High-Performance and Energy-Efficient FPGA-GPU-CPU Heterogeneous System Implementation

Chaoyu Zhang, Hexuan Yu, Yuchen Zhou, and Hai Jiang

1 Introduction

In 2019, Intel introduced the oneAPI programming model which provides a comprehensive and unified portfolio of developer tools that can be used across hardware and applications to take advantage of the oneAPI programming model. This latest programming model can execute on multiple target hardware platforms ranging from CPU, GPU, and FPGA [1]. Thus, selecting the most suitable hardware architecture based on the diversity and pattern of different applications to enhance power efficiency and performance needs to be deeply discussed.

FPGA accomplishes applications through hardware logic design, which is reconfigurable integrated circuit, for unique characters and advantages. FPGA is a balancing act between ASICs and general-purpose processors [2]. On the one hand, comparing with CPU or GPU, the hardware programming makes it more power efficient than general-purpose processors at the cost of lower flexibility and high complexity of software programming [3]. As the execution of applications is based on pre-designed hardware logic rather than separate instructions, FPGA can directly be connected to inputs and can offer very high bandwidth with lower latency. On the other hand, the reconfigurability provides more flexibility than ASICs and lower developmental cost as well as shorter periods [4].

FPGA architecture is composed of CLBs (configure logic blocks), programmable routing, and programmable I/O cells as shown in Fig. 1; CLBs are composed of SRAM (static random access memory) cells in the form of loop-up tables (LUTs) to implement combinational and sequential logic; programmable routing architecture

C. Zhang (✉) · H. Yu · Y. Zhou · H. Jiang
Department of Computer Science, Arkansas State University, Jonesboro, AR, USA
e-mail: chaoyu.zhang@smail.astate.edu; hexuan.yu@smail.astate.edu; yuchen.zhou@smail.astate.edu; hjiang@astate.edu

H. R. Arabnia et al. (eds.), *Advances in Parallel & Distributed Processing, and Applications*, Transactions on Computational Science and Computational Intelligence, https://doi.org/10.1007/978-3-030-69984-0_35

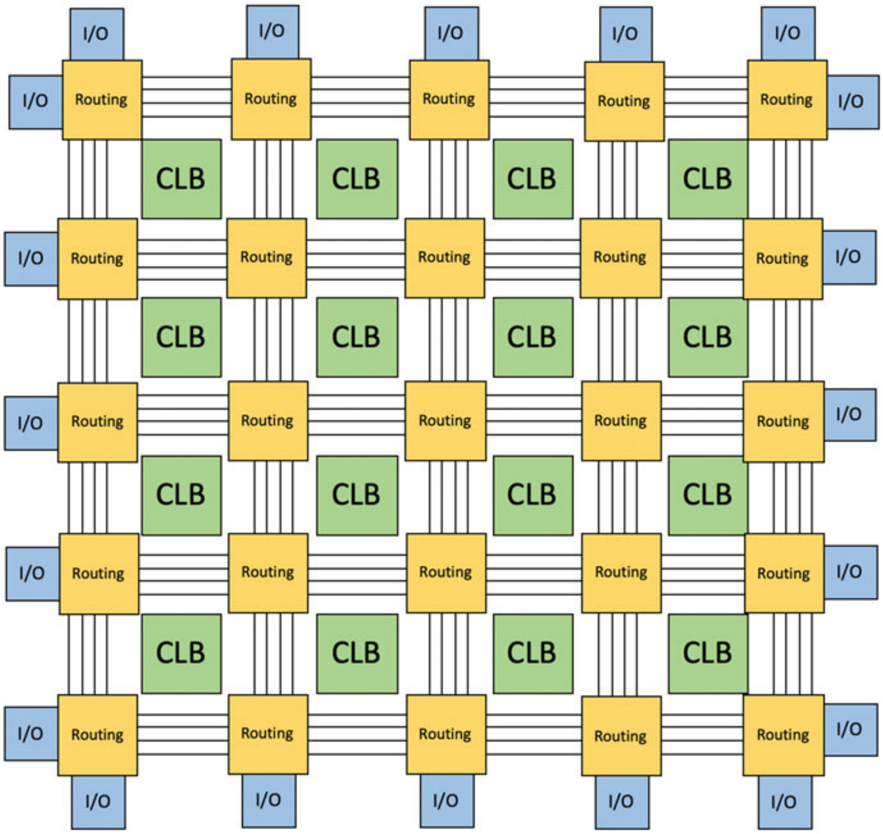


Fig. 1 FPGA high-level architecture

provides a self-defined interconnection between CLBs and also gives routing connections among logic blocks and I/O blocks to complete a fully functional circuit; and I/O cells are important components connecting the external peripherals and controlling the communications.

Thus, FPGA can be reconfigured to implement logic by reassigning the content of the LUTs and reconnecting the routing configuration. Modern FPGAs add more mature and efficient modules like DSP (digital signal processors), large memory blocks (block RAMs), and different I/O controllers (DDR, PCIe, network, etc.). These components enhance general purpose computing and save logic blocks implementing LUTs [5, 6].

GPU can offer good peak performance and high memory bandwidth. They have been widely deployed in high-performance computing (HPC) systems as accelerators. The multiprocessors employ a SIMT (single instruction multiple threads) to manage hundreds of threads. Each thread is mapped into one SP core and executes independently with its own instruction address and register state.

