Maxeler Acceleration Technology

MaxCompiler is a compiler system for Maxeler hardware acceleration solutions using FPGAs. This white paper describes the components of MaxCompiler and illustrates accelerator programming using an example application.

Figure 1 sketches the architecture of a Maxeler hardware acceleration system which equips one or more FPGAs attached to a set of memories and connected to a host CPU via PCI Express channels. MaxRing interconnects (not shown in *Figure 1*) establish high bandwidth communication channels between the FPGAs on the accelerator.

Accelerating an application involves identifying the runtime intensive parts and turning them into an FPGA configuration. The FPGA configuration comprises arithmetic data-paths for the computations (the kernels) and modules orchestrating the data I/O for these kernels (the manager). Separating computation and communication into kernels and manager is beneficial as it allows for deeply pipelined kernels without data hazards which is key to achieving high performance.

Speedups are further increased by exploiting parallelism between several independent computation pipelines within kernels and by using several kernels. The number of pipelines and kernels that can be mapped to the accelerator is limited only by the parallelism inherent in the application and the size of the FPGA. While increasing the speedup, the parallel execution of several pipelines and kernels greatly stresses the memory system. Maxeler hardware acceleration solutions deliver the required high memory bandwidth by their customized memory architectures featuring multiple memory ports and configurable memory controllers.

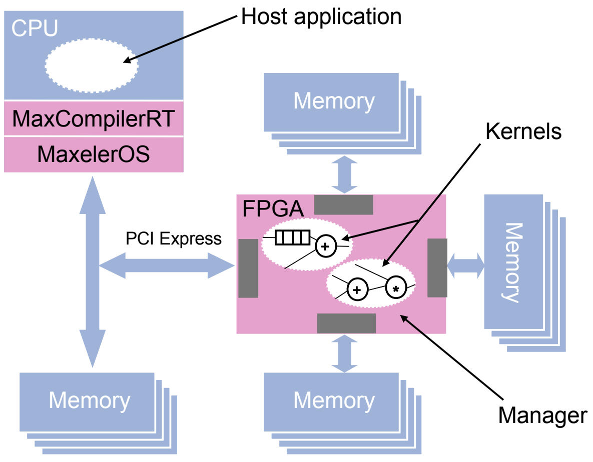

Figure 1: Maxeler accelerator architecture.

Development Tool Flow

Accelerating an application requires the user to develop three program parts:

- Kernel(s)
- Manager configuration
- Host application

MaxCompiler includes tools to support all three steps, the Kernel Compiler, the Manager Compiler and MaxelerOS, Maxeler's system for bridging between hardware and software. *Figure 2* presents the development tool flow with the main components of MaxCompiler.

The developer directs both the Kernel and the Manager Compiler by programs written in Java. However, using the tools requires only minimal familiarity with Java. The Kernel and Manager Compilers translate the kernels and the manager configuration into a low-level hardware description and target either a simulation of the design or a full hardware build resulting in an FPGA configuration. MaxelerOS connects the software and hardware parts of an accelerated application and provides the basis for C/C++ and Fortran interfaces available to the user. MaxCompiler is fully integrated with CAD tools from the FPGA vendors to gen-

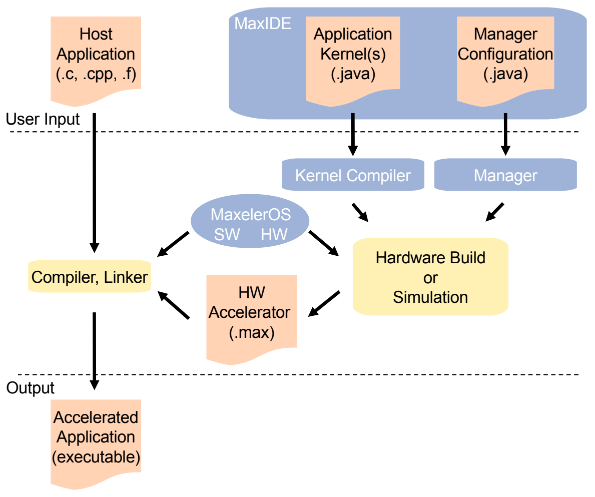
Host Application (.c, .cpp, .f)
MaxIDE
Application Kernel(s) (.java)
Manager Configuration (.java)
User Input
Kernel Compiler
Manager
MaxelerOS SW HW
Compiler, Linker
Hardware Build or Simulation
HW Accelerator (.max)
Output
Accelerated Application (executable)

