Evaluation of a Floating-Point Intensive Kernel on FPGA

A Case Study of Geodesic Distance Kernel

Zheming Jin(✉), Hal Finkel, Kazutomo Yoshii, and Franck Cappello

Argonne National Laboratory, Argonne, IL 60439, USA
{zjin, hfinkel, kazutomo, cappello}@anl.gov

Abstract. Heterogeneous platforms provide a promising solution for high-performance and energy-efficient computing applications. This paper presents our research on usage of heterogeneous platform for a floating-point intensive kernel. We first introduce the floating-point intensive kernel from the geographical information system. Then we analyze the FPGA designs generated by the Intel FPGA SDK for OpenCL, and evaluate the kernel performance and the floating-point error rate of the FPGA designs. Finally, we compare the performance and energy efficiency of the kernel implementations on the Arria 10 FPGA, Intel's Xeon Phi Knights Landing CPU, and NVIDIA's Kepler GPU. Our evaluation shows the energy efficiency of the single-precision kernel on the FPGA is 1.35X better than on the CPU and the GPU, while the energy efficiency of the double-precision kernel on the FPGA is 1.36X and 1.72X less than the CPU and GPU, respectively.

Keywords: HPC · FPGA · Floating-point operation · OpenCL

1 Introduction

Compared to central processing units (CPUs) and graphics processing units (GPUs), field programmable gate arrays (FPGAs) have major advantages in reconfigurability and performance achieved per watt. This development flow has been augmented with high-level synthesis (HLS) flow that can convert programs written in a high-level programming language to Hardware Description Language (HDL) [1]. Using high-level programming languages such as C, C++, and OpenCL for FPGA-based development could allow regular software developers, who have little FPGA knowledge, to take advantage of the FPGA-based application acceleration.

OpenCL is an open-source standard for data-parallel heterogeneous computing, which supports CPUs, GPUs, FPGAs, and other accelerators. OpenCL specifies functionality that vendors need to implement for their hardware features and programming interfaces. In addition, OpenCL makes it easier for a portable design across multiple hardware platforms and allows developers to optimize the functions for a specific architecture.

The Intel FPGA SDK for OpenCL supports their Cyclone-, Stratix-, and Arria-series FPGA platforms [2–4]. Xilinx offers a complete SDAccel development

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 664–675, 2018.
https://doi.org/10.1007/978-3-319-75178-8_53

environment for OpenCL-based application acceleration on their Kintex-series and Virtex-7 FPGA products [5].

Recent publications [6–9] on optimizing OpenCL applications on FPGAs show that there are few detailed analyses of the mapping of various floating-point operations to FPGAs for a floating-point intensive kernel. The analysis and evaluation of mapping floating-point operations described in a high-level programming language to hardware are important because a user can optimize a design that enables the compiler to reduce FPGA resource usage and increase performance.

To this end, this paper presents our research on the evaluation of a floating-point intensive kernel compiled with the Intel FPGA SDK for OpenCL employing the Nallatech 385A FPGA board. The analyses of this kernel reveal how the compiler optimizes the single- and double-precision kernels and maps each floating-point arithmetic operation in the kernel to the corresponding hardware floating-point operator.

The kernel is representative of other floating-point intensive kernels. As far as the authors know, it has not been evaluated previously on the FPGA-based computing platform. In this paper, we first introduce the kernel identified in a geographical information system (GIS) and analyze the FPGA designs generated by the compiler. Then we measure the kernel execution time and the floating-point error rate of the FPGA implementations. Finally, we compare the performance and energy efficiency of the kernels on the Arria 10 FPGA, the Intel Xeon Phi Knights Landing CPU, and the NVIDIA's K80 GPU.

2 Background

As a brief overview of the OpenCL programming model, an OpenCL application consists of host and kernel programs. Its host program is written in standard C/C++ that runs on most modern microprocessors. The host allocates data arrays in the global memory that will be read by the kernel. When the data are ready for the kernel, the host launches the kernel that will be executed on the FPGA device(s). A kernel typically executes computation by reading data from global memory as specified by the host, processing it, and then writing the results back into global memory. When the results are ready, they can be read by the host for validation and post-processing.