The Nvidia GPU has a multilevel memory hierarchy. The types of memory can be classified as global memory, shared memory, and registers. The effective bandwidth of each type of memory depends significantly on the access pattern. Global memory is relatively large but has a much higher latency compared with the on-chip shared memory. The shared memory is on-chip memory, much faster than the global memory, but we also need to avoid the problems of bank conflict and limited size. After Nvidia Kepler architecture introduces a new warp-level intrinsic called the shuffle operation, it allows the threads of a warp to exchange data with each other directly by operating registers of threads without going through shared (or global) memory [7]. The shuffle instruction has a lower latency than shared memory access and does not consume shared memory space for data exchange, so this can present an attractive way for applications to rapidly interchange data. Threads are organized in warps. A warp is defined as a group of 32 threads of consecutive thread IDs. These warps of threads are organized into thread blocks. Thread blocks are distributed among SMs and split into warps scheduled by SIMT units. All threads in the same thread block share the same shared memory and can synchronize themselves by a barrier. Threads in a warp execute one common instruction at a same time [8, 9].

In this paper, our goal is select the most suitable hardware architecture based on the diversity and patterns of different applications to enhance power efficiency and performance. With the support of OpenCL for FPGA and CUDA for GPU, we schedule different kinds of workloads on specific accelerators. Figure 2 shows the overview of an heterogeneous system consisting of different processing elements including the CPU, GPU, and FPGA. The main feature of this architecture is using current computing platforms with FPGA and GPU that communicate via PCIe within the system. Therefore, each application can be deployed onto a certain accelerator for the sake of high performance and minimized power consumption. However, choosing a proper processing element for a specific workload, with maximum performance and minimum energy consumption, needs more experiments.

To illuminate this area, we examined the performance and energy consumption of processing units such as CPU, GPU, and FPGA, with the Rodinia Benchmark Suite, version 3.1 [10] and CUDA code samples [11]. Rodinia is designed for heterogeneous computing infrastructures. A vision of heterogeneous computer systems that incorporate diverse accelerators is widely shared among researchers and many industry analysts. Programming model overview is shown in Fig. 3 (OpenCL host and OpenCL kernel). The host code is for programming a host application running on a host PC to manage an FPGA device at runtime with a set of common API (application programming interface) and is compiled using a standard C compiler to generate a host binary. The kernel code is launched to FPGA or GPU. As for the FPGA kernel, it is compiled with the Intel FPGA OpenCL compiler and converted into synthesizable Verilog HDL files. Then aocx files with FPGA configuration information are generated by Intel Quartus Prime. The aocx file is downloaded to FPGA at runtime of the application on host through APIs, and the input data required for the kernel and the output resulting data are transferred via PCIe bus [4, 12].

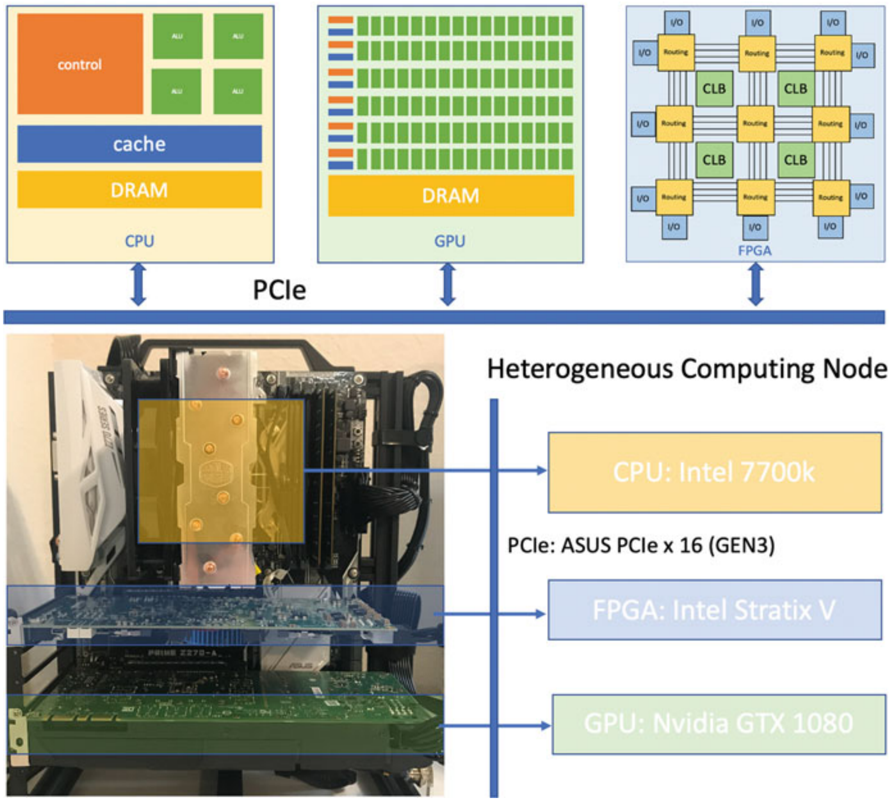
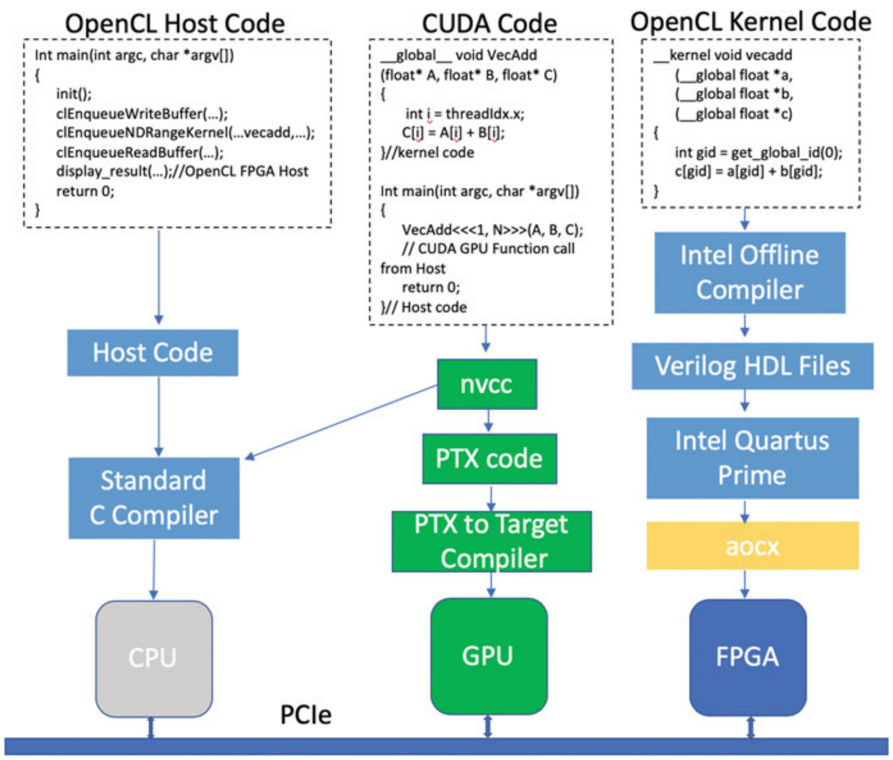