Figure 2: Maxeler development tool flow.

erate the bitfiles to program the FPGAs. A hardware build automatically runs FPGA synthesis and backend tools and generates a file with the FPGA configuration bitstream and other data used to access the accelerator at runtime, the `.max` file. The hardware build process further provides reports with estimates on hardware resource usage and data for making performance projections.
The host application is compiled and linked with the `.max` file and the software part of MaxelerOS to the application executable. This executable includes all the code necessary to deal with the acceleration hardware, such as downloading the FPGA configuration and setting up the required data flows between CPU, FPGAs and memories.

Kernels

The Maxeler Kernel Compiler is a hardware compiler, and as such the program describes computations structurally (computing in space) rather than specifying a sequence of processor instructions (computing in time). A kernel is a streaming core with a data flow described by a unidirectional graph without cycles. Kernel graphs comprise several different node types:

- ○ Computation nodes perform arithmetic and logic operations (e.g., `+`,`*`,`<`,`&`) as well as type casts to convert between floating point, fixed point and integer variables.
- ▭ Value nodes provide parameters which are either constant or set by the host application at runtime.
- ◇ Stream 'offsets' allow access to elements at different positions in data streams.
- ▽ Multiplexer nodes are for taking decisions.
- ⬡ Counter nodes are for catching specific stream positions such as boundary conditions.
- ▼▲ I/O nodes connect the kernel to the manager and serve for streaming data in and out.

```
HWType flt = hwFloat(8,24);
HWVar x = io.input("x",  flt );

HWVar x_prev = stream.offset(x, −1);
HWVar x_next = stream.offset(x, +1);

HWVar cnt = control.
              count.simpleCounter(32, N);
HWVar sel_nl = cnt > 0;
HWVar sel_nh = cnt < (N−1);
HWVar sel_m = sel_nl & sel_nh;

HWVar prev = sel_nl ? x_prev :  0;
HWVar next = sel_nh ? x_next : 0;
HWVar divisor = sel_m ? 3.0 :  2.0;

HWVar y = (prev+x+next)/divisor;
io.output("y",  y,  flt );
```

Figure 3: Moving average kernel description.

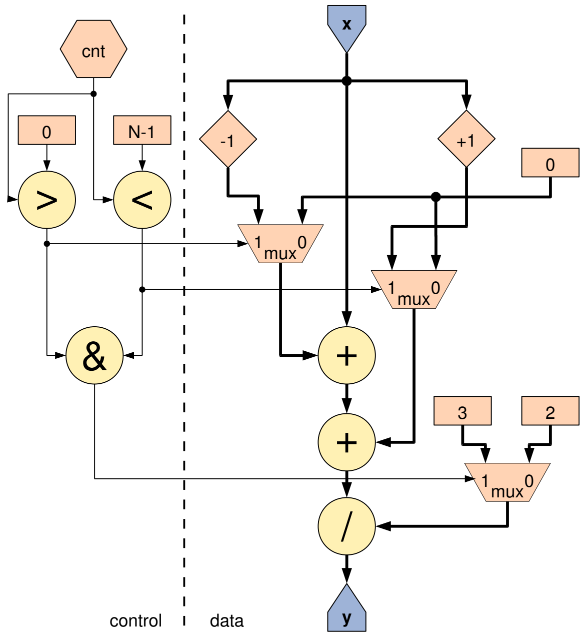

Figure 4: Graph for the moving average kernel.

