives which may be composed into floating point units supporting a variety of different precisions. Existing studies show that the area, power and delay of floating point operations is increased when double precision operators are used in place of single precision operators. However very litle existing work seeks to answer the question of how to efficiently implement a wider range of different precisions on both FPGAs and GPUs. The unique contributions of this paper are :

- An investigation into how half (16-bit) and double-double (128-bit) precision floating point operations can be efficiently implemented on both GPUs and FPGAs.
- A comparison of the performance achievable by GPUs and FPGAs for half and double-double precision computations.

Our hope is that this can be used to guide future work on 'heterogeneous' computing. This will allow numerical analysts to efficiently implement high-precision at the points where it is needed in a computation, and optimize for power and throughput in areas where it can be safely reduced.

2 Methodology

In this section we provide an overview of the equipment and benchmark used for our experiments. We follow this in Section 3 with a deeper look at the GPU and FPGA implementations and report comparative performance in Section 4.

2.1 Equipment

For these experiments, the NVidia Tesla C1060 [2] has been selected as an exemplary GPU architecture. The Tesla range of products is purposely designed for parallel high performance computing and GPGPU programming. The C1060 is implemented in a 55nm process and has hardware support for IEEE-754 compliant single and double precision data-types. It has 240 streaming processor cores running at 1.3GHz. These cores are clustered together in groups of 8 as SIMD processors, with those 8 cores sharing a 16KByte local memory. Either OpenCL or CUDA APIs can be used for programming, expressing computation as explicitly parallel kernels for offload to SIMD processors within the GPU.

We have selected a Xilinx Virtex-6 [3] FPGA (XC6VLX195T) for this comparison study. This device is implemented in a 40nm process and our synthesis scripts targeted a 250MHz clock rate for each implementation. It was programmed using RTL design-entry incorporating floating-point cores from Xilinx Coregen[4] and FloPoCo[5].

2.2 Benchmark and metrics

Matrix-Matrix multiplication has been chosen as a benchmark algorithm for this work. It forms a core component of many numerical algorithms and the cornerstone of the LINPACK benchmarks. For input matrices of size $n \times n$, the computational complexity of a naïve matrix-matrix multiplication algorithm varies as $O(n^3)$, exceeding the communications requirement which scales as $O(n^2)$. This means that for large-enough matrices, we can be sure that the algorithm performance is bounded by the computational capabilities of our two platforms. We measure the performance of our applications by counting the throughput in FLOPs (floating-point operations per second). Both peak power and total energy consumption are also useful metrics in large HPC environments, but a comparison of these lies outside the scope of this work.

3 Implementation

In this section, we consider how to design an efficient implementation of a matrix-matrix multiplication on GPUs and FPGAs using half and double-double precision arithmetic operands. Section 3.1 and Section 3.2 demonstrate how this can be achieved on a GPU and FPGA platform respectively. We then follow this in Section 4 with a performance comparison.

3.1 GPU Implementation

Many optimized GPU implementations exist. We have selected the SGEMM implementation from [6] which was distributed in early versions of the CUBLAS [7] library. The Tesla C1060 GPU does not have dedicated support for *computation* using half-precision data types. We instead implement these using native single-precision floating-point units. However, we can still exploit reduced-precision operands to improve performance by improving the storage density of operands in global and local memories. Where the asymptotic compute bounds are not reached, we would expect this to deliver an improvement in performance. In global-memory, CUDA stores half-precision operands as `unsigned short` and provides interfaces via `__half2float()` and `__float2half()` intrinsics [8]. These conversions are hardware-accelerated single-cycle GPU operations.

Neither CUDA nor the GPU architecture provide hardware support for double-double precision operations. Researchers have built libraries for use of double-double precision [9][10] based on underlying double precision primitives. We chose a library based on [9] for our experiments. The library stores double-double precision numbers in an abstract data type and provides functions for

arithmetic operations using double-double precision based on the IEEE-754 standard. A high accuracy double-double addition requires 16 basic double-precision operations and a multiplication requires 3 basic and 4 fused multiply-add double-precision operations.