Fig. 2 Heterogeneous programming composed of CUDA and OpenCL

Most recent research focuses on a single accelerator, such as FPGA and GPU. The integration of GPU and FPGA hasn't been addressed often. Although several studies have attempted to explore this issue, related FPGA implementations are based on hardware description languages such as VHDL and Verilog at quite low-level. Therefore, task-level scheduling task characteristics over FPGA-GPU-CPU heterogeneous architecture are the biggest difference between our work and previous research.

The remainder of this paper is organized as follows: Sect. 2 gives an introduction to FPGA-GPU architectures. Section 3 describes the FPGA-GPU programming model. Section 4 introduces the FPGA-GPU benchmark kernels. Section 5 describes the FPGA-GPU heterogeneous implementation and result analysis. Finally, concluding remarks are stated in Sect. 6.



```
Int main(int argc, char *argv[])
{
    init();
    clEnqueueWriteBuffer(...);
    clEnqueueNDRangeKernel(...vecadd,...);
    clEnqueueReadBuffer(...);
    display_result(...);//OpenCL FPGA Host
    return 0;
}
```

```
__global__ void VecAdd
(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}//kernel code

Int main(int argc, char *argv[])
{
    VecAdd<<<1, N>>>(A, B, C);
    // CUDA GPU Function call
from Host
    return 0;
}// Host code
```

```
__kernel void vecadd
    (__global float *a,
    (__global float *b,
    (__global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

Fig. 3 FPGA OpenCL and GPU CUDA programming model

2 FPGA-GPU-CPU System Architectures

2.1 FPGA

Comparing FPGA with general-propose processors such as CPU or GPU, the hardware programming makes it more power efficient than general-purpose processors with a cost of lower flexibility and high complexity of software programming. Since the application execution is based on pre-designed hardware logic rather than separate instructions, FPGA can directly be connected to inputs and offer very high bandwidth with lower latency. Moreover, the reconfigurable characters provide more flexibility than ASICs with a cost of lower development investment and shorter development period. However, FPGA programming is not as simple as C programming used in general-propose processors. To create an FPGA design, the application is processed by Intel OpenCL SDK. Eventually, an FPGA bitstream is created. One of the most time-consuming processing is synthesizing hardware description into a netlist, which is just a "list of nets," connecting gates or flip-flops together. With Intel Stratix V FPGA, the total synthesis time is up to several hours, including the time to establish a large scale routing connections [13].

Intel Stratix V FPGA is capable to be deployed as an accelerator, as shown in Fig. 4. In this FPGA, the “soft-logic” consists of ALMs (Adaptive Logic Modules) with multiple CLBs, and the “hard-logic” consists of DSPs, block RAMs, multiple controllers, transceivers, and phase-locked loops (PLL). In the Stratix V FPGA, each ALM consists of multiple-input LUTs, adders and carry logic, and registers (Flip-Flops). Each adaptive LUT is capable of implementing multiple combinations of different functions including one 6-input function.

The DSP block in Intel Stratix V FPGA is designed to implement 9-bit, 18-bit, 27-bit, and 36-bit word lengths fully registered addition, multiplication, and fused multiply and add operation; one 27-bit-by-27-bit integer or fixed-point multiplication; data cascading, fast Fourier transform; and multiple several filters.

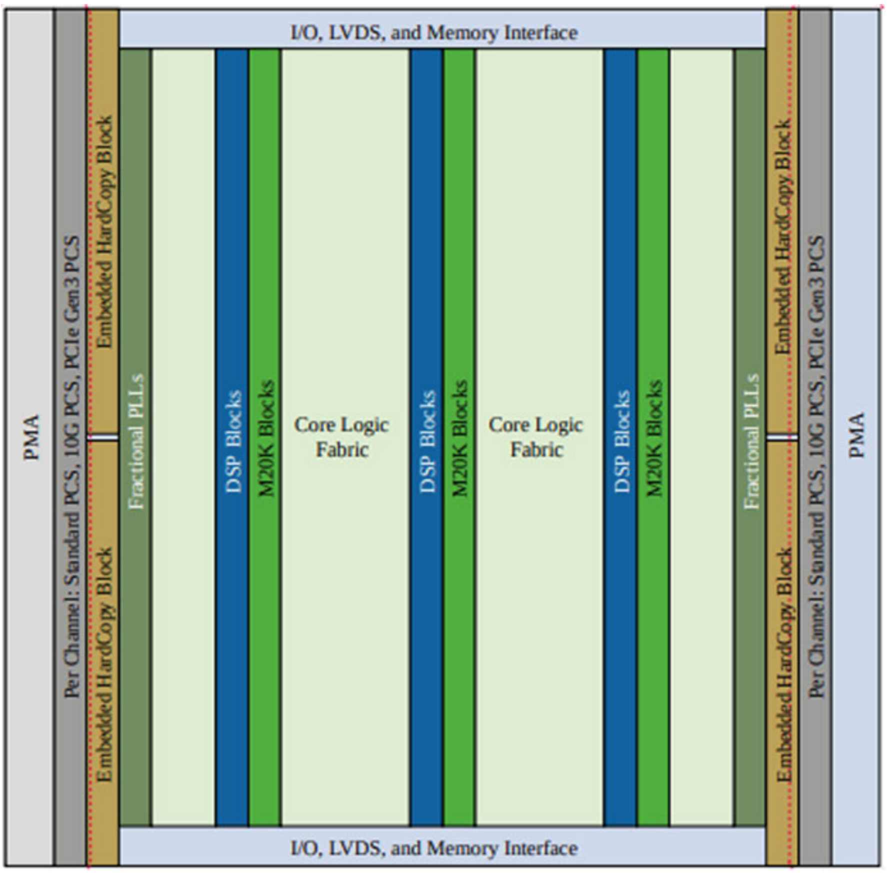


Fig. 4 Intel Stratix V FPGA architecture

Furthermore, multiple DSPs can be chained to implement dot products or other complex operations.

M20K block is RAM module in the Intel Stratix V device, capable of storing a maximum of 20 K bits of data. Each block has two independent ports for bidirectional read and write. Data can be stored in each block up to 40-bit word-length with 9-bit memory address. Furthermore, M20K is suitable to implement first-in first-out buffers (FIFO) or shift registers. Arrays of M20K blocks could be built as buffers [14].

2.2 GPU

