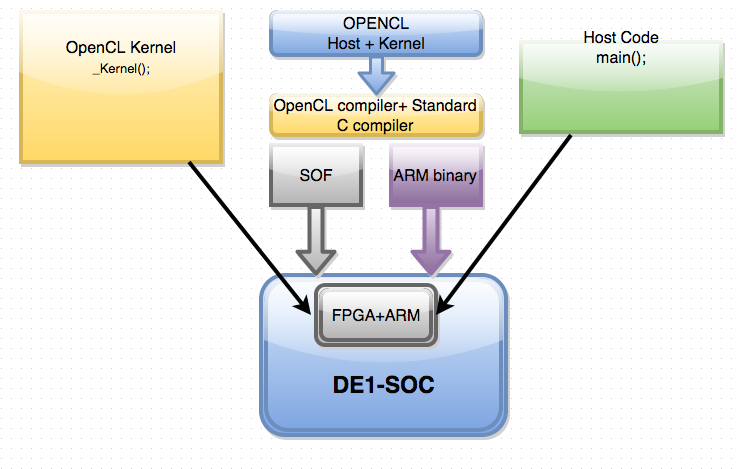
**OpenCL**

The OpenCL standard inherently offers the ability to describe parallel algorithms to be implemented on FPGAs, at a much higher level of abstraction than hardware description languages (HDLs) such as VHDL or Verilog. Although many high-level synthesis tools exist for gaining this higher level of abstraction, they have all suffered from the same fundamental problem. These tools would attempt to take in a sequential C program and produce a parallel HDL implementation. The difficulty was not so much in the creation of a HDL implementation, but rather in the extraction of thread-level parallelism that would allow the FPGA implementation to achieve high performance. With FPGAs being on the furthest extreme of the parallel spectrum, any failure to extract maximum parallelism is more crippling than on other devices. The OpenCL standard solves many of these problems by allowing the programmer to explicitly specify and control parallelism. The OpenCL standard more naturally matches the highly-parallel nature of FPGAs than do sequential programs described in pure C.

The creation of designs for FPGAs using an OpenCL description offers several advantages in comparison to traditional methodologies based on HDL design. Development for software programmable devices typically follows the flow of conceiving an idea, coding the algorithm in a high-level language such as C, and then using an automatic compiler to create the instruction stream. This approach can be contrasted with traditional FPGA-based design methodologies. Here, much of the burden is placed on the designer to create cycle-by-cycle descriptions of hardware that are used to implement their algorithm.

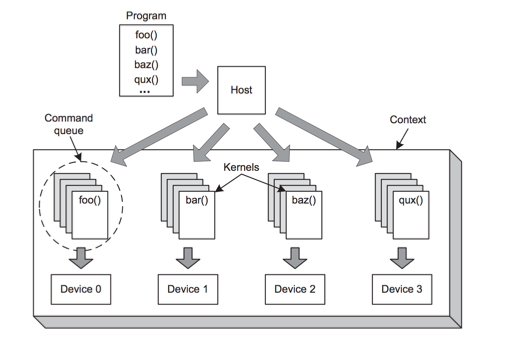
The traditional flow involves the creation of data-paths, state machines to control those data-paths, connecting to low-level IP cores using system level tools (e.g., SOPC Builder, Platform Studio), and handling the timing closure problems since external interfaces impose fixed constraints that must be met. The goal of an OpenCL compiler is to perform all of these steps automatically for the designers, allowing them to focus on defining and optimizing their algorithm rather than focusing on the tedious details of hardware design. Designing in this way allows the designer to easily migrate to new FPGAs that offer better performance and higher capacities because the OpenCL compiler will transform the same high-level description into pipelines that take advantage of the new FPGAs. One of the reasons behind this is because OpenCL is device independent, the programmer simply specifies the device that will be used on the host side, and the compiler will automatically optimize the code for that device. Figure 1 shows an overview of how the system is connected.



***Figure 1.Block Diagram of the Overall Setup***

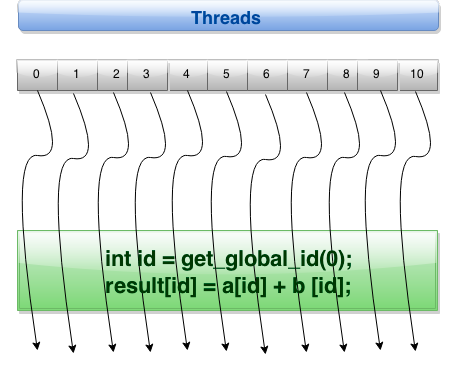
Down the middle of figure 1, we see the various compilers. The Altera OpenCL compiler generates an SOF file that is used to specify the specific hardware configures that the FPGA needs to use. This file contains all the place and route information. The standard C compiler generates the binaries that will run on the host.

As figure 1 shows, DE1-SOC board with FPGA+ARM core is the board used in this project. The host can be written in C++, C to configure the OpenCL Kernel. There are 6 data structures that the host manages; the first of which being the platform. The platform is used to identify vendors implementation of OpenCL. It gives a way to identify a way to access the device. The device is the hardware that the kernel will run on, for the course of this project that device is an  FPGA. One OpenCL program is not limited to one device; multiple devices can be run using the same host code to achieve large scale parallelism. The next key data structure the host is responsible for is the program. The program container is used to hold a list of kernels. Kernels are the specific functions or algorithms that will be ran, this is usually the computationally intensive task that is required to be ran. The kernel is also device independent, meaning that it can be ran on any device the that has OpenCL support(GPU’s, CPU’s, FPGA’s). The next big data structure is the command queue. It is the primary source of communication between the host and the device, the host sends the kernels that need to be ran to the command queue to get queued up to be executed. The last vital data structure is the context, which  is used to manage the connected devices. Figure 2 shows all of the various data structures are connected together. The DE1-SoC includes a 16-Kbyte memory that is implemented inside the FPGA. This memory is organized as 16K x 8 bits, and spans addresses in the range 0x08000000 to 0x08003FFF

****

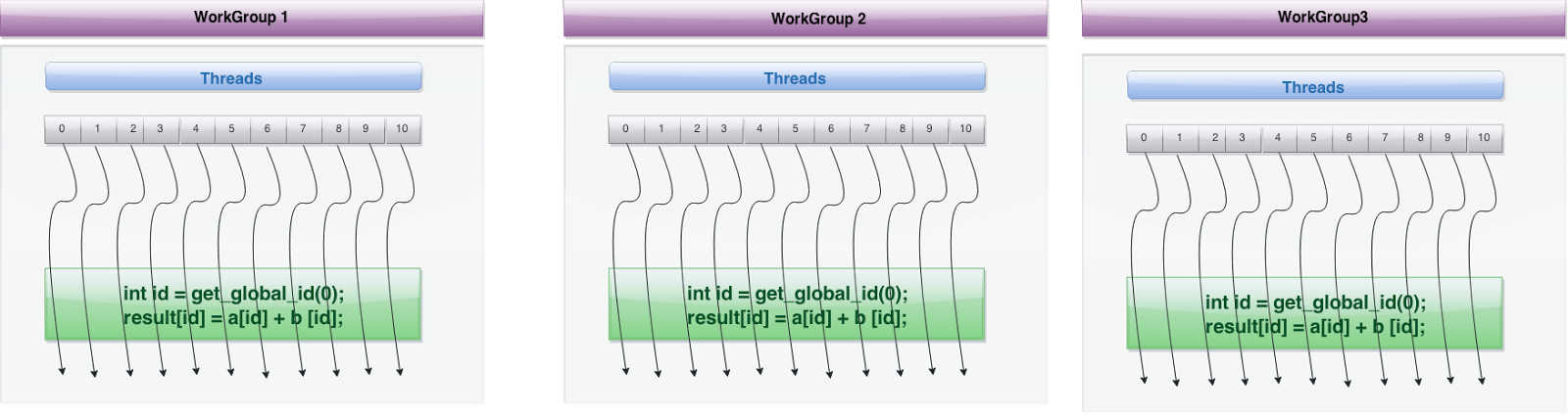
**Fig. 2 Device Management using Context Switching**

One of the key aspects of using OpenCL is the ability to parallelize tasks. An OpenCL kernel is executed via an array of work items, and all work items run a copy of the same code. Each individual work item is also assigned its own ID, which it can use to index a specific chunk of memory to run its computation on.



**Fig. 3 Basic thread layout**

Figure 3 shows how one work item is used to execute data independent operations in parallel via global ID. Several workgroups can be created that encompass these work items as shown in figure 4. OpenCL can divide the monolithic work item array into work groups.

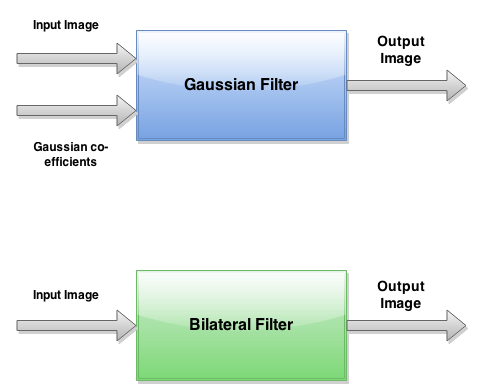


**Fig 4. An overview of workgroups**

Parallel work is submitted to available devices by launching kernels. The Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”. These work items executing within the same work group can be synchronized with each other via barriers or memory fences. The work items in different work groups can’t sync with each other, except by launching a new kernel.

**RTL Schematics**

This project is done at much higher abstraction level than RTL, and RTL schematic is automatically generated by the tool and compiler. The schematic varies based on the implementation of the design. A high level block diagram which corresponds to logical functionality of the design is shown below. There are two different Kernels which implements two filters. There hardware modules are different from each other and are generated in isolation, but when the main program is run these hardware blocks are instantiated in the design and used for computation.

****