The library only supports double-double precision operations on the GPU (not on the host processor), therefore numbers are transferred to the device using double-precision and then transformed to double-double precision using a library function. Since a double-double number was represented as a struct of two double precision numbers and additional temporary numbers were used for mathematical operations, the additional register pressure means computation must spill into local memory within each GPU multiprocessor. This has an impact on performance that is explored in Section 4.

3.2 FPGA Implementation

Current FPGAs do not include hardened floating-point units. Instead floating points units are constructed using a combination of hardened DSP-block primitives and LUT logic. We have used floating-point cores from two vendors to implement our experiments. Xilinx LogiCORE IP [4] (Version 5.0) exploits the low-level architectural details of Xilinx FPGA fabric to provide high quality floating-point operators. For precision beyond 80 bits, arithmetic operators generated by FloPoCo [5] were used. FloPoCo does not offer as wide a range of options as Xilinx IP cores for trading-off latency and resource usage but allows us to generate cores with a wider range of precisions. All the operators selected were fully pipelined.

In our experiments, we have aimed to maximize the design throughput by targeting full utilization of the FPGA. To ensure this, we ran preliminary experiments to calculate how many adders and multipliers would fit on the selected FPGA. For matrix-matrix multiplication of square matrices of order $n \times n$ (where n is a power of 2), a naïve algorithm requires n multipliers and $n - 1$ adders. After having an estimate of resources used by each precision's arithmetic operators, the next step in the design process was to divide up the total available resources on the FPGA and find the approximate number of arithmetic units that can be implemented.

For matrix-matrix multiplication of large matrices, blocking was used to divide up the matrix. The block size should be small enough to be stored on the on-chip RAM but large enough to hide external memory latency. A moderate size of 64×64 was chosen that suits both conditions. However, for this section it is assumed that all the data is already stored in on-chip Block RAM for multiplying 64x64 matrix and the results are to be stored in the same memory as well. A discussion of more realistic memory architecture can be found in [11] but for large enough matrices, the throughput of calculations on individual 64×64 blocks should limit performance.

The FPGA logic utilization supports an implementation of 2^n multipliers and $2^n - 1$ adders for a particular precision. This means, a block of 2^n row elements

and 2^n column elements can all be multiplied in parallel. A binary reduction tree is then used to produce each element of the output matrix.

4 Results

In this section, we first consider the performance of different numerical precisions on each separate platform. We follow this with a comparison of the relative performance achievable on an FPGA or GPU when deploying different numerical precisions.

For the GPU platform, Figure 1 shows the relative performance using each different precision. For each precision, the vertical-axis shows the number of floating point operations which can be completed each second (in GFLOPs). In all precisions, where large enough matrices are multiplied, the problem is compute-bound by the throughput of the individual floating-point operators. This throughput varies for each precision with double-precision calculations achieving approximately 2.3× lower throughput than single-precision operators (200GFLOPs vs 85GFLOPs).

The double-double precision computation implemented on the GPU platform achieves fewer than 8GFLOPs, approximately 11× slower than the double precision computation. This is in line with predictions since each double-double operation requires 7 double-precision operators and each addition requires 16 basic double precision operations. The half-precision floating point results deliver on our expectation that when compute-bound, they mirror the single-precision results.