The GPU architecture is built with an array of multi-threaded SMs (streaming multiprocessors), within each of SM composed of SP (scalar processor) cores. The multiprocessors employ a single instruction multiple threads (SIMT) model to manage hundreds of threads. Each thread is mapped into one SP core and executes independently with its own instruction address and register state.

GPU devotes more transistors to data processing. This conceptually works for highly parallel computations because GPU can hide memory access latencies with computation instead of avoiding memory access latencies through large data caches and flow control. Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations.

The Nvidia GPU has a multilevel memory hierarchy. The types of memory can be classified as global memory, shared memory, and registers. The effective bandwidth of each type of memory depends significantly on the access pattern. Global memory is relatively large but has a much higher latency compared with the on-chip shared memory. Global memory is not cached, so it is important to follow the right access pattern to achieve good memory bandwidth. Threads are organized in warps. A warp is defined as a group of 32 threads of consecutive thread IDs. A half-warp is either the first- or second-half of a warp. The most efficient way to use the global memory bandwidth is to coalesce the simultaneous memory accesses by threads in a half-warp into a single memory transaction [8]. Since it is on chip, the shared memory is much faster than the global memory, but we also need to avoid the problems of bank conflict. After Nvidia Kepler architecture introduce a new warp-level intrinsic called the shuffle operation, it allows the threads of a warp to exchange data with each other directly by operating registers of threads without going through shared (or global) memory. The shuffle instruction has a lower latency than shared memory access and does not consume shared memory space for data exchange, so this can present an attractive way for applications to rapidly interchange data.

Programming Nvidia GPU for general-purpose computing is supported by the Nvidia CUDA (Compute Unified Device Architecture) environment. CUDA programs on the host (CPU) invoke a kernel grid, which runs on the device (GPU). The same parallel kernel is executed by many threads. These threads are organized into thread blocks. Thread blocks are distributed to SMs and split into warps scheduled by SIMT units. All threads in the same thread block share the same shared memory of size 48 KB and can synchronize themselves by a barrier. Threads in a warp execute one common instruction at a time. This is referred to as warp-level synchronization. Full efficiency is achieved when all 32 threads of a warp follow the same execution path. Branch divergence causes serial execution.

3 FPGA-GPU Programming Model

In this heterogeneous computing node, we use a mixed compilation solution to cooperate different processing elements. Figure 5 shows that the flow of the compilation was implemented in multilingual programming composed of CUDA and OpenCL, and therefore separate compilation was needed. The CUDA code and OpenCL host code were compiled with nvcc and g++ separately, and the generated object files were linked using nvcc to generate an executable and linkable format (ELF) file. The OpenCL kernel code for the computation was compiled offline using the Intel FPGA OpenCL compiler [15].

3.1 Intel FPGA SDK for OpenCL

OpenCL is an open-source standard for programming heterogeneous systems. The OpenCL-based applications consisted of the host code that executes on the host CPU and can be written in compatible high-level language and C-based device kernel code. It provides APIs for controlling the accelerator and communicating

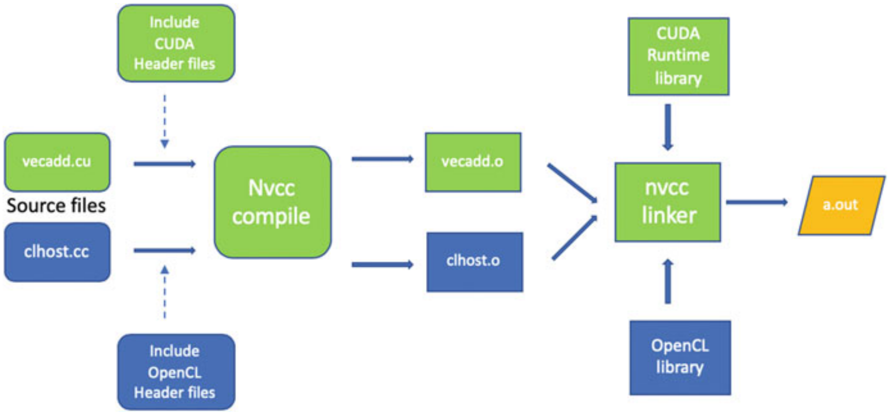


Fig. 5 Flow of heterogeneous compilation

between the host processor and the accelerator. The typical work flow of OpenCL computation is allocating host memory for input data and transferring data to the FPGA device memory by PCIe bus; once transferring is completed, FPGA kernel is launched for computation. After that, output data are sent back to the host memory [16].