Intel and Xilinx websites provide OpenCL literature on implementation, low-level optimization, and programming interfaces for their hardware features. In many cases, an optimized kernel with loop unrolling, vectorization, and compute-unit duplication can achieve better performance on FPGAs, but the resource usage of the resulting implementations limits the degree of task and data parallelism. In addition, the modules in the low-level kernel system architecture – including the memory access interface, local memory usage, work-group dispatch, and the interconnection network – affect kernel performance.

3 Related Work

Underwood showed that the use of FPGAs is promising for running applications with floating-point addition, multiplication and division [10]. Since then, FPGAs have been gradually decreasing the gap to GPUs and many-core CPUs for particular applications in terms of peak performance, power consumption, and sustained performance. [11].

In [12], the authors showed that the performance of the double-precision floating-point matrix multiplication on FPGAs has a 3.48X improvement over that of the processor, while the power per GFLOP of the FPGA is 7.64X lower than that of the processor. In addition, the FPGA slices of the 64-bit floating-point addition unit and multiplication unit is on average 2.5X and 3.1X more than those of the 32-bit floating-point units, respectively. Due to the FPGA size constraint, the authors only studied the floating-point add and multiply units.

In [13], the authors presented application characteristics to FPGA, CPU, and GPU platform mapping using three applications. For their future work, they suggested a direct comparison between CUDA and a high-level language for FPGAs.

In [6], the authors demonstrated that the OpenCL-based FPGA implementation of a fractal encoding kernel is 3X and 114X faster than a GPU and a multi-core CPU, respectively, while consuming 12% and 19% of the power, respectively. They compared the results on Altera Stratix IV 530 and Stratix V A7 FPGAs with a NVIDIA Fermi C2075, a 40 nm GPU; and an Intel Xeon W3690 host processor, a 32 nm CPU. Our FPGA results on the Arria 10 GX1150 are compared against the NVIDIA K80, a 28 nm GPU; and an Intel Xeon Phi Knights Landing 7210, a 14 nm CPU. This takes into account technological advances in the hardware platforms.

In [9], the authors implemented the Monte Carlo simulations option pricing with three HLS tools from Altera, Xilinx, and Maxeler, and compared the results among FPGA, CPU, and GPU accelerator platforms. Their results showed that the HLS tools are suited to accelerating parallel-friendly algorithms. The study, however, didn't analyze how floating-point operators in the kernel are implemented on each FPGA board.

The OpenCL kernels in the CHO benchmark [14] contain implementations of IEEE-standard double-precision floating-point operations using 64-bit integers, but none of the kernels have floating-point computations. For a subset of the OpenCL-based Rodinia benchmark suite, the authors achieved 3.4X greater energy efficiency using a Stratix V FPGA in comparison to a NVIDIA K20c GPU [8]. Due to the compiler and board support package issues for their Arria-10 FPGA board at the time, the results may not reflect the best performance for each kernel.

A key to efficient FPGA implementation for complicated floating-point operations is to use multiplier-based algorithms to leverage the large amount of hardened DSP resources integrated into the FPGA devices [15]. For example, Arria 10 FPGAs—Intel's first FPGAs that natively support single-precision floating-point computation using dedicated hardened circuitry—delivers 3.8X increased performance and 3.6X better energy efficiency than the Stratix V results for the SGEMM kernel [16].