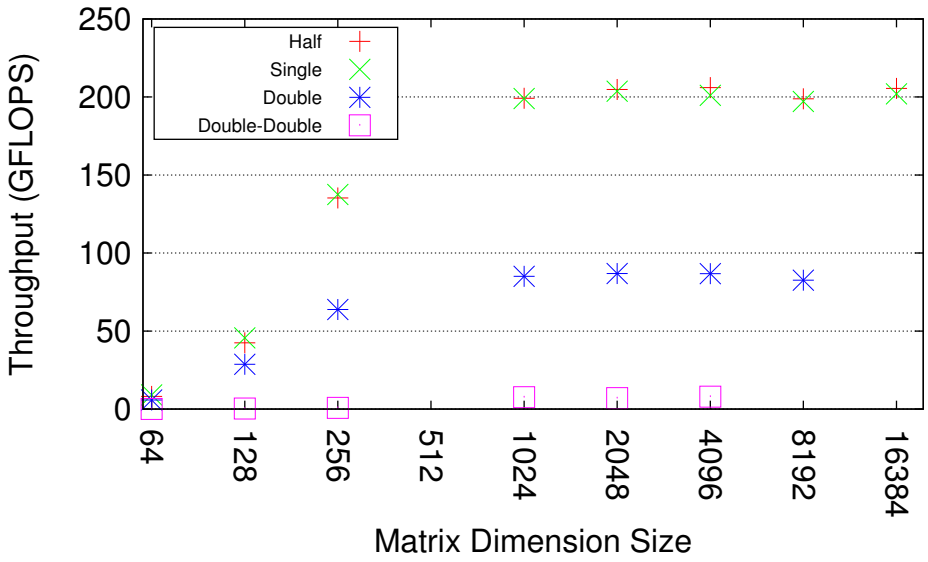

Fig. 1: Comparison of GPU performance using different matrix sizes and operand precisions.

Table 1: Logic utilization for FPGA implementations with varying precisions.

Logic Utilization	Half	Single	Double	Dbl-dbl
Number of Slice Registers	35%	61%	43%	32%
Number of Slice LUTs	56%	92%	66%	82%
Number of Block RAM/FIFO	83%	83%	55%	27%
Number of DSP48E1s	80%	60%	100%	87%

Each different precision becomes compute-bound at a different matrix size. This reflects the different memory-system overheads for each implementation. Curiously, the half-precision does not deviate from the single-precision results. This indicates that reduced-precision data storage has not delivered significant benefits in improving memory-throughput. An interesting comparison would be to compare these results with those generated using different matrix-blocking sizes and thread-group allocations. This might be achieved using an auto-tuned GPU library such as Atlas [12]

For the FPGA platform, Table 1 shows how logic utilization varies for various target precisions. Each precision must match the ratio of addition operators to multiplication operators necessary for the algorithm to the ratio which can be delivered by the FPGA fabric. For single and double precision implementations, the designs utilize greater than 90% of LUT and DSP48 resources respectively. This high-utilization is not achieved for the half precision implementation, where Block RAM availability, specifically port availability, limits the number of operators which may be implemented on the FPGA.

Table 2 shows the performance that can be achieved using the FPGA implementations. All implementations target 64×64 matrix block sizes. The table indicates that single precision FPGA implementations achieve approximately $3\times$ the compute throughput of double-precision implementations ($\sim$68GFLOPS vs $\sim$23GFLOPS). Where double-double precision is used, the FPGA implementation achieves 5GFLOPS, a $16\times$ reduction over the performance of single precision. The specialized compute operators for half-precision operators mean a half-precision implementation can reach 83GFLOPS, a $1.2\times$ improvement over single precision.

To compare the performance of the two platforms, Table 2 shows the achievable performance (in GFLOPs) for different matrix sizes and precisions. Where results are presented in grey, they indicate the FPGA implementation has achieved greater computational throughput than the equivalent GPU implementation. These cases occur when smaller matrices are multiplied and is a direct result of the large overhead of executing a GPU kernel and higher memory latency in our GPU implementation.

These results are represented graphically in Figure 2. This graph makes clear the very significant performance penalty that GPUs face when moving from hardware-supported single and double precision operation to using non-native double-double precision operations. By comparison, the FPGA implementations see a much more gradual degradation of performance as the numerical accu-

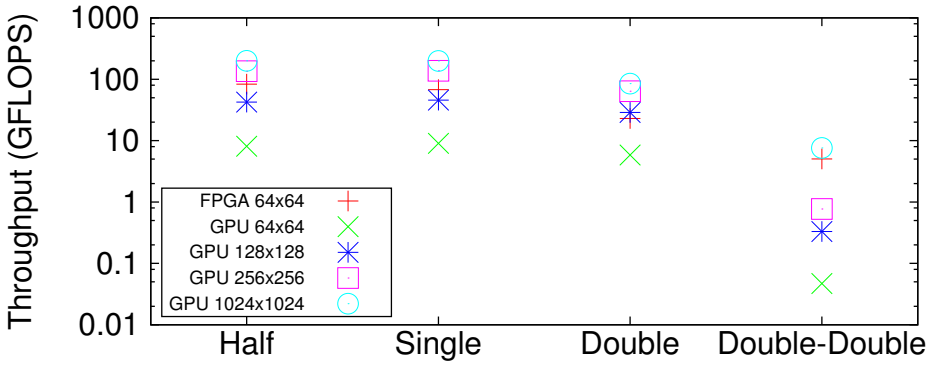

Fig. 2: Comparison of FPGA and GPU performance for various precisions and matrix sizes.

Table 2: Performance (in GFLOPS) for varying precisions and matrix sizes in FPGA and GPU implementations.

Precision	Performance (GFLOPS)				
	FPGA 64x64 (on-chip)	GPU 64x64	GPU 128x128	GPU 256x256	GPU 1024x1024
Half	83.12	8.09	42.45	135.3	199.17
Single	67.68	9.039	45.59	137.5	198.98
Double	22.9	5.825	28.72	63.79	85.06
Double-Double	4.93	0.047	0.33	0.771	7.60

racy of operators is increased. This can be attributed to the specialized circuits produced by the Coregen [4] and FloPoCo[5] tools.

Overall, the results suggest that for large dense matrices and for sufficiently large matrices, GPU platforms deliver greater throughput than competing FPGA platforms, but for smaller matrices, the combination of specialized operator structures and the absence of large kernel setup times makes FPGA implementation competitive.

5 Conclusion and Future Work

This work focused on analyzing the performance of non-standard precision on GPU and FPGA. An arithmetically dense program, matrix-matrix multiply, was implemented for various data sizes. The results showed that GPU implementations outperforms FPGA for larger data sizes but underperform for smaller sizes where the memory latency and kernel start overhead become significant. FPGAs have good vendor support for custom floating-point formats and we would expect this gap to increase further, in favour of FPGA implementation if even more exotic number representation were selected.

While this work has delivered a comparison of the *throughput* of the two platforms, other performance metrics warrant further investigation. Firstly, the accuracy of computations; in this work the non-standard precision results were converted to single or double precision on the same platform and compared for correctness with a verified matrix-matrix multiplication running on the same platform. Because of the error induced while converting between formats, this error criteria may be unduly lenient. However for future work, the results can be compared with a verified library to precisely estimate the correctness of non-standard computations.

Comparison of power and energy usage will also be an interesting study. With massively parallel supercomputing systems comprising of hundreds of GPUs and FPGAs a specific value of watts/GFLOPS for non-standard precision can be a key specification in system design.

Finally, the comparison can be extended to different benchmarks. GPUs are tuned to deliver very high performance for matrix-matrix multiply calculations. We would expect the gap in performance between the two platforms to narrow in other algorithms where the flexible memory systems and efficient synchronization allow FPGAs to achieve a higher proportion of peak performance than GPUs.

References

1. Dennard, R., Gaensslen, F., Rideout, V., Bassous, E., LeBlanc, A.: Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. Solid-State Circuits, IEEE Journal of **9**(5) (1974) 256–268
2. NVIDIA Corporation, Santa Clara, U.: Tesla C1060 Computing Processor Board. (January 2010)
3. Xilinx Corporation: Virtex-6 Family Overview. Technical Report DS150 (Jan 2012)
4. Xilinx Corporation: LogiCORE Floating-Point Operator v5.0. (2011)
5. De Dinechin, F., Pasca, B.: Designing Custom Arithmetic Data Paths with FloPoCo. Design & Test of Computers, IEEE **28**(4) (2011) 18–27
6. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune Dense Linear Algebra. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press (2008) 31
7. NVIDIA Corporation: CUBLAS library v5.5. Technical report (2013)
8. NVIDIA Corporation: CUDA library documentation 4.1. http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online
9. Thall, A.: Extended-Precision Floating-Point Numbers for GPU Computation. In: ACM SIGGRAPH 2006 Research posters, ACM (2006) 52
10. Lu, M., He, B., Luo, Q.: Supporting Extended Precision on Graphics Processors. In: Proceedings of the Sixth International Workshop on Data Management on New Hardware, ACM (2010) 19–26
11. Minhas, U.: GPU vs FPGA: A Comparative Performance Analysis for Non-Standard Precision. Master's thesis, Imperial College London (2013)
12. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimizations of Software and the ATLAS project. Parallel Computing **27**(12) (2001) 3 – 35