In OpenCL, each thread is called a work-item and multiple work-items are grouped to form a work-group. To execute an application, the thread space is distributed over multiple work-groups. Within each work-group, work-items are synchronized using barriers, and data can be shared between the work-items using the fast on-chip local memory. However, the only way to share data between different work-groups is through the slow off-chip memory. The number of work-items in a work-group is called the local work size, and the total number of work-items necessary to fully execute an application is called the global work size. Work-items and work-groups can be arranged in multiple dimensions, up to three, in an index space called an NDRange [17].

In OpenCL, multiple memory types are defined. Global memory is inserted on the FPGA board with largest memory size but much slower. It can be accessed by all work-items of all work-groups. Global memory consistency is only guaranteed after a kernel is executed completely. Local memory is on-chip memory of the device and can be used to share data between the work-items within a work-group. Each work-group has its own local memory space, the local memory space of a work-group cannot accessed by other work-groups, and local memory consistency is guaranteed by barriers. Constant memory is an on-board and read-only memory for fast data access. Private memory belongs to each work-item as registers with very limited size [18].

Intel FPGA SDK for OpenCL provides the necessary APIs and runtime to program and use PCIe or system-on-chip (SoC) FPGA similar to GPU or other accelerators. The necessary IP cores communicate between the FPGA, external DDR memory, and PCIe, alongside with necessary PCIe and DMA drivers for communication between the host and the FPGA provided by the board manufacturers in form of a board support package (BSP). It relieves the programmer from the burden of having to manually set up the IP cores and create the drives. It is designed with traditional HDL-based FPGA designs. Some BSPs also provide the possibility to send and receive data using FPGA on-board network ports. Runtime compilation of OpenCL kernels is not possible for FPGA due to very long placement and routing time. Therefore, the OpenCL kernel needs to be compiled offline into an FPGA bitstream and then loaded at runtime by the host code to reprogram the FPGA and execute the application [15, 16].

3.2 Nvidia nvcc for CUDA

The source files for CUDA applications consist of a mixture of conventional codes. CUDA compilation separates the device functions from the host code, compiles the device functions using proprietary Nvidia compilers or assemblers, compiles the host code using a stander C compiler that is available on the host platform, and afterward embeds the compiled GPU functions as load images in the host object file. In the linking stage, specific CUDA runtime libraries are added to support remote SIMD procedure calls and provide explicit GPU manipulation and generate object files including OpenCL library for FPGA linked by nvcc to generate an executable and linkable format file.

This compilation involves several splitting, compilation, preprocessing, and merging steps for each CUDA source file. Finally, nvcc linker is used to connect object files of FPGA-GPU together. It is the purpose of the CUDA compiler driver nvcc to hide the intricate details of CUDA compilation from developers and be more convenient for programmers [8, 9].

4 FPGA-GPU Heterogeneous Kernels

Multiple benchmark suites have been proposed as representatives of HPC applications to evaluate different hardware and compilers. We can take advantage of the existing OpenMP and CUDA implementations for evaluating CPU and GPU. Furthermore, we port and optimize FPGA kernels based on Intel OpenCL SDK to figure out the meaningful performance and energy efficiency comparison between different hardware architectures and show the strengths and weaknesses of different hardware devices. We implement and analyze several famous wildly used kernels of Rodinia Benchmark Suite 3.1 and port CUDA code samples to FPGA, expecting our FPGA-GPU specific scheduling for workloads has unique advantages over CPUs or CPU-GPU system. In this study, the heterogeneous system consisted with the Intel i7-7700k CPU, the Nvidia GTX-1080 GPU, and the Intel Terasic Stratix-V GX FPGA.