When implementing real-word large floating-point functions on an FPGA, a general rule of thumb is that the clock speed of a design implementation would degrade as

```
void
geodesic_distance (TYPE * restrict lat1,
                   TYPE * restrict lon1,
                   TYPE * restrict lat2,
                   TYPE * restrict lon2,
                   TYPE * restrict out)
{  ---- BB0 ----
   i = get_global_id(0) ;  // return work-item ID
   rad_lon1 = lon1[i] * TO_RADIAN ;
   rad_lat1 = lat1[i] * TO_RADIAN ;
   rad_lon2 = lon2[i] * TO_RADIAN ;
   rad_lat2 = lat2[i] * TO_RADIAN ;

   tu1 = COMPRESSION_FACTOR * sin ( rad_lat1 ) /
                              cos ( rad_lat1 ) ;
   tu2 = COMPRESSION_FACTOR * sin ( rad_lat2 ) /
                              cos ( rad_lat2 ) ;

   cu1 = 1.0 / sqrt ( tu1 * tu1 + 1.0 ) ;
   su1 = cu1 * tu1 ;
   cu2 = 1.0 / sqrt ( tu2 * tu2 + 1.0 ) ;
   s = cu1 * cu2 ;
   baz = s * tu2 ;
   faz = baz * tu1 ;
   x = rad_lon2 - rad_lon1 ;
   ---- BB1 ----
   do  {
     sx = sin ( x ) ;
     cx = cos ( x ) ;
     tu1 = cu2 * sx ;
     tu2 = baz - su1 * cu2 * cx ;
     sy = sqrt ( tu1 * tu1 + tu2 * tu2 ) ;
     cy = s * cx + faz ;
     y = atan2 ( sy, cy ) ;
     sa = s * sx / sy ;
     c2a = - sa * sa + 1.0;
     cz = faz + faz ;
     if ( c2a > 0.0 ) cz = -cz / c2a + cy ;
     e = cz * cz * 2.0 - 1.0 ;
     c = ( ( -3.0 * c2a + 4.0 ) * FLATTENING + 4.0 ) * c2a *
        FLATTENING / 16.0 ;
     d = x ;
     x = ( ( e * cy * c + cz ) * sy * c + y ) * sa ;
     x = ( 1.0 - c ) * x * FLATTENING + rad_lon2 - rad_lon1 ;
   } while ( fabs ( d - x ) > EPS ) ;
   ---- BB2 ----
   x = sqrt ( ELLIPSOIDAL * c2a + 1.0 ) + 1.0 ;
   x = ( x - 2.0 ) / x ;
   c = 1.0 - x ;
   c = ( x * x / 4.0 + 1.0 ) / c ;
   d = ( 0.375 * x * x - 1.0 ) * x ;
   x = e * cy ;
   s = 1.0 - e - e ;
   s = ( ( ( ( sy * sy * 4.0 - 3.0 ) * s * cz * d / 6.0 - x ) *
         d / 4.0 + cz ) * sy * d + y ) * c * POLAR_RADIUS ;
   out[i] = s;
}
```

Fig. 1. Pseudocodes for the geodesic distance kernel.

FPGA resource utilization rises above 70–80%. This high-resource utilization often requires more effort spent on placement, routing, and timing optimization. Intel FPGA SDK for OpenCL version 16.0.2 Pro Prime, for example, fails to generate FPGA implementations for two kernels in the CHO benchmark due to routing congestion [17]. In addition, floating-point results generally do not strictly match across different heterogeneous computing platforms. For example, Leeser et al. give an example of the numerical accuracy difference in the sequential and parallel versions of a floating-point intensive program [18] when analyzing the behavior of an OpenCL floating-point benchmark on different heterogeneous architectures.

4 OpenCL Kernel Implementation

4.1 Kernel Description

The geodesic distance kernel calculates the distance between two geographic coordinates on the earth's surface. Earth's shape is modelled as an ellipsoid. The shortest distance between two points along the surface of an ellipsoid is along the geodesic. The methods for computing the geodesic distance are available in GIS, software libraries, standalone utilities, and online tools [19]. The OpenCL kernel is based on the open-source implementation [20] of the solution to the inverse geodesic problem [21].

Figure 1 presents the pseudocodes for the kernel. Each coordinate of a point is represented as latitude and longitude in degrees. The default type of the coordinate is double-precision floating-point type. The kernel is composed of three building blocks (BB0, BB1 and BB2) annotated in Fig. 1, and is floating-point intensive with more than 100 floating-point arithmetic operations.

