Maximum Performance Computing

Maximum Performance Computing (MPC) changes the classical computer science optimization from ease-of-programming to maximizing performance and minimizing total cost of computing. We maximize performance by constructing compute engines to generate one result per clock cycle, wherever possible. Ease-of-programming is still important but takes second place to performance, computational density and power consumption.

As such MPC focuses on mission critical, long running computations with large datasets and complex numerical and intense statistical content. At Maxeler we drive MPC via 'Multiscale Dataflow Computing'. This white paper describes the components of MPC and illustrates how we program MPC dataflow computers.

One Maxeler Dataflow Engine (DFE) combines 10^4 arithmetic units with 10^7 bytes of local **f**ast SRAM (**F**MEM) and 10^{11} bytes of 6-channel **l**arge DRAM (**L**MEM). MaxelerOS allows the DFEs and CPU to run in parallel, so while the DFEs are processing the data, the CPU performs the non-time-critical parts of an application.

Our MPC programming environment comprises of MaxCompiler, MaxelerOS (running within Linux and DFEs themselves), MaxIDE, our fast DFE software simulator, and a comprehensive debug environment.

CPUs talking to DFEs

In a Maxeler MPC system, the CPUs are in control and drive the computations on the DFEs. The SLiC (Simple Live CPU) interface is Maxeler's API for CPU-DFE integration: at its simplest level DFE computation can be added to an application with a single function call, while for more fine-grained control SLiC provides an "action"-based interface. SLiC interface calls are automatically generated from the corresponding DFE program pieces. SLiC interfaces allow DFE programs to be shared and reused in library and application settings, such as MATLAB, spreadsheets, etc.

Figure 1: 1U rackable MPC-X node with 8 Dataflow Engines (DFEs) and Infiniband interconnect for communicating with standard CPU nodes running Linux via network switches.

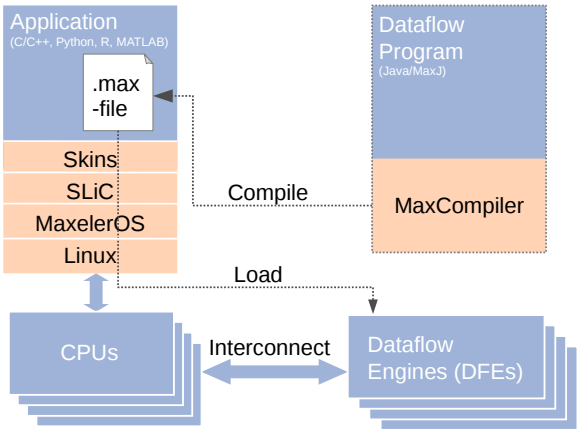

Figure 2: MPC System Architecture with Linux-based software stack (SLiC, Skins, MaxelerOS).

SLiC Skins provide bindings for a range of languages including C/C++, R, Python, and MATLAB. Skins enable users of these environments to access DFEs without needing to learn how to program DFEs in order to get started.

Using DFEs

There are two parts to an MPC application: a CPU and a DFE component, as sketched in *Figure 2*. DFE configurations are created using Maxeler's MaxCompiler and compiled to a `.max` file. The `.max` file contains automatically generated SLiC functions simplifying integration of DFE function-

```
1  from Convolve import "*" # Import the Convolve DFE
2
3  N = 1000          # initialize data e.g. read from file
4  weights = load_weights(N, N)
5  x = load_2d_arr_x(N, N)
6  y = load_2d_arr_y(N, N)
7
8  # load weights into DFE Memory
9  Convolve_loadWeights(N, N, weights)
10
11 # With weights now in DFE, can convolve many datasets
12 s = Convolve(N, N, x, y)
13 t = Convolve(N, N, x, s)
```

Figure 3: DFE Convolution from Python Skin.

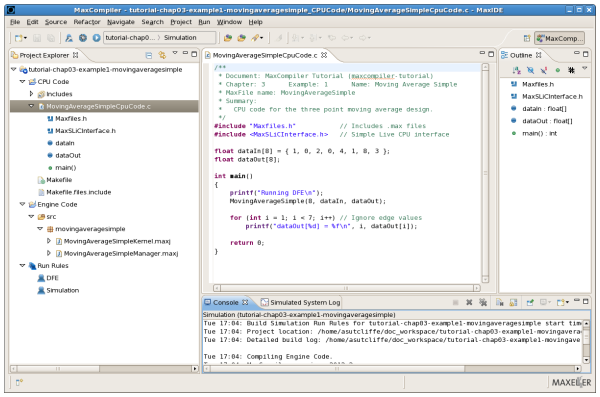

Figure 4: MaxIDE provides an integrated GUI to aid development of both DFE and CPU code.

ality into the CPU source, which can itself be written in a variety of languages. *Figure 3* shows how a DFE can be used from Python. SLiC has created a Python Skin, `Convolve.py`, that defines two auto-generated function calls: one function loads the DFE Memory, the other streams data from the CPU over the interconnect to the DFