Open Cl

- ✔ A framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), field-programmable gate arrays (FPGAs) and other processors.
- ✔ Open Cl includes a language for programming these devices, and application programming interfaces (APIs) to control the platform and execute programs on the compute devices.

A parallel language that provides us with two distinct advantages

- ✔ **Parallelism is declared by the programmer**
  Data parallelism is expressed through the notion of parallel threads which are instances of computational kernels

  Task parallelism is accomplished with the use of queues and events that allow us to coordinate the coarse grained control flow

- ✔ **Data storage and movement is explicit**
  Hierarchical Memory model
  Registers
  Accelerator Local Memory
  Global off-chip memory

Open Cl Structure

Natural separation between the code that runs on accelerators and the code that manages those accelerators .
"Host" code is pure software that can be executed on any sort of conventional microprocessor Soft processor, Embedded hard processor, external x86 processor

The kernel code is 'C' with a minimal set of extensions that allows for the specification of parallelism and memory hierarchy
Likely only a small fraction of the total code in the application.
Used only for the most computationally intensive portions.
Accelerator = Processor + Memory combo

Open Cl Host Program

A pure software written in standard 'C', Communicates with the Accelerator Device via a set of library routines which abstract the
communication between the host processor and the kernels.

```
main()
{
read_data_from_file( … );
manipulate_data( … );

clEnqueueWriteBuffer( … ); // Copy data from Host to FPGA

clEnqueueTask( … ); // Ask the FPGA to run a particular kernel
```

(..., my_kernel, ...);

clEnqueueReadBuffer(...); // copy data from
FPGA to Host

display_result_to_user(...);
}

Open Cl Kernels

Data-parallel function
Defines many parallel threads of execution
Each thread has an identifier specified by “get_global_id”
Contains keyword extensions to specify parallelism and memory hierarchy

__kernel
void
sum(__global const float *a, __global const float *b, __global float *answer)
{
int xid = get_global_id (0);
answer[xid] = a[xid] + b[xid];

}

Loopy Driver Generator

Is a framework for both, users and developers. The Loopy Driver Generator is intended to provide embedded system developers with a simple, convenient API to communicate with their designed hardware components.

This interface provides methods for synchronous as well as asynchronous communication over different communication media, for example Ethernet.

Such an interface for communication with VHDL components has not been done so far, though extensive research exists concerning efficient communication between PCs and FPGAs over different communication channels, mainly Ethernet.

The architecture has been prototyped with a C++ frontend communicating with a Xilinx Virtex 6 ML-605 FPGA over Ethernet. The design of the driver generator enables easy extension to support other frontend languages, FPGA boards or transport media. The host-side driver is written in C++11.

Xilinx vivado High-Level Synthesis accelerates IP creation by enabling C, C++ and System C specifications to be directly targeted into Xilinx. All Programmable devices without the need to manually create RTL. Supporting both the ISE® and Vivado design environments Vivado HLS provides system and design architects alike with a faster path to IP creation by :

- Abstraction of algorithmic description, data type specification (integer, fixed-point or floating-point) and interfaces (FIFO, AXI4, AXI4-Lite, AXI4-Stream)

- Directives driven architecture-aware synthesis that delivers the best possible QoR
- Fast time to QoR that rivals hand-coded RTL
- Accelerated verification using C/C++ test bench simulation, automatic VHDL or Verilog simulation and test bench generation
- Multi-language support and the broadest language coverage in the industry
- Automatic use of Xilinx on-chip memories, DSP elements and floating-point library

High Level Synthesis VS OpenCL

The two companies' paths, then, can actually be viewed as complementary. The difference is the target audience. Altera's strategy appears to favor the software engineer as the driver, and Xilinx's strategy appears to favor the hardware engineer. In the short term, it is unlikely that this difference will be the deciding factor in any team's decision to use one vendor or the other for the FPGA portion of their design.

The evolution of programmable devices, like FPGAs, has greatly accelerated in the last years, in an attempt to satisfy market demands. Some of the most important ones are the reduction of time necessary to develop a product with regard to ASICs or the greater exploitation of parallelism by FPGAs in comparison with CPUs or even GPUs. Although these demands are extremely ambitious, the solutions that major manufacturers, as Altera, with OpenCL SDK, and Xilinx, with Vivado HLS, are setting in motion are no less ambitious.

One of the first advances, which meant a revolution in architectures, was the launch of SoC (System on Chip) devices that integrate in one chip processor and programmable logic; that is, software and hardware. This duality allows SoC to gather the best of both worlds: the flexibility and reduced development time of software and the high throughput of hardware. However, the difficulty in the design of systems for SoC devices is achieving the trade-off that provides the end product with the **desired programmability** without giving up speed. An efficient solution would be to implement in the programmable logic the software functions that consume more execution time. This alternative is leading to a new revolution led by **high-level synthesis tools** that translate the functions meant to be accelerated in synthesizable code in the FPGA.

Xilinx HLS

For instance, in the case of Vivado HLS, the algorithm is designed in C/C++/System C and it is debugged within the same development environment. Afterwards, the algorithm is synthesized by generating the hardware code or RTL, being necessary to add the suitable directives to **get a good performance in terms of latency and throughput.** Finally, the RTL generated is exported to a block with the interface that is more adequate to the system in which it will be integrated.

OpenCL

In the case of the Altera OpenCL SDK, implementation is done in OpenCL, a standard of parallel programming of heterogeneous systems. From then on, the design flow is very similar to the Vivado HLS one. The Altera SDK **converts automatically kernel functions into hardware accelerators,**

adding the appropriate interfaces.

So, in an initial analysis, these tools seem to satisfy the market demands laid out at the beginning of this discussion. On one hand, they automate some of the most complex process in the hardware design flow, reducing the development time of a product compared with ASICs. On the other hand they allow to implement in logic the software functions that consume more CPU cycles, speeding their execution.

Commonalities

These synthesis tools, as **Vivado HLS** and **OpenCL** SDK, allow generating a FPGA implementation from a processor implementation in a high-level language. It saves having to make an additional hardware design where the developer would implement again the algorithm in a hardware description language (VHDL or Verilog), managing the FPGA logic: resources, transfers between registers.

Reference

http://www.gradiant.org/en/news/news/696-herramientas-de-sintesis-de-alto-nivel-para-fpgas-una-nueva-revolucion.html
http://www.eejournal.com/archives/articles/20130312-highlevel/