4.2 Analyses of Kernel Implementations

The Intel FPGA SDK for OpenCL compiler generates three block modules in Verilog HDL corresponding to the three building blocks in the kernel. Table 1 shows the number of double-precision floating-point operator instances in the Verilog HDL codes generated by the compiler without any floating-point optimization options enabled. From the arithmetic expressions in the BB0, the compiler instantiates four divide operators (dp_div), two square root operators (dp_sqrt) and two combined sine and cosine operators (dp_sincos) in the HDL library of the Intel FPGA SDK for OpenCL. There are only 12 multiplications in the BB0, but the number of instantiated multipliers (dp_mul) is 13. The generated Verilog HDL code reveals that the compiler performs a global optimization to include the multiplication "su1 * cu2" from the BB1, as "su1", "cu2", and their product have no dependency with other variables in the BB1.

For the BB1, the compiler produces the expected number of operators for sincos, atan2, and sqrt operations. The compiler, however, instantiates 18 multipliers, less than the number of multiplications in the expressions. The compiler optimizes away the multiplications in "cz * 2.0" and "−3.0 * c2a". For the divide operations, the compiler instantiates two dividers and converts the "divide by 16.0" operation to an adjustment to the exponent of the result.

Table 1. Number of double-precision floating-point operators instantiated by the compiler without using floating-point optimization

Operator	BB0	BB1	BB2	Total
dp_mul	13	18	13	44
dp_div	4	2	3	9
dp_sincos	2	1	0	3
dp_atan2	0	1	0	1
dp_sqrt	2	1	1	4

Table 2. Number of double-precision floating-point operators instantiated by the compiler using option "–fpc"

Operator	BB0	BB1	BB2	Total
dp_div	4	2	3	9
dp_sincos	2	1	0	3
dp_atan2	0	1	0	1
dp_sqrt	2	1	1	4
int_mul	13	18	13	44

For the BB2, the compiler attempts to optimize away the "multiply by constant" operations and is able to factor out the common product "x * x" in the block. Therefore, the compiler instantiates 13 multiply operators. For the divide operations, the compiler does not optimize away the "x/6.0", as "x/6" is not precisely equivalent to "x * 1/6.0". Therefore, three dividers are instantiated.

Overall, the compiler instantiates 44 multiply, nine divide, three sincos, one atan2, and four square root operators. It does not instantiate other floating-point operators from the HDL IP library. Instead, they are directly implemented using combinational and sequential logics. While the compiler supports the optimization option of replacing a * b + c with a multiply-and-add (MAD) operator, a double-precision MAD operator is not available in the IP library.

The Intel FPGA OpenCL programming guide [22] describes how users can reduce the amount of floating-point hardware resources with the "–fpc" option of the compiler command. The option removes floating-point rounding options and conversions whenever possible.

Table 2 shows the number of double-precision floating-point operators of each type instantiated by the compiler when using the optimization option. The option removes intermediary roundings and conversions when possible and changes the rounding modes to round towards zero for multiply and add operations. Compared to the results in Table 1, the option directs the compiler to instantiate 44 54 $\times$ 54-bit integer multiply operators because mantissa multiplication requires a 54 $\times$ 54-bit hardware multiplier.

While another floating-point optimization option, "–fp_relaxed", can lead to more efficient hardware resource usage by relaxing the order of arithmetic floating-point operations, the FPGA resource usage report does not show resource reduction for the kernel.

For the single-precision floating-point kernel, Table 3 shows the number of operators of each type instantiated by the compiler without any floating-point optimization options enabled. The compiler instantiates multiply (sp_mul), add (sp_add), subtract (sp_sub), and compare (sp_cmp) operators from the IP library. The compiler optimizes the multiply and add operations with multadd (a * b + c) and dot2 (a * b + c * d) operators. Compared to the double-precision implementations, the compiler can generate high-performance hardened floating-point implementations by taking advantage

Table 3. Number of single-precision floating-point operators instantiated by the compiler.