The benchmarks tested on Intel FPGA and Nvidia GPU for performance and energy efficiency analysis are listed as follows:

– **MM**: Matrix multiplication is based on CUDA sample matrix multiplication, computing $C = A * B$ in parallel [11].
– **FFT**: Fast Fourier transforms (FFTs) are exploited in a wide variety of fields ranging from computer science to natural sciences and engineering. With the rising data production bandwidths of modern FFT applications, judging best which algorithmic tool to apply can be vital to any scientific endeavor [11].
– **NW**: Needleman-Wunsch is a nonlinear global optimization method for DNA sequence alignments. The potential pairs of sequences are organized in a 2D matrix [10].
– **GE**: Gaussian elimination computes result row by row, solving for all of the variables in a linear system. The algorithm must synchronize between iterations, but the values calculated in each iteration can be computed in parallel [10].

Name	Dwarfs	Area
Matrices Multiplication	Dense Linear Algebra	Linear Algebra
Fast Fourier Transforms	Spectral Methods	Digital Signal Processing
Needleman-Wunsch	Dynamic Programming	Bioinformatics
Gaussian Elimination	Dense Linear Algebra	Linear Algebra
Back Propagation	Unstructured Grid	Pattern Recognition
Speckle Reducing Anisotropic Diffusion	Structured Grid	Image Processing

Fig. 6 Summary of tested kernels

- **BP**: Backpropagation is a machine learning algorithm that trains the weights of connecting nodes on a layered neural network. The application's activations are propagated from the input to the output by layer, and the error between the observed and requested values in the output layer is propagated backward to adjust the weights and bias values. In each layer, the processing of all the nodes can be done in parallel [10].
- **SRAD**: Speckle reducing anisotropic diffusion is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). It is used to remove locally correlated noise, known as speckles, without destroying the important image features [10].

Figure 6 here is the summary of implementations.

5 Implementation and Results Analysis

We shall start our experiment with a group of performance comparison between CPU and other accelerators. Our results are the accelerator speedups (CPU running time is divided by accelerator running time) to demonstrate that no single architecture is the best for all workloads due to the incredible diversity. In Fig. 7, most of the cases except FFT show GPU has the best performance, comparing with 9.0 times speedup over CPU in general. In NW, BP, and SPRAD, the computing speedups of FPGA and GPU are basically at the same level but slightly inferior. However, in MM and GE, GPU computing speed is much faster than the other cases. One exception is FFT operation speedup on FPGA which is much higher than the rest of all hardware devices.

Due to the unique customized deep pipeline of FPGA, which successfully avoid instruction overhead in most of the other general-propose accelerators with inserted pipeline pragmas into target loops, programmers can implement pipeline in FPGA more easily. To eliminate data dependency, we need to process the data in advance

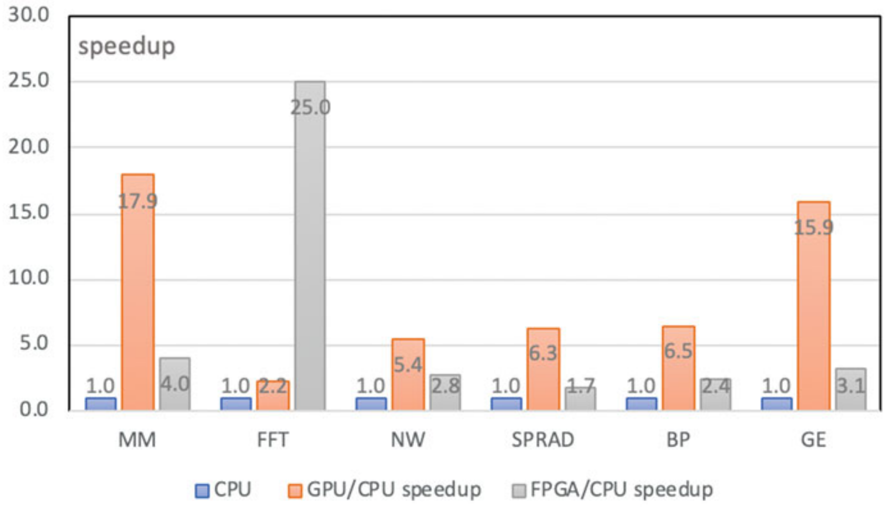


Fig. 7 Single accelerator performance speedup comparison

so that it is split into data blocks that can be calculated in parallel. Pipeline interval and depth are two key factors for measuring pipeline performance. They represent the number of cycles between the start of two consecutive iterations and the processing of the entire data iteration. The smaller the interval, the higher the pipeline throughput, the greater the depth, the higher the achievable speed [17, 18].

Another reason is that multiple programmable logic modules can perform calculations independently and simultaneously. This is similar to current multi-core and SIMD technologies. But related to SIMD technology, FPGA concurrency can be performed between different logic functions, not limited to performing the same function at the same time. That's why FFT has the best performance and dynamic programming. Unstructured grid kernels also yield speedup almost as same level as GPU.