The Kernel Compiler is effectively a Java software library and as such, kernel graphs are created by writing a Java program and executing it. *Figure 3* shows the kernel code for a 3-point moving average over N values with 2-point averages at the boundaries:

$$y_i = \begin{cases} (x_i + x_{i+1})/2 & \text{if } i = 0 \\ (x_{i-1} + x_i)/2 & \text{if } i = N{-}1 \\ (x_{i-1} + x_i + x_{i+1})/3 & \text{otherwise} \end{cases}$$

The program basically consists of a series of statements defining input and output streams and computations on streams. The computations are expressed in an equation-based style using regular

```
Manager manager = new Manager("MAV",
    MAX2BoardModel.MAX24412C);
KernelParameters p =
    manager.makeKernelParameters();
Kernel k = new MovingAverageKernel(p);

manager.setKernel(k);
manager.setIO(
    link ("x", DRAM(LINEAR)),
    link ("y", DRAM(LINEAR))
    );
manager.build();
```

Figure 5: Manager for moving average.

Java expressions on `HWVar` objects.

Figure 4 depicts the kernel graph for the moving average which splits into a data part (right-hand side) and a control part (left-hand side). Kernel graphs are directly mapped to hardware and then data streams through the arithmetic nodes. Efficient streaming kernels strongly emphasize the regularity of the data flow, making the actual computations look like a side-effect of streaming. Such streaming kernels lend themselves to deeply pipelined hardware implementations which are key to achieving high performance on FPGAs.

Manager

The manager wraps kernels and orchestrates their data I/O. Manager functions include user configurable I/O streams to PCI Express, to other FPGAs on the acceleration system via MaxRing and to DRAM memory.

For accessing the accelerator memory, MaxelerOS provides a library of pre-defined address generators for common addressing patterns, such as linear access, 2D array access in linear or strided mode and 3D array block access.

The host application on the CPU can configure and steer the data flow via the software runtime interface supported by MaxelerOS, for example by setting start addresses and block sizes for memory accesses and streaming data to/from the FPGA via PCI Express.

Figure 5 shows the configuration of a manager for the moving average kernel that connects its input and output streams to the on-card memory. The type of memory access requested is `LINEAR`, which will lead to address generators being placed on the chip to create a linear sequence of memory addresses.

The Maxeler development tool flow allows the user to program the manager in the same Java environment as the application kernels. The manager program typically instantiates the kernel(s) and manager and configures them. The Manager Compiler turns a manager configuration specified with Java into corresponding hardware.

```
// Set up memory address generators on the FPGA to
// stream the X and Y arrays to/from the kernel.
int64_t datasize = n*sizeof(float) / BURST_SIZE;
max_memory_stream_set_access_linear(dramctrl, "cmd_x", size);
max_memory_stream_set_start_address(dramctrl, "x", addr_x);
max_memory_stream_set_enable(dramctrl, "x", 1);

max_memory_stream_set_access_linear(dramctrl, "cmd_y", size);
max_memory_stream_set_start_address(dramctrl, "y", addr_y);
max_memory_stream_set_enable(dramctrl, "y", 1);

max_memory_commit_setting(device, dramctrl, FPGA_A);

// Trigger computation on the FPGA. The Kernel will run until
// it has processed n items, then the function will return.
max_run(device,
    max_runfor("MAVKernel", n),
    max_end()
);
```

Figure 6: Host application for the moving average example (fragment).

Host Application

The host application sits on top of MaxelerOS (see *Figure 1*) and is usually written in C/C++ or Fortran. The host application calls functions to load the FPGA configuration and to initialize DMA transactions to transfer data from the host memory to the accelerator and vice versa. The host application also sets up data streams in the manager and triggers the execution of the kernels. The code in *Figure 6* shows a fragment of the host application for the moving average example.

The code configures the address generators for the memory input and output streams to read a block of `n` items from `memaddress_x` and to write a similarly sized block to `memaddress_y`. After the memory settings have been committed to the FPGA device, the computation on the FPGA is triggered with the moving average kernel computing `n` items.

