*Parallel Computing: Technology Trends*
*I. Foster et al. (Eds.)*

*doi:10.3233/APC200020*

# Empowering Parallel Computing with Field Programmable Gate Arrays

Erik H. D'HOLLANDER [a,1]
[a] *Electronics and Information Systems Department, Ghent University, Belgium*

**Abstract.** After more than 30 years, reconfigurable computing has grown from a concept to a mature field of science and technology. The cornerstone of this evolution is the field programmable gate array, a building block enabling the configuration of a custom hardware architecture. The departure from static von Neumann-like architectures opens the way to eliminate the instruction overhead and to optimize the execution speed and power consumption. FPGAs now live in a growing ecosystem of development tools, enabling software programmers to map algorithms directly onto hardware. Applications abound in many directions, including data centers, IoT, AI, image processing and space exploration. The increasing success of FPGAs is largely due to an improved toolchain with solid high-level synthesis support as well as a better integration with processor and memory systems. On the other hand, long compile times and complex design exploration remain areas for improvement. In this paper we address the evolution of FPGAs towards advanced multi-functional accelerators, discuss different programming models and their HLS language implementations, as well as high-performance tuning of FPGAs integrated into a heterogeneous platform. We pinpoint fallacies and pitfalls, and identify opportunities for language enhancements and architectural refinements.

**Keywords.** FPGAs, high-level synthesis, high-performance computing, design space exploration

## 1. Introduction

Parallel Computing is predominantly focusing on high performance computing. It typically covers three major domains: architectures, algorithms and applications. These three components are considered a joint force to accelerate computations. Modern applications have an ever growing processing need. This is the case for many areas such as artificial intelligence, image processing, Internet of Things, and big data, not to mention cyber physical systems, which is the topic of one of the Horizon 2020 calls by the European Commission [14]. Despite an exponential rise of computing power for CPUs in general, we see that since 2003 the performance rate is dropping [19], first of all due to the Dennard scaling [9], which is actually a problem of warming up caused by a continuously rising clock frequency. A second aspect, not related to technology but to the type of application, is Amdahl's law [1], which is analyzed using parallel program models in section 5. The third one is the end of Moore's law [26], because transistors cannot prac-

[1] E-mail: Erik.DHollander@UGent.be.

tically be made smaller than a few silicon atoms. This means that we have to go new ways to accelerate the computations. One of the accelerators notoriously present in our today's desktops and supercomputers is the graphics processing unit. A GPU is an excellent device for parallel number crunching, originally rooted in image processing, but which since many years has outgrown this application domain by an order of magnitude. While GPUs are voracious number crunchers, graphics processing units are not a one size fits all architecture for every problem on the earth. A GPU is best suited for massively parallel computations. It has a fixed architecture consisting of many parallel processing engines, and a fast thread manager which is able to switch very quickly between threads waiting for data from the memory.

A major alternative emerging as a rising star is the field programmable gate array. In contrast to the GPU, the field programmable gate array has not a fixed but a flexible, reconfigurable architecture. An FPGA behaves metaphorically speaking like a chameleon because it is able to adapt itself to the type of the algorithm. This compute engine allows to build an ad hoc architecture which is a one-to-one mapping of the algorithm onto the hardware. There is no program, no instructions, just an interconnection of computing elements and control logic performing the task implied by the algorithm. This brings us to a very regular layout of the field programmable gate array: it consists of logic elements, computing elements such as digital signal processors, memory elements or RAM blocks, input-output elements at the border, clock signals to synchronize all the operations and finally a network of programmable interconnections used to configure the control and data paths of the design (Fig. 1).

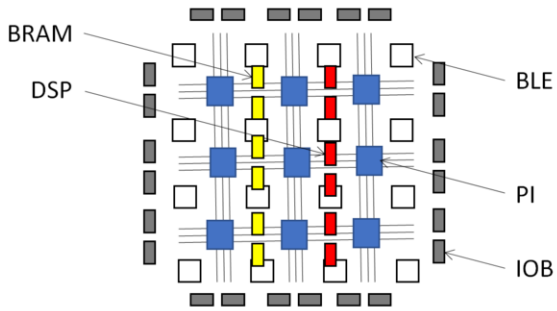


**Figure 1.** FPGA architecture with basic logic elements, programmable interconnects, I/O pins, block RAM and digital signal processors.

The so-called programming of an FPGA involves mapping a logic design, including DSPs and their interconnections, onto the physical hardware of the FPGA. This is done in two steps. First, a high-level synthesis or HLS compiler converts the program into a task graph, assigns the nodes of the graph to computing resources of the FPGA, maps the edges to control and data paths between the computing elements and generates a low-level hardware description in VHDL or Verilog. In the second step, the hardware description is mapped, placed and routed onto the physical available resources of the FPGA. The result is stored in a binary file, called the bitstream, which is uploaded to the configuration memory of the FPGA. After configuration, i.e. setting the interconnections, the programmed FPGA can operate e.g. as a data flow engine.

The rest of this paper is organized as follows. The origins of FPGAs are presented in section 2. The back-end of an FPGA compiler is described in section 3 and section 4 introduces the front end of high-level synthesis compilers, including performance factors and design exploration. The differences between FPGA and GPU accelerators are addressed in section 5, in particular focusing on the typical programming styles and application areas of both accelerators when using the same programming language OpenCL. Section 6 explores an integrated FPGA-CPU environment, including shared virtual memory, coherent caching and high bandwidth, as presented in the HARP-v2 platform. The power of this platform is illustrated by a guided image filter application. In section 7 some common misconceptions are addressed and concluding remarks are given in section 8.

## 2. History

Field programmable gate arrays have come a long way. They found their origin in the 60s when researchers were looking for new computer architectures as an alternative for the stored program concept of von Neumann. Not that the stored program concept was a bad idea, on the contrary: it is an ingenious innovation to put instructions and data in the same memory and to reconfigure the processor during instruction fetch and decode cycles. Why then look for new architectures? Well, by configuring the control and data paths at each instruction, the von Neumann computer trades in performance for flexibility.

The first person to challenge the von Neumann concept was Gerald Estrin from UCLA [12]. His idea was to extend the traditional computer, which he called a fixed architecture, with a variable part, used to implement complex logic functions. He built a six by six motherboard to hold configurable computing and memory elements. The compute elements implemented functions much more complex than the simple instructions in the fixed architecture. The backside of the motherboard consists of a wiring panel to interconnect the different components. Programming the "fix+variable" computer was quite a challenge, e.g. the computation of the eigenvalues of a symmetric matrix is described in a publication of 20 pages and resulted in a speedup of 4 with respect to an existing IBM computer [13]. Although these results are modest, they show that implementing an algorithm directly into hardware is beneficial. The question remains how this can be done efficiently.

In 1975 Jack Dennis of MIT pioneered a new architecture which he called the dataflow processor [10]. The architecture consists of a large set of computing elements which can be arbitrarily interconnected and which fire when data is present on the inputs. Programming the dataflow processor consists of interconnecting the computing elements according to the dataflow graph of the algorithm and percolating the data from the top. There are two performance gains. First, since the program is not stored in a memory, the instruction fetch and instruction decode phases are eliminated, leading to a performance gain of 20 to 40%. Second, the parallelism is maximally exploited, because it is only limited by the dependencies in the algorithm and by the available computing elements. A drawback of the first dataflow architecture is that the computing resources may be underutilized when occupied by a single data stream. Therefore Arvind, a colleague of Jack Dennis, proposed the tagged dataflow architecture [2] in which several separate data streams are identified by tokens with different colors and tokenized data compete for the compute resources in the system.

Why did these new architectures not materialize? Obviously, the technology was lagging behind, compared to the large-scale integration of today and also silicon compilers were still in their infancy. These problems have been largely overcome by the confluence and synergistic effects of hardware and software developments together with a major progress in the detection and management of parallelism in programs. Parallel computing techniques and versatile micro-electronics created a fertile ground for a new kind of architecture, the field programmable gate array. Almost simultaneously two companies were established. In 1974 Altera (now Intel) was founded by graduates of the University of Minnesota and in 1975 Xilinx was created with people from university of Illinois. Together these companies own more than 80% of the FPGA market. Nowadays, the field of FPGAs has grown into a mature hardware technology enhanced by two software developments. First, the integrated development tools now incorporate cycle accurate logic simulation, power and performance estimations and resource usage reports. Second, the support of high-level synthesis languages allows algorithms and programs to be written in C or OpenCL, and converted into low-level VHDL for implementation on the FPGA, thereby gaining several orders of magnitude in design exploration and development time. Nevertheless, programming FPGAs with high-level synthesis tools still requires a minimal notion of the underlying hardware.

## 3. Compiler back end

The smallest compute element of an FPGA, the lookup table or LUT, calculates an arbitrary logic function of a number of input variables. Several logic functions with the same inputs are calculated in parallel with the same LUT. Since the propagation delay of a LUT is much smaller than the clock cycle time of the FPGA, many LUTs can be cascaded into a long chain within the time frame of one clock cycle and thus create complex functions, such as an integer addition or multiplication. When the combined propagation delay becomes larger than the clock cycle time, the result is put into a flip-flop. The combination of a lookup table and the flip-flop is called a basic logic element. This element can be configured either as a logic function or as a state element, depending on the use of the flip-flop.

A high-level synthesis compiler converts a high-level program into LUTs, DSPs, gates, and their interconnections. Consider a sequence of functions operating on a data stream, $y = f_1(f_2(\ldots f_n(x)))$. Each of the functions is synthesized such that the combinational execution time is less than one clock cycle time and the result is stored in a flip-flop where it is ready for the next function. This creates a pipeline with a latency equal to the number of functions and a throughput of one result per cycle, as soon as the pipeline is filled with data. How is a pipeline created by the compiler? Let us look at a simple example, $y = ax + b$, represented by the task graph in figure 2. Assuming that the multiplication and the addition require one clock cycle each, then the compiler inserts registers or flip-flops to hold the intermediate results. The computation takes three clock cycles, one to read the input elements, one for the multiplication and one for the addition. Let us now extend this operation for arrays of $n$ elements, $y[i] = a[i]x[i] + b[i]; \quad i = 0 \ldots n$ as described in a loop statement. In this case, the compiler creates extra control logic to fetch the input variables, launch the computation and store the output.

The output of an HLS compiler is a description of the data- and control paths, memory, registers, flip-flops, and interconnection logic, written in a low-level hardware lan-

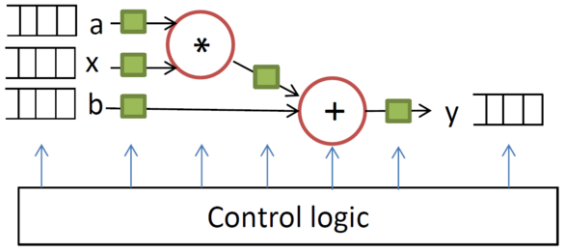


**Figure 2.** Task graph, registers (squares), I/O data streams and control logic for $y = ax + b$.

guage such as VHDL or Verilog. Next, the placement and routing of the hardware description are done by mapping, placing and routing the design onto the FPGA, taking into account the available resources. The resulting configuration of the hardware design is written into a bitstream file used to set the interconnections in the FPGA. The bitstream is moved to a persistent memory (e.g. flash memory) and from there it is stored in the configuration memory at the boot time of the FPGA. It is important to note that high-level synthesis compilers such as Vivado HLS of Xilinx or OpenCL of Intel run typically in about five minutes whereas the hardware place and route tools run in the order of five hours.

## 4. Compiler front end

Before long there were early birds attempting to create high-level synthesis compilers, mostly based on the knowledge and ideas of parallel computing developed in the 80s. The efficiency of these compilers was limited, partly due to the lack of detailed inside knowledge and proprietary information from the FPGA vendors [27]. Everything has turned around since Xilinx and Altera, now Intel, have developed high-level synthesis tools for their own devices, leading to Vivado HLS [28] by Xilinx or OpenCL [17] by Altera and Intel. Besides these imperative C-like languages, Maxeler developed a dataflow-oriented language Maxeler Java [3].

### *4.1. Design space exploration*

The approach to obtain efficient hardware with high-level synthesis is the same for OpenCL and Vivado HLS: maximize the compute power using DSPs and maximize the reuse of local data, since on-chip BRAM memory is limited. This calls for algorithms with a high arithmetic density, i.e. a large number of operations per byte I/O [8]. The first step is to adapt the algorithm to the computational model of an FPGA and carry out an initial performance evaluation. Next, pragmas are inserted to improve the performance taking into account the available resource budget of DSPs, BRAMS, and LUTs. In general, it is not required to know low-level programming or detailed characteristics of the FPGA because most information is available in the compiler reports. Further refining can be done by simulation, emulation, and profiling. The interactive phase of generating an efficient design is called the design space exploration.

### 4.2. Performance factors

Key performance factors during the design space exploration are:

- *clock frequency.* A small clock cycle time limits the amount of work that can be done in one cycle and therefore more flip-flops and cycles will be needed to implement the design. This has an impact on speed and resource consumption. An HLS compiler will optimize the clock frequency, based on the value requested by the user and the capabilities of the hardware.
- *pipelining.* If the compiler is able to create pipelines with $n$ stages, the execution speed increases to $n$ times the speed of the non-pipelined version.
- *parallelism.* When an algorithm has $m$ independent data streams, the compiler can organize $m$ parallel pipelines and therefore multiply the speedup by the number of pipelines. Examples are executing pipelined dot product calculations of a matrix multiplication in parallel or using parallel streams in systolic computations [16]. The challenge here is to feed all pipelines simultaneously.

The bottom line is that the performance of an FPGA depends on two basic principles: cram as much as possible operations into a pipeline and create as many pipelines as possible which can be fed simultaneously.

### 4.3. The initiation interval

For maximum performance, pipelines need new data in each and every clock cycle. Pipeline bubbles and gaps are caused by memory or data dependencies. Memory dependencies occur when the number and width of data ports to the cache or DDR memory are insufficient to provide continuous parallel data streams to all pipelines. Data dependencies occur when computing elements have to wait more than one cycle for the results of other computing elements in order to carry out the computation. Both memory and data dependencies involve an action from the programmer. Off-chip memory dependencies due to bandwidth limitations are removed by using on-chip cache cores or reduced using fast interfaces, such as PCI-Express. Memory dependencies caused by contention for the limited number of on-chip BRAM memory ports are avoided by partitioning data over many parallel accessible memory banks. Data dependencies occur between loop iterations when the result of one iteration is used in a subsequent iteration. The impact of loop carried dependencies is expressed by the initiation interval, $II$. The initiation interval of a loop is the number of cycles one has to wait before a new iteration can start. Ideally, the initiation interval $II = 1$. However, due to data or memory dependencies, the initiation interval may become larger than 1 for example, $II = 2$. In that case, the performance drops by 50%. The initiation interval is therefore one of the most important loop characteristics shown in the compiler reports.

There are a number of hints to improve loop operations: loops are unrolled to increase the number of parallel iterations and generate more opportunities for pipelining. Complete unrolling may lead to an overuse of resources, requiring partial loop unrolling. The bandwidth usage is improved by coalescing load operations, leading to single wide data fetches instead of doing sequential loads. Memory locations unknown at compile time, caused by pointer arithmetic or complex index calculations, are an area of concern. In nested loops it is better to represent matrices as multidimensional arrays instead of using a computed index into a single array.

To illustrate the impact of design space exploration, consider the program in listing 1. A constant is added to a matrix in a double nested loop, l1 and l2. The pragmas of the Vivado HLS compiler used are unroll, pipeline and array partitioning. The pragmas are applied selectively according to the scenarios in table 1. Unrolling the outer loop gives no speedup. Unrolling the inner loop creates 512 parallel iterations, generating a modest 6-fold speedup. The reason is that FPGA memory banks have 2 ports, therefore we can only read two values per cycle, i.e. start 2 iterations in parallel in each cycle. The inner loop takes 512/2 = 256 cycles, whereas the same loop without pragmas takes 512*3 = 1536 cycles (3 cycles for respectively read, add and write). The solution is to increase the number of memory banks by partitioning the arrays `din` and `dout`. This scenario leads to a better speedup of 512, but still leaves an initiation interval of $II = 3$ in the outer loop, because the parallel iterations of the inner loop don't overlap and therefore an inner loop iteration takes 3 cycles. Pipelining the outer loop implies unrolling the inner loop according to the compiler documentation. Without partitioning, we again obtain a speedup of 6, due to the memory bottleneck. Pipelining the inner loop, plus array partitioning, creates overlapping iterations, an initiation interval $II = 1$ and a speedup of 1529. The same happens if we pipeline the outer loop, which implies unrolling and pipelining the inner loop completely.