In Fig. 8, it is obvious that FPGA has significant power efficiency advantages over any other processors. Therefore, in every benchmark, Stratix V FPGA can achieve higher power efficiency than GTX 1080 GPU as well as Intel i7 7700k CPU. The largest power efficiency advantage was observed in the GE benchmark test, that is, the power efficiency of Stratix V FPGA was 5.6 times of the same generation CPU. In MM, FPGA shows the great power efficiency amount GPU and FPGA, which is 6.3 times better than GPU. Because the compiled hardware circuit is used instead of executing instruction by instruction, FPGA has relatively stable and more efficient energy performance.

The general ratio of FPGA and GPU in computing speed as well as power efficiency is shown in Fig. 9. FPGA has more advantages in energy consumption in different examples, and GPU has better performance in a large number of parallel data calculations.

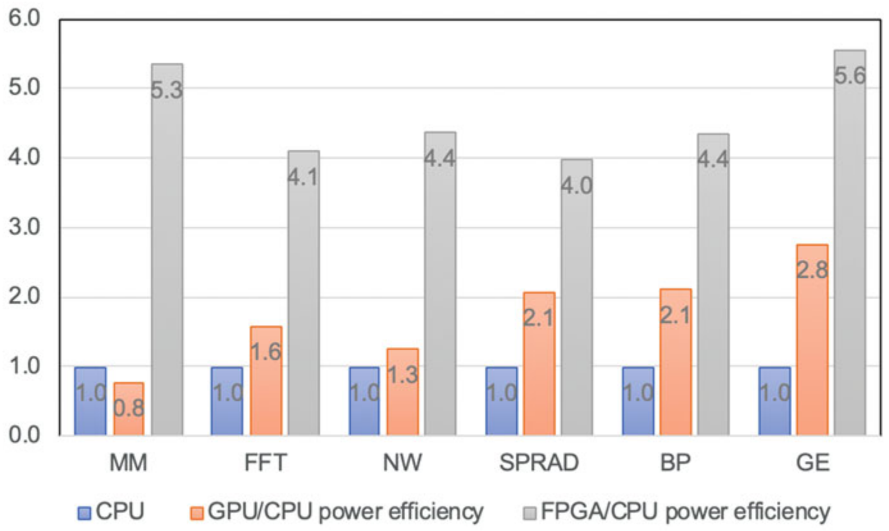


Fig. 8 Single accelerator power-efficiency comparison

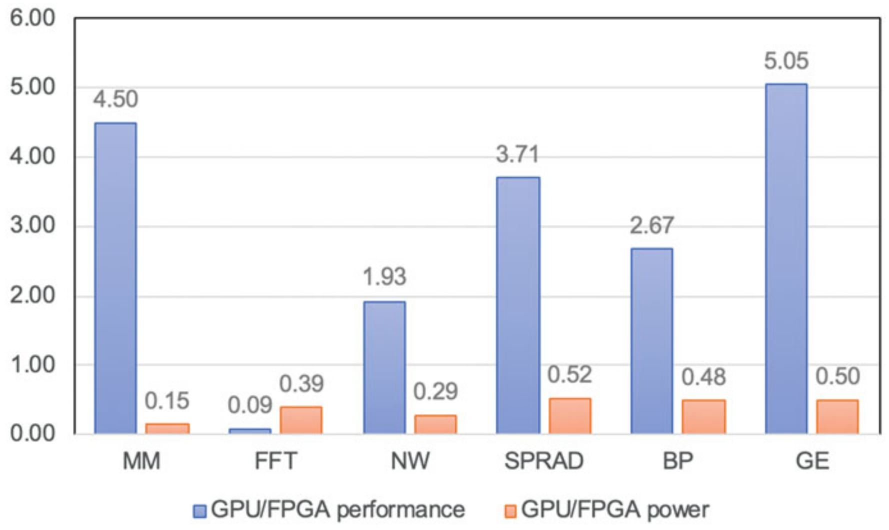


Fig. 9 Speedup and power efficiency of FPGA and GPU comparison

In order to enable different applications to achieve better computing performance or better energy efficiency on different hardware accelerators, we propose a high-performance-oriented scheduling scheme and an energy-efficient scheduling scheme. A high-performance scheduling scheme means that if this task cannot perform faster on the FPGA than the performance of the GPU or CPU, then all subsequent tasks of this type are performed on the GPU. An energy-efficient scheduling scheme means that if the computational energy efficiency of this task on the GPU cannot be better than that of the FPGA, then such tasks will be executed on the FPGA. In Fig. 10, the optimal computing performance cannot be achieved in either the single GPU or CPU or FPGA. The performance-oriented scheduler launches the FFT kernel to FPGA and the rest of the benchmarks to GPU for better performance. This could be 12.8 times faster than all running on the CPU. In our proposed energy-efficient scheduling, all benchmarks run on FPGA. Since no accelerator can achieve such energy efficiency result as FPGA does. This also proves that FPGA has huge advantages in energy efficiency, and it is 4.6 times faster than single CPU scheduling. Especially when low-latency, low power consumption, and real-time large-data operations are required, this advantage could be unique and irreplaceable. In the past few years, the slowdown of Moore's law has made heterogeneous systems to be the breakthrough that has the potential to achieve better performance and energy efficiency [19, 20]. A common heterogeneous system consists of CPU and GPU as the most widely used combo. GPU has high performance across multiple application domains with rich software ecosystem enabling programmers to adopt them without facing many programmable barriers. Compared with other accelerators, FPGA can provide better power efficiency. On the downside, developing applications on FPGA requires hardware design knowledge, which is often the main obstacle to programmers. To alleviate this problem, advanced comprehensive frameworks using languages

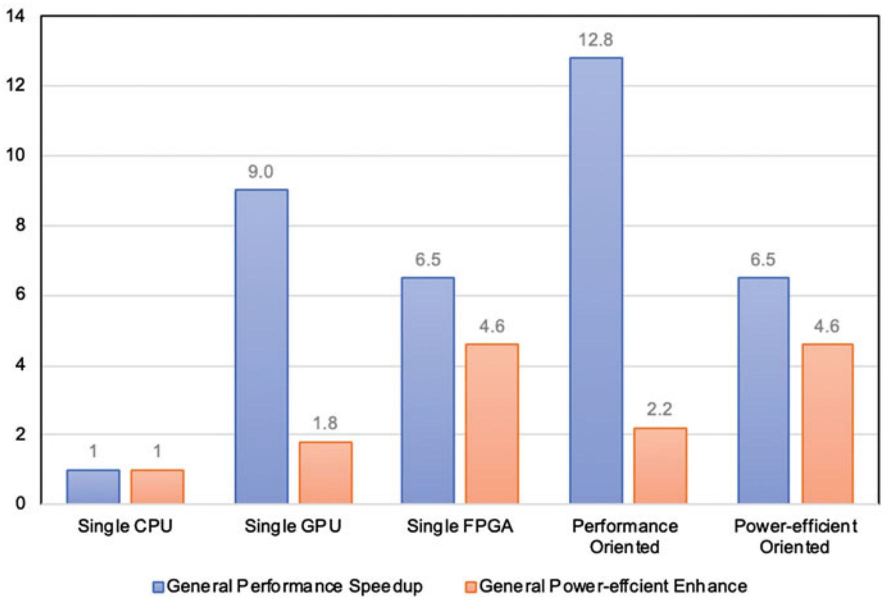


Fig. 10 General speedup and power efficiency of different scheduling

such as OpenCL have emerged to increase programmer usability. Therefore, it is far-reaching practical significance to design and explore deeper FPGA-GPU-CPU heterogeneous systems and to schedule among different hardware according to the characteristics of tasks.

6 Conclusion

In this paper, we first designed an alternative system composed of GPU, FPGA, and CPU. On top of this, we analyzed the hardware composition of the system and the different characteristics of hardware in terms of efficiency and high performance. After that, we introduced a feasible programming model that can run and schedule our assigned tasks on different hardware, choosing the most suitable hardware architecture for each application. On the FPGA-GPU-CPU system, high-performance-oriented and high-energy-efficiency-oriented working kernel scheduling are realized. This heterogeneous system can achieve better efficiency and higher performance than a single computing unit.

Intel proposed OneAPI in 2019. In order to make it easier to use heterogeneous accelerated systems across hardware, this includes CPU, GPU, and FPGA. In the future, we are likely to see supercomputers composed of these three. We hope to take this opportunity to explore the high performance and high efficiency of heterogeneous computing as an early exploration of resource management and dynamic scheduling and even some hardware virtualization in a system composed of FPGA-GPU-CPU. In summary, all in all, more diverse heterogeneous systems are worthy of our further research in order to break the slowdown of Moore's law and optimize energy efficiency.

References

1. Intel® oneAPI Programming Guide (Beta), https://software.intel.com/en-us/oneapi-programming-guide
2. H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, S. Matsuoka, Evaluating and optimizing opencl kernels for high performance computing with FPGA, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC'16* (IEEE Press, Piscataway, 2016), pp. 35:1–35:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014951
3. M.P. Véstias, H.C. Neto, Trends of CPU, GPU and FPGA for high-performance computing, in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)* (2014), pp. 1–6
4. R. Kobayashi, N. Fujita, Y. Yamaguchi, A. Nakamichi, T. Boku, GPU-FPGA heterogeneous computing with OpenCL-enabled direct memory access, in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2019)
5. I. Kuon, R. Tessier, J. Rose, FPGA Architecture (2008)

6. Alter FPGA Architecture White Paper, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf
7. CUDA C++ Programming Guide Memory Hierarchy, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-hierarchy
8. CUDA C++ Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
9. R. Li, Y. Saad, GPU-accelerated preconditioned iterative linear solvers. J. Supercomput. **63**, 443–466 (2013). https://doi.org/10.1007/s11227-012-0825-3
10. S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, K. Skadron, Rodinia: a benchmark suite for heterogeneous computing, in *IEEE International Symposium on Workload Characterization (IISWC)*, Austin (2009)
11. Samples for CUDA Developers which demonstrates features in CUDA Toolkit, https://github.com/NVIDIA/cuda-samples
12. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide, https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html
13. H.R. Zohouri, A. Podobas, S. Matsuoka, Combined spatial and temporal blocking for high-performance stencil computation on FPGA using opencl, in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA'18* (ACM, New York, 2018), pp. 153–162. [Online]. Available: https://doi.org/10.1145/3174243.3174248
14. Introducing Innovations at 28 nm to Move Beyond Moore's Law, https://www.intel.com.tw/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01125-stxv-28nm-innovation.pdf
15. Khronos OpenCL Working Group, The OpenCL Specification: Version 1.0, 10 June 2009. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf
16. H.R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, S. Matsuoka, Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City (2016)
17. Intel Corporation, Intel FPGA SDK for OpenCL: Best Practices Guide, 4 May 2018. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf
18. Intel Corporation, Intel FPGA SDK for OpenCL: Programming Guide, 14 June 2018. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf
19. D. Weller, F. Oboril, D. Lukarski, J. Becker, M. Tahoori, Energy efficient scientific computing on FPGA using OpenCL, in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA'17* (ACM, New York, 2017), pp. 247–256. [Online]. Available: https://doi.org/10.1145/3020078.3021730
20. H. Esmaeilzadeh, E. Blem, R.S. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, in *38th Annual International Symposium on Computer Architecture (ISCA)*, San Jose (2011)