The code of *Figure 6* constitutes one phase of the application. Overall, the example has a sequence of three phases. The first phase streams raw data from the host over the PCI Express input to the accelerator memory. The second phase (the compute phase shown in *Figure 6*) streams data from

```
sel_nl.watch("sel_nl", Radix.BINARY);
sel_nh.watch("sel_nh", Radix.BINARY);
cnt.watch("cnt");
x_prev.watch("x_prev");
x_next.watch("x_next");
```

Figure 7: Debug code for moving average example.

the accelerator memory through the kernel back into memory. Finally, the third phase streams the resulting averaged data from the accelerator memory back to the host over the PCI Express output.

Debugging

The Maxeler Kernel and Manager Compilers generate representations of kernels and manager modules in the form of graphical images as part of the compilation process. Visualizing these graphs facilitates the analysis of the generated accelerator hardware.

Kernel designs can be rapidly developed by building and testing in simulation. The Maxeler fast Kernel Simulator offers visibility into the execution of a Kernel that is not available in a real FPGA implementation and, crucially, has a much quicker build time than for hardware output. The simulation of a design will run much more slowly than a real FPGA implementation, so hardware output can be used later in the design cycle to test with large data sets. To facilitate debug, the developer can add 'watches' to kernel designs to observe stream values over time. *Figure 7* shows a block of code which could be added to the moving average example in *Figure 3* to watch some of the stream variables. Output from these watches is generated in the form of a CSV file which can, for example, be viewed in a spreadsheet. *Figure 8* shows output from adding the proposed watches to the moving average example. Note some values are 'X' when the stream is offset beyond the total data available. Once the behavior has been verified in simulation,

sel_nl	sel_nh	cnt	x_prev	x_next
0	0	0	X	8
1	0	1	9	7
1	0	2	8	6
1	1	3	7	X

Figure 8: Debug output for moving average example with input $9, 8, 7, 6$ and $N = 4$.

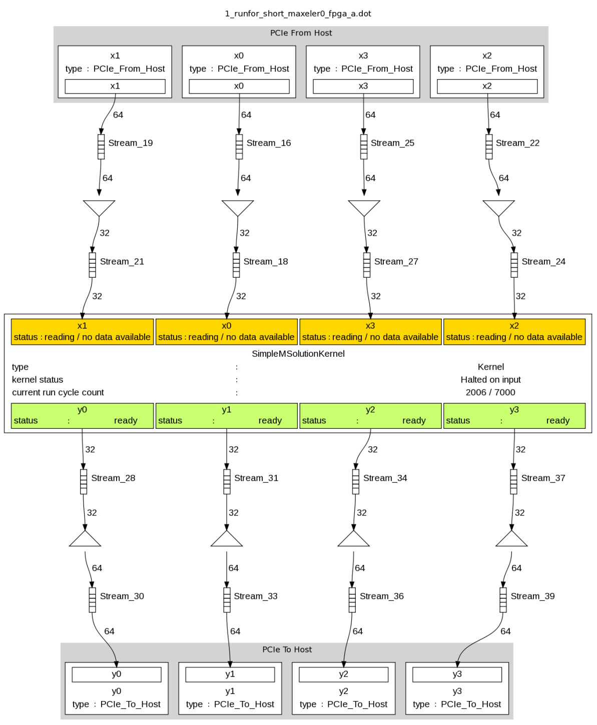

Figure 9: Debugging in hardware with MaxDebug.

then the design can be built in hardware. In hardware, debugging facilities are still available in the form of hardware exceptions (detecting overflow, underflow, divide by zero and invalid operations) and MaxDebug (shown in *Figure 9*), a graphical tool for debugging data flow scheduling in the complete accelerated system. MaxDebug allows the user to see how much data the Kernel has processed, its current status and how much data has passed through the input and output streams.

Optimization

During a hardware build MaxCompiler generates reports with hardware resource usage statistics. These resource usage figures drive design optimizations and are instrumental for analyzing the impact of using certain operators and programming styles on the size and, thus, the performance of the resulting hardware accelerators.

Training and Education

Maxeler Technologies provides educational material and workshops on programming streaming kernels, using MaxCompiler and creating complete hardware acceleration solutions.