**Listing 1** Design space exploration, optimal case, II=1.

```
#define N 512
void nestedloop(int din[N][N], int dout[N][N]) {
#pragma HLS ARRAY_PARTITION variable=din  factor=256 dim=2
#pragma HLS ARRAY_PARTITION variable=dout factor=256 dim=2
l1:  for (int i = 0; i < N; i++) {
#pragma HLS PIPELINE
l2:      for (int j = 0; j < N; j++) {
            dout[i][j] = din[i][j] + 40;
         }
     }
  return;
}
```

| Scenario | Outer (l1) | Inner (l2) | Array partitioning | II | Speedup |
|---|---|---|---|---|---|
| Unroll | x | | | | 1 |
| Unroll | | x | | 256 | 6 |
| Unroll | | x | x | 3 | 512 |
| Pipeline | x | | | 256 | 6 |
| Pipeline | | x | x | 1 | 1,529 |
| Pipeline | x | | x | 1 | 1,529 |

**Table 1.** Design space exploration using unrolling, pipelining and array partitioning.

This simple design space exploration shows a performance improvement of more than 1500 with a number of well-chosen pragmas.

## 5. FPGA vs GPU programming: pipelining vs parallelism

Since more than two decades the performance gain of processors is hampered by clock rate and transistor scaling constraints. In accelerators such as GPUs, low clock rates have been replaced by explicit parallelism, yielding chips with hundreds or even thousands cores per chip die [21]. The exponential growth of parallel cores requires a proportional scaling of the problem parallelism. Recently, Hennessy and Patterson have illustrated that the law of Amdahl comes into play for the diminishing processing speed between the years 2011 and 2015 [19]. In this period GPUs were growing at a fast pace, and it reminds us that the basic power of GPUs is not always applicable in every algorithm. Amdahl's law specifies that we cannot diminish the execution time below the critical execution path of serial calculations, even if everything else is parallelized. This has two consequences for HPC programming: 1) the amount of serial computations in a program needs to be minimized and 2) highly parallel computers or GPUs require applications with ample parallelism. The first consequence is visible in figure 3, displaying the maximum speedup with $p$ processors as a function of the critical serial part. Even with only 2% serial operations, the speedup is limited by 50, irrespective of the number of processors used, see figure 3. For the impact of the second consequence, we look at the distribution of the parallelism in a program.

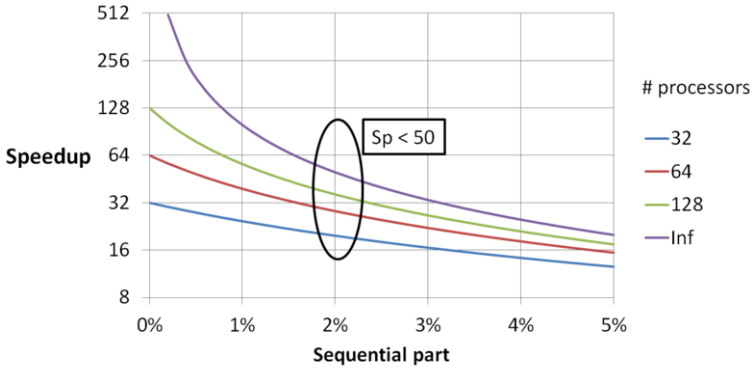


**Figure 3.** Application dependent speedup limit, Amdahl's law.

### 5.1. Distribution of parallelism in ordinary programs

Ideally, all $p$ processors of a multiprocessor work all the time, so the parallelism is $p$. Ruby Lee studied three other distributions [23]: the equal time, equal work and inverse work hypothesis. Equal time means that the program executes an equal amount of time with respectively 1, 2 up to $p$ processors operating in parallel. Equal work means that the amount of work executed with $i$ processors is proportional to $i, i = 1 \ldots p$. The inverse work hypothesis means that the amount of work done by $i$ processors in parallel is inversely proportional to $i$. These hypotheses give rise to three analytical formulas for the speedup, respectively $S_{et} = O(p)$ for the equal time distribution, $S_{ew} = O(p/\ln(p))$ for the equal work distribution and $S_{iw} = O(\ln(p))$ for the inverse work distribution, with

*et* = equal time, *ew* = equal work and *iw* = inverse work hypothesis. As an example, a workload of 48 units executed on 4 processors has a minimal execution time of 12 units. Equal time, equal work and inverse work distributions of the parallelism increase the execution time to respectively 19.2, 25.0 and 32.8 units. Ruby Lee also analyzed the parallelism of a large number of programs using the Paraphrase compiler at Illinois [22,23]. The results are shown in table 2.

| Available processors | Speedup $S_p$ | Hypothesis |
|---|---|---|
| 1–10 | $O(p)$ | Equal time |
| 11–1000 | $O(p/\ln(p))$ | Equal work |
| 1001–10000 | $O(\ln(p)) \ldots O(p/\ln(p))$ | Transient |
| >10000 | $O(\ln(p))$ | Inverse work |

**Table 2.** Speedup bounds for applications with diminshing parallelism.

In this study, the exploitable parallelism diminishes with a growing number of available processors. Although this is an old empirical observation, it may reflect that Amdahl's law is actually a law of diminishing parallelism. Since GPUs are largely dependent on massive parallelism this may also explain the decreasing rise of the computing speed in the years 2011-2015 due to Amdahl's law.

### *5.2. The case of OpenCL*

Partly due to the success of OpenCL for programming GPUs, FPGA vendors Altera/Intel and also Xilinx have selected this language for high-level synthesis [7]. However, GPUs and FPGAs are fundamentally different and this is reflected in the way an OpenCL compiler for FPGAs is designed and used.

GPUs have ample parallel cores called streaming multi-processors. These are most useful for SIMD calculations, which are launched in OpenCL by the NDRange kernel. An NDRange kernel executes the same iteration on parallel processors for each index in the *n*-dimensional iteration space of a nested loop. In a GPU architecture, the index points of a parallel loop are organized into groups of 32 threads, each executing the same instruction of the iteration for different index values [24]. Such a group of 32 threads is called a warp. All warps compete for execution and the warp scheduler assigns a SIMD instruction to a warp for which all data are available. This means that a GPU operates strictly in a single instruction multiple data or SIMD mode. When the data for a warp are not available, the warp scheduler assigns another warp for which data are ready. The fast thread switching allows to hide the memory latency of the waiting threads.

| GPU | FPGA |
|---|---|
| Fast warp (thread) scheduler | Fixed configuration, no thread switching |
| Independent iterations | Loop carried dependencies OK |
| Massively parallel (SIMD) | Pipelined execution (MISD) |
| Large memory | Small memory footprint |
| Send $\rightarrow$ Calculate $\rightarrow$ Receive | Streaming data, comp./comm. overlap |

**Table 3.** Operational differences between GPU and FPGA.

When implementing this mode of operation on an FPGA, we are faced with a number of discrepancies and operational differences, see table 3. First, an FPGA has no warp scheduler, since the design of a program is fixed in the configuration memory of the FPGA. This precludes thread switching to hide the memory latency, one of the major performance factors in GPUs. As a consequence, all data for the warp need to be available at all times, creating an extra hurdle since FPGAs usually have a small memory footprint and no cache on chip. Second, all threads in a warp operate independently and therefore require parallel loops without loop carried dependencies between iterations. This precludes the generation of long pipelines, which are favorable for execution on FPGAs. Third, since a GPU has its own hierarchy of memories and caches, the data is moved to the GPU, processed, and the results are sent back to the CPU. In contrast, FPGAs are best fit for long streams of data that are processed using overlapping computation and buffered communication. It becomes clear that FPGAs and GPUs are not so much competitors, but have a complementary role when it comes to different application domains.

### *5.3. Vector types in OpenCL*

As an alternative to the resource-hungry NDRange loop control, OpenCL for FPGAs supports the more efficient vector data types. Vector types allow to operate in a SIMD fashion on vector data. This creates parallel pipelines, thereby multiplying the pipeline performance by the number of parallel data streams operating in lockstep. An example is the use of a vector type in the matrix multiplication $C = A \times B$ by specifying the rows of matrix $B$ as `float8` vectors. In this way, 8 elements of the product matrix are calculated simultaneously within the pipelined inner loop (see listing 2). This results in 8 new values per cycle.

Listing 2: Vectored pipelines. Matrix B has data type float8. Loop j creates 8 pipelines each operating in SIMD fashion on 8 vector elements in loop k.

```
#define T float8
__kernel void __attribute__((task))
matmul (__global float * restrict A,
        __global T * restrict B,__global T * restrict C)
{
  for (int i = 0; i<N; i++)
    for (int j = 0; j<N; j += 8)
    { // 8 parallel pipelines
        T vsum = 0;
#pragma unroll 8
      // generate pipelined k-loop
        for (int k = 0; k<n; k++)
            vsum += A[i*N+k] * B[k*N/8 +j/8];
        C[i*N/8+j/8] = vsum;
    }
}
```

## 6. Advancing FPGA accelerator integration

FPGAs have a limited amount of on-chip storage, much less than GPUs. Therefore, the interaction between an FPGA accelerator and the CPU is more crucial than for GPUs. FPGA connects with a data stream either using a PCI-express bus, by accessing device data directly such as with video input or via an on-chip interconnect such as the AXI bus in the Zynq. A tighter integration is beneficial, but presents a number of other problems such as using a common cache, translating the virtual addresses of the CPU into addressable locations in the FPGA and sharing virtual memory between CPU and FPGA.

### *6.1. The HARP platform*

In order to explore several design improvements, Intel has created a research platform consisting of a fast Xeon Broadwell processor and an Arria 10 FPGA, together with extra hardware for caching, and a high-speed CPU-FPGA interconnection [29]. In addition, the OpenCL language is supported for this platform [4]. The Heterogeneous Architecture Research Platform (HARP) introduces three innovations shown in figure 4. First, the local DDR RAM at the FPGA side is replaced by a transparently shared DDR memory at the CPU side. This includes IP cores translating physical FPGA addresses into virtual CPU addresses, a Quick Path Interconnect and two PCI-Express channels, providing a combined maximum bandwidth of 28 GB/s, managed by the channel steering logic. Second, the FPGA contains a fixed hardware cache of 64 KB as well as an application-specific cache, generated by the OpenCL compiler. Third, the OpenCL implementation includes coherent shared virtual memory (SVM) support between the FPGA and the CPU.