Operator	BB0	BB1	BB2	Total
sp_mul	13	16	13	42
sp_add	1	4	6	11
sp_sub	1	4	4	9
sp_multadd	2	6	3	11
sp_div	4	2	3	9
sp_sincos	2	1	0	3
sp_dot2	0	1	0	1
sp_atan2	0	1	0	1
sp_sqrt	2	1	1	4
sp_cmp	0	2	0	2

of the native floating-point operators offered by Arria 10 FPGA devices [23]. The compiler, however, does not discover additional multiply and add operations using the "-cl-mad-enable" optimization flag. When the optimization option "–fpc" or "–fp-relaxed" is enabled for the single-precision floating-point kernel, the compiler may ignore the option and generate the same Verilog HDL codes.

5 Experimental Results

5.1 Experimental Setup

We chose the Intel Xeon Phi Knights Landing (KNL) 7210 processor with 64 cores and four threads per core as the target CPU, with high-bandwidth on-package memory in cache mode. The program is compiled using an Intel C compiler, version 2018 Beta, with the "-O3" option, OpenMP, and AVX-512 SIMD instruction enabled. Its system thermal design power is 215 W, and its idle CPU package power is approximately 60 W [24].

We chose the NVIDIA K80 with 2,496 cores as the target GPU. Its peak performance is 2.8 TFLOPS for double-precision, and 0.95 TFLOPS for single-precision. The GPU's power limit is 149 W with an idle power of 74.15 W with persistence mode enabled. The program is compiled with CUDA Toolkit 7.5.

We used the Intel's FPGA SDK for OpenCL version 16.0.2 Pro Prime to compile the OpenCL kernels into the hardware configuration files. The target FPGA board is a Nallatech 385A, a PCIe-based FPGA accelerator card that features an Arria 10 GX1150 FPGA device, PCIe x8 Generation 3 host interface, and two banks of 4 GB DDR3 memory. The theoretical peak floating-point performance of the Arria10 chip is 1.5 TFLOPS, and the theoretical peak memory bandwidth is approximately 34 GB/s. The FPGA board's idle power is 27.3 W.

The input test data are retrieved from Maxmind's world cities database [25] that includes city, region, country, latitude, and longitude. In our experiment, we extracted 2^{21} cities with unique locations around the world. We chose four cities (Mumbai,

Sydney, Federal District Mexico, and London) from which the kernel computed distances to each of the 2^{21} cities.

5.2 Resource Usage, Performance, and Power

Tables 4 and 5 show the FPGA resource usage of double- and single-precision implementations of the kernel respectively. Replication of the compute unit is represented as "cuX", where X indicates the replication times. The maximum frequency (Fmax) of the double-precision kernels is approximately 230 MHz. Since each compute unit requires 515 DSPs, and there are a total of 1,518 DSPs on the target device, only two duplicate kernels (cu2) can be implemented. The approximate 30% logic utilization for each kernel also constrains the number of duplicate kernels. Compared to the double-precision floating-point kernel, the single-precision version can accommodate nine duplicate compute units (cu9), as shown in Table 5. However, the Fmax decreases from 280 MHz to 212 MHz, as the number of compute units increase from one to nine.

The kernel execution time is a performance metric that measures the execution time of a kernel on an FPGA device. Figure 2 shows that the kernel execution time of a single double-precision compute unit is 198.9 ms and 196.9 ms for cu1 (without –fpc) and cu1(–fpc), respectively. For two compute units, the kernel execution time depends on the local work size. When the local work size ranges from 2^4 to 2^{20}, the kernel execution time reaches the minimum values of 100.5 ms and 103.5 ms, respectively.

For one single-precision compute unit (cu1), as shown in Fig. 3, the execution time of the kernel is 75 ms, 62% less than the execution time of the double-precision kernel. For multiple compute units, the kernel execution time also depends on the local work size. The kernel execution time reaches the minimum values of 21.1 ms for cu4 when the local work size is 2^{14}, and 13 ms for cu9 when the local work size is 2^8.

The FPGA power consumption results of the double- and single-precision floating-point kernel are shown in Figs. 4 and 5, respectively. When there is one compute unit, the power is 35.6 W and 34.7 W for the double-precision floating-point kernel and its resource-optimized version, respectively. The power of one single-precision floating-point kernel is only 30.7 W. The power increases to the

Table 4. Resource usage and maximum frequency of the double-precision kernel implementations.

	cu1	cu1 (fpc)	cu2	cu2 (fpc)
Logic utilization	36%	28%	61%	45%
Memory bits	14%	14%	22%	21%
RAM blocks	25%	25%	44%	38%
#DSPs	515	515	1030	1030
Fmax (MHz)	230	233	227	221

Table 5. Resource usage and maximum frequency of the single-precision kernel implementations.

	cu1	cu4	cu9
Logic utilization	15%	28%	49%
Memory bits	8%	12%	17%
RAM blocks	18%	35%	63%
#DSPs	160	640	1440
Fmax (MHz)	280	255	212

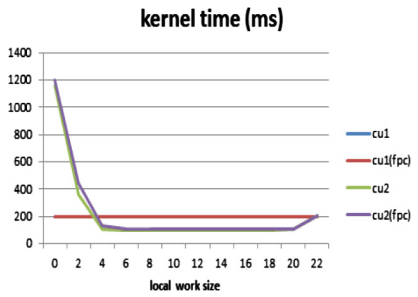

Fig. 2. Kernel execution time of the double-precision implementations. The local work size in the *x* axis indicates $2^{\text{local work size}}$.

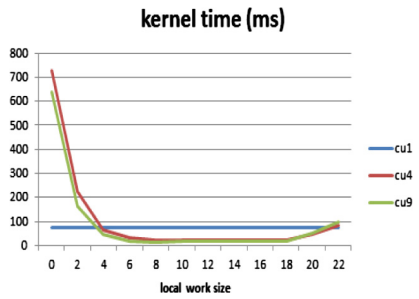

Fig. 3. Kernel execution time of the single-precision implementations. The local work size in the *x* axis indicates $2^{\text{local work size}}$.

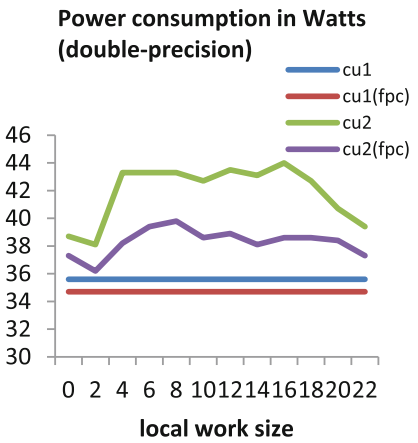

Fig. 4. Power consumption of the double-precision kernel implementations. The local work size in the *x* axis indicates $2^{\text{local work size}}$.

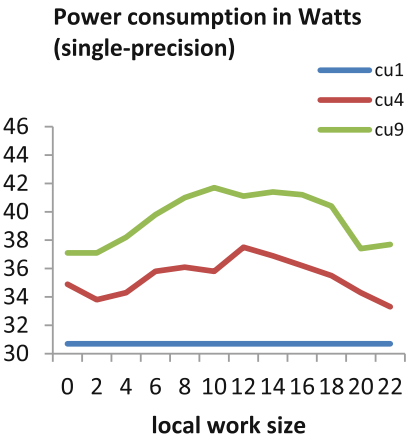

Fig. 5. Power consumption of the single-precision kernel implementations. The local work size in the *x* axis indicates $2^{\text{local work size}}$.

maximum 44 W for two double-precision compute units and a maximum of 41.7 W for nine single-precision compute units. While reducing the FPGA resource usage can effectively reduce the power, the results show that power consumption is also related to local work size for multiple compute units.

6 Comparison of CPU, GPU, and FPGA Results

In our experiment, the execution time of the kernel averages over 256 iterations. The CPU power is measured with an in-house energy trace utility, the GPU power is measured with the NVIDIA Management Library, and the FPGA power is measured

with Nallatech's board support package. For the GPU implementations, we use standard math functions instead of floating-point intrinsic functions [26]. In addition, we do not employ any floating-point optimizations provided by the CPU and GPU compilers.

As shown in Table 6, the CPU consumes the highest power (190 W), the FPGA the lowest power (44 W). Due to the DSP and logic resource constraints on the FPGA device, its execution time is more than 5X slower than the CPU and GPU for the double-precision kernel, and less than 3.25X slower for the single-precision kernel. The execution time on the GPU and CPU differ by approximately 1 ms for each kernel.

We define *energy efficiency* as the number of normalized distance calculations in millions in a second per watt:

$$\text{Energy efficiency} = \frac{\text{n}}{\text{kernel time} \times \text{maximum power} \times 1.0\text{E}6} \tag{1}$$

where n is the normalized size of the input data (i.e., a pair of double-precision coordinates equivalent to two pairs of single-precision coordinates).

As shown in Fig. 6, the GPU has the best energy efficiency (6.51) for the double-precision kernel, while the FPGA has the best energy efficiency (15.36) for the single-precision kernel. The energy efficiency of the single-precision kernel is better than that of the double-precision kernel on each platform. The energy efficiency of the single-precision kernel on the FPGA is 1.35X better than the K80 and KNL7210, while the energy efficiency of the double-precision kernel on the FPGA is 1.36X and 1.72X less than the CPU and GPU, respectively.

Table 6. Performance and energy efficiency of CPU, GPU and FPGA for the double-precision (DP) and single-precision (SP) kernels.

	CPU_{DP}	CPU_{SP}	GPU_{DP}	GPU_{SP}	$FPGA_{DP}$	$FPGA_{SP}$
Execution time (ms)	18.3	4	17.7	5.4	100.5	13
Maximum power (W)	190	190	145.5	136.7	44	42

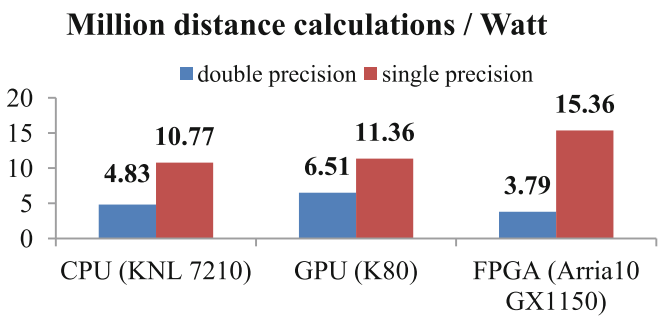

Fig. 6. Million distance calculations per watt for the single-precision and double-precision kernels on the three platforms.

7 Conclusion

We introduce the floating-point intensive geodesic distance kernel, analyze the FPGA designs generated by the compiler, and evaluate the kernel performance, resource usage, and error rate for the FPGA implementations. Two compute units can be realized for the double-precision version of the kernel on the Arria 10 GX1150, while nine can be used for the single-precision version. Single-precision floating-point computation is suitable for the current generation of FPGA devices, based on FPGA performance, resource usage, and energy efficiency of single- and double-precision floating-point kernel implementations.

In the case of the geodesic distance kernel, the energy efficiency of the single-precision kernel is 1.35X better than the GPU and CPU, while the energy efficiency of the double-precision kernel is 1.36X and 1.72X less.

The FPGA results are promising as the upcoming 14-nm Stratix 10 GX FPGA devices are power aware [27] and provide more DSPs, memory, and adaptive logic resources [28]. The GX 2800 device, for example, has 933,120 ALMs, 5,760 DSPs and 11,721 M20 memory blocks, which will allow more than double the compute units to be implemented for the kernel.

Acknowledgement. We thank the anonymous reviewers and the shepherd for their comments. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

1. Koch, D., Hannig, F., Ziener, D. (eds.): FPGAs for Software Programmers. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26408-0
2. Intel FPGA SDK for OpenCL Cyclone V SoC Getting Started Guide. Intel (2017)
3. Intel FPGA SDK for OpenCL Stratix V Network Reference Platform Porting Guide. Intel (2017)
4. Intel FPGA SDK for OpenCL Arria 10 GX FPGA Development Kit Reference Platform Porting Guide. Intel (2017)
5. Wirbel, L.: Xilinx SDAccel Whitepaper. Xilinx (2014)
6. Chen, D., Singh, D.: Fractal video compression in OpenCL: an evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In: Proceedings of 18th Asia and South Pacific Design Automation Conference, pp. 297–304 (2013)
7. Fifield, J., et al.: Optimizing OpenCL applications on Xilinx FPGA. In: Proceedings of 4th International Workshop on OpenCL. ACM, New York (2016)
8. Zohouri, H.R., et al.: Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In: International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, pp. 409–420 (2016)
9. Inggs, G., et al.: Is high level synthesis ready for business? A computational finance case study. In: 2014 International Conference on Field-Programmable Technology (FPT), Shanghai, pp. 12–19 (2014)

10. Underwood, K.: FPGAs vs. CPUs: trends in peak floating-point performance. In: Proceedings of 12th ACM International Symposium on Field-Programmable Gate Arrays, pp. 171–180. ACM Press (2004)
11. Véstias, M., Neto, H.: Trends of CPU GPU and FPGA for high-performance computing. In: 2014 24th International Conference on Field Programmable Logic and Applications, pp. 1–6 (2014)
12. Govindu, G., et al.: Area and power performance analysis of floating-point-based application on FPGAs. In: Proceedings of 7th Annual Workshop High-Performance Embedded Computing, USA (2003)
13. Che, S., et al.: Accelerating compute-intensive applications with GPUs and FPGAs. In: Symposium on Application Specific Processors, USA, pp. 101–107 (2008)
14. Ndu, G., et al.: CHO: towards a benchmark suite for OpenCL FPGA accelerators. In: 3rd IWOCL International Workshop on OpenCL, California, USA (2015)
15. Taking Advantage of Advances in FPGA Floating-Point IP Cores. Altera (2009)
16. Enabling High-Performance Floating-Point Designs. Intel (2016)
17. Jin, Z., et al.: Evaluation of CHO benchmarks on the Arria 10 FPGA using the Intel FPGA SDK for OpenCL. Argonne Leadership Computing Facility, Argonne National Laboratory, ANL/ALCF-17/4 (2017)
18. Leeser, M., et al.: OpenCL floating point software on heterogeneous architectures–portable or not. In: Workshop on Numerical Software Verification (NSV) (2012)
19. Wikipedia Webpage: https://en.wikipedia.org/wiki/Geographical_distance
20. GpsDrive Homepage: http://www.gpsdrive.de/
21. Geographiclib Homepage: https://geographiclib.sourceforge.io/2009-03/geodesic.html
22. Intel FPGA SDK for OpenCL Programming Guide. UG-OCL002. Intel (2016)
23. Arria 10 Native Floating-Point DSP IP Core User Guide. Intel (2016)
24. Jeffers, J., et al.: Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition. Morgan Kaufmann Publishers, San Francisco (2016)
25. Maxmind Database Homepage: https://www.maxmind.com/en/free-world-cities-database
26. CUDA C Programming Guide. NVIDIA (2017)
27. Leveraging the Intel HyperFlex FPGA Architecture in Intel Stratix 10 Devices to Achieve Maximum Power Reduction. Intel (2016)
28. Stratix 10 GX/SX Device Overview. Intel (2016)