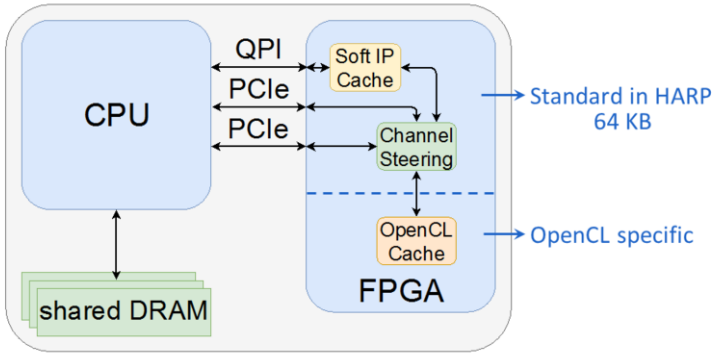


**Figure 4.** HARP architecture with coherent shared memory, fast interconnection and 2 local caches.

There are a number of questions that can be addressed: how efficient is the cache, how large is the bandwidth between the FPGA and the CPU and what is the benefit of the shared memory between the CPU and the FPGA. Regarding the bandwidth, we found that the real achieved bandwidth is between 15 and 16 GB/s, depending on the buffer size [15]. Regarding the cache, we have to make the distinction between the fixed 64 KB Soft IP cache of the FPGA Interface Unit (FIU) and a special OpenCL cache which is implemented by the compiler based on the data structures and usage in the OpenCL

program. In order to study the cache performance, we exploited the temporal locality by repeatedly reading a buffer with increasing length from the CPU. The specialized OpenCL cache was able to double the bandwidth with respect to the fixed cache in the Arria 10, see figure 5.

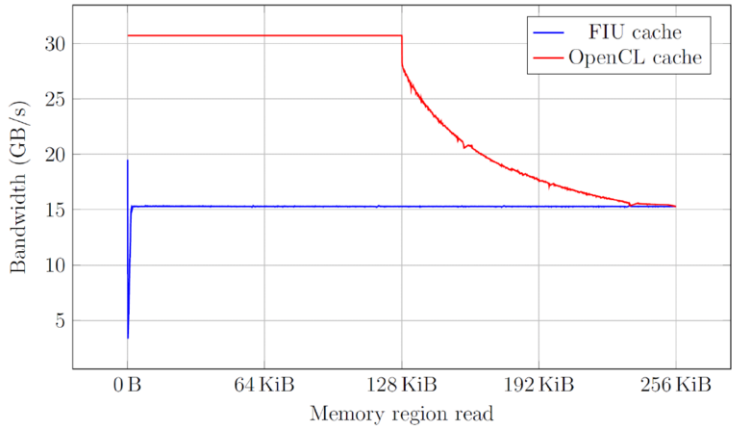


**Figure 5.** Cache efficiency: the OpenCL application-generated cache doubles the available bandwidth (red). The FIU cache was measured by disabling the OpenCL cache using the volatile keyword (blue).

Finally, in a traditional FPGA-CPU configuration, the CPU memory cannot be accessed by the FPGA, therefore OpenCL buffers are required to explicitly send and receive data, which is time-consuming. Furthermore, cache coherency is lost. In the HARP platform, a shared virtual memory (SVM) space has been implemented to transparently access data in the memory of the processor. As a result, data is sent to, or read from the FPGA on demand. This improves the communication efficiency and avoids extra buffer space in the FPGA.

### *6.2. Case study: a guided image filter*

The impact of the innovations were tested using the "guided image filter" (GIF) image processing algorithm [18]. The guided image filter takes two images: a raw noisy input image $I$ and an image with extra information $G$, for example LIDAR data, which is used to improve the noisy image. GIF is a convolution algorithm in which an output pixel $O_i$ is obtained by averaging the input pixels $I_j$ multiplied with weighted guidance pixels $W(G_j)$ in a window of radius $r$ ($r = 1,3,5...$) centered around pixel $i$. The general form is

$$O_i = \sum_j W(G_j) I_j \tag{1}$$

The algorithm uses a sliding window buffer to receive streaming input data and reuses most of the data in consecutive pixel calculations, see figure 6.

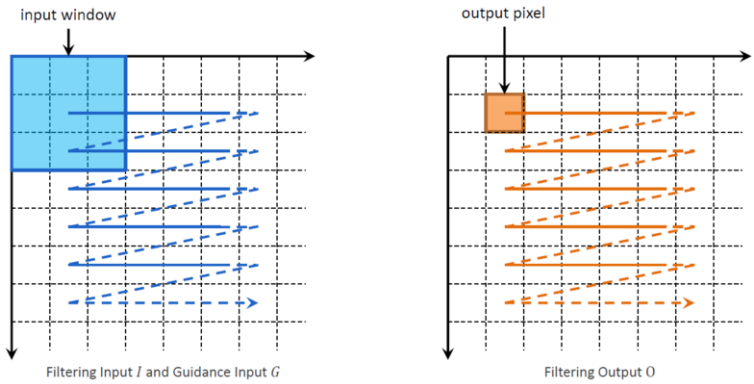


**Figure 6.** Sliding window with radius $r = 1$ in the guided filter algorithm. Windows $I$ and $G$ are used to calculate pixel $O$.

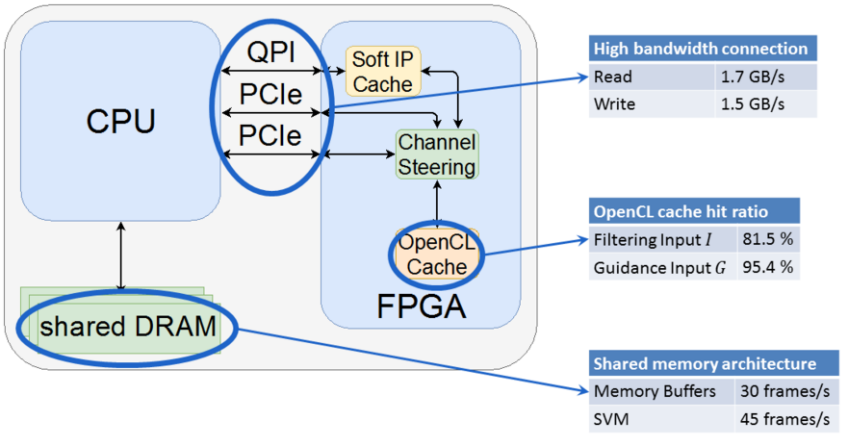


**Figure 7.** Measurements of the guided filter on the HARPv2 platform: bandwidth, OpenCL generated cache hit rate, impact of using shared virtual memory vs. copying data into local memory buffers.

### *6.3. Results*

The key performance indicators (KPIs) of this example are the bandwidth, the cache efficiency and the impact of the shared memory. The KPIs obtained in [15] are summarized in figure 7. The algorithm has been tested on full HD color images (1920x1080x3 pixels) with radius $r = 3$. The pipelined and vectorized design is compute-bound and has a data throughput of 1.7 GB/s for reads and 1.5 GB/s for writes. The OpenCL cache hit rate is 81.5% and 95.4% for the input and guidance images respectively, showing the favorable impact of an algorithm-specific designed cache. The use of shared virtual memory instead of loading data into local buffers increases the frame rate from 30 to 45 frames per second. These results are obtained using floating-point operations and correspond to the maximum design fitting on the FPGA. As mentioned in [15], a larger radius generates more computations per frame. Using fixed-point instead of floating-point calculations, a radius $r = 6$ can be implemented, yielding 74 frames per second.

This example shows that OpenCL and the HARP platform go together well. The HARP platform offers a high bandwidth and a shared memory architecture. OpenCL, on the other hand, adds a design-specific generated cache and the use of shared virtual memory to the application developer.

## 7. Fallacies and pitfalls

The world of FPGAs is exciting, but also challenging, there are a lot of high expectations, but also misconceptions. We would like to address some of these particular items or issues here.

*One kernel fits all sizes*

It is reasonable to expect that one compute kernel can be used for any size of data. This is mostly not true because the hardware generated depends on the kernel arguments. E.g. the number of rows and columns of a matrix multiplication are used to reserve local buffers, since the FPGA has no dynamic memory. Smaller matrices are fine, but one loses the unused resources. Larger matrices require other techniques such as block matrix multiplication to get the job done [11]. An FPGA kernel is thus less flexible than CPU or GPU procedures.

*Switching kernels in an FPGA or a GPU is equally fast*

This is true only if there are enough resources to store multiple kernels simultaneously in the logic fabric. Otherwise, switching between multiple kernels implies a full or partial reconfiguration for each kernel, and this takes in the order of milliseconds or even seconds [30]. In a GPU, a kernel switch is as fast as a procedure call. However, running multiple kernels simultaneously by sharing processors is more complicated and involves merging several kernels [31].

*OpenCL is a standardized language for FPGAs*

While OpenCL is a standard, its usage, attributes, pragmas and programming model are not standardized across the FPGA vendors. This applies also to the compiler reports which are used to optimize the design. E.g. the initiation interval is tabulated for each loop in the Xilinx reports whereas it is specified for basic blocks in the Intel reports. The Xilinx software gives the total number of cycles as well as the expected frequency which allows to calculate the execution time of the kernel and even the number of GFlops straight from the compiler report. Intel shows the number of cycles for each node in the task graph and each basic block. It is not obvious to calculate the total number of cycles from the task graph of the program. See also the tutorial [20].

*FPGA programming eats time*

As is mentioned in section 3, compiling an HLS program typically requires in the order of several minutes, whereas implementing the hardware design into a bitstream can take many hours. Since HLS resource usage and cycle time reporting is quite accurate and not time-hungry, HLS design space exploration shows a much more favorable development cycle time than what is generally assumed.

*An OpenCL program for a GPU is easily portable to an FPGA*

The GPU architecture differs fundamentally from the FPGA, e.g. FPGAs have a small memory and no hardware thread switching. Hardware thread reordering, proposed in [25], doubles the resource usage due to arbitration logic and extra memory management. The GPU parallel programming style using the NDrange kernel consumes a lot of resources when used in the FPGA. OpenCL kernels for FPGAs are mostly single work items, which are then pipelined. Finally, FPGAs require a thorough design space exploration to optimize resource usage or to obtain a significant speedup. For these reasons, it is often better to redesign the algorithm from the ground up to orient the program to the characteristic features and optimizations of an FPGA [5].

## 8. Conclusion

Field programmable gate arrays have become a significant player in parallel computing systems. While the support of a common OpenCL language suggests a transparent use of GPUs and FPGAs accelerators, the application domain, algorithm selection and programming style is substantially different. The initial high expectations have made way for a more realistic view and a focus on better tools and design concepts, with good results. A number of key improvements to be expected are a tight integration of FPGAs in SoCs, enhanced compiler reporting, standardized pragmas as well as increased use of HLS languages for reconfigurable computing in software engineering curricula [6].

## References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer conference - AFIPS '67 (Spring)*, page 483, Atlantic City, New Jersey, April 1967. ACM Press.

[2] Arvind, Kathail, Vinod, and Pingali, Keshav. A Dataflow Architecture with Tagged Tokens. Technical report MIT-LCS-TM-174, Massachusetts Institute of Technology, September 1980.

[3] Tobias Becker, Oskar Mencer, and Georgi Gaydadjiev. Spatial Programming with OpenSPL. In Dirk Koch, Frank Hannig, and Daniel Ziener, editors, *FPGAs for Software Programmers*, pages 81–95. Springer International Publishing, Cham, 2016.

[4] Anthony M. Cabrera and Roger D. Chamberlain. Exploring Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2. In *Proceedings of the International Workshop on OpenCL - IWOCL'19*, pages 1–10, Boston, MA, USA, 2019. ACM Press.

[5] Nicola Cadenelli, Zoran Jaksi, Jord Polo, and David Carrera. Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads. *Future Generation Computer Systems*, 94:148–159, May 2019.

[6] Luca P. Carloni, Emilio G. Cota, Giuseppe Di Guglielmo, Davide Giri, Jihye Kwon, Paolo Mantovani, Luca Piccolboni, and Michele Petracca. Teaching Heterogeneous Computing with System-Level Design Methods. In *Proceedings of the Workshop on Computer Architecture Education - WCAE'19*, pages 1–8, Phoenix, AZ, USA, 2019. ACM Press.

[7] Tomasz S. Czajkowski, David Neto, Michael Kinsner, Utku Aydonat, Jason Wong, Dmitry Denisenko, Peter Yiannacouras, John Freeman, Deshanand P. Singh, and Stephen Dean Brown. OpenCL for FPGAs: Prototyping a Compiler. In *Proceedings of the 2012 International Conference on Engineering of Reconfigurable Systems & Algorithms*, pages 3–12. CSREA Press, July 2012.

[8] Bruno da Silva, An Braeken, Erik H. D'Hollander, and Abdellah Touhafi. Performance Modeling for FPGAs: Extending the Roofline Model with High-Level Synthesis Tools. *International Journal of Reconfigurable Computing*, 2013:1–10, 2013.

[9] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974.

[10] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *ACM SIGARCH Computer Architecture News*, volume 3, pages 126–132. ACM, 1975.

[11] Erik H. D'Hollander. High-Level Synthesis Optimization for Blocked Floating-Point Matrix Multiplication. *ACM SIGARCH Computer Architecture News*, 44(4):74–79, January 2017.

[12] Gerald Estrin, Bertram Bussell, Rein Turn, and James Bibb. Parallel processing in a restructurable computer system. *IEEE Transactions on Electronic Computers*, 12(6):747–755, 1963.

[13] Gerald Estrin and C. R. Viswanathan. Organization of a "Fixed-Plus-Variable" Structure Computer for Computation of Eigenvalues and Eigenvectors of Real Symmetric Matrices. *Journal of the ACM (JACM)*, 9(1):41–60, 1962.

[14] European Commission. Computing technologies and engineering methods for cyber-physical systems of systems. RIA Research and Innovation action ICT-01-2019, Horizon 2020, October 2018.

[15] Thomas Faict, Erik H. D'Hollander, and Bart Goossens. Mapping a Guided Image Filter on the HARP Reconfigurable Architecture Using OpenCL. *Algorithms*, 12(8):149, July 2019.

[16] Xinyu Guo, Hong Wang, and Vijay Devabhaktuni. A Systolic Array-Based FPGA Parallel Architecture for the BLAST Algorithm. *ISRN Bioinformatics*, 2012:1–11, 2012.

[17] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. *Design of FPGA-Based Computing Systems with OpenCL*. Springer International Publishing AG, 2018.

[18] Kaiming He, Jian Sun, and Xiaoou Tang. Guided Image Filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(6):1397–1409, June 2013.

[19] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, January 2019.

[20] Kenter, Tobias. OpenCL design flows for Intel and Xilinx FPGAs - using common design patterns and dealing with vendor-specific differences. In *Sixth International Workshop on FPGAs for Software Programmers*, Barcelona, September 2019.

[21] Peter Kogge and John Shalf. Exascale Computing Trends: Adjusting to the" New Normal" for Computer Architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.

[22] Ruby Bei Loh Lee. Performance bounds in parallel processor organizations. In *High Speed Computer and Algorithm Organization*, pages 453–455. Academic Press, Inc, 1977.

[23] Ruby Bei-Loh Lee. *Performance characterisation of parallel processor organisations*. PhD thesis, Stanford University, Stanford, 1980.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[25] Amir Momeni, Hamed Tabkhi, Gunar Schirner, and David Kaeli. Hardware thread reordering to boost OpenCL throughput on FPGAs. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 257–264, Scottsdale, AZ, USA, October 2016. IEEE.

[26] G.E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, January 1998.

[27] A. Podobas, H. R. Zohouri, N. Maruyama, and S. Matsuoka. Evaluating high-level design strategies on FPGAs for high-performance computing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017.

[28] Moritz Schmid, Christian Schmitt, Frank Hannig, Gorker Alp Malazgirt, Nehir Sonmez, Arda Yurdakul, and Adrian Cristal. Big Data and HPC Acceleration with Vivado HLS. In Dirk Koch, Frank Hannig, and Daniel Ziener, editors, *FPGAs for Software Programmers*, pages 115–136. Springer International Publishing, Cham, 2016.

[29] Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, and Austin Baylis. Scalable Window Generation for the Intel Broadwell+Arria 10 and High-Bandwidth FPGA Systems. In *FPGA '18*, pages 173–182, Monterey, CA, 2018. ACM Press.

[30] Kizheppatt Vipin and Suhaib A Fahmy. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. *ACM Computing Surveys*, 1(1):39, January 2017.

[31] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369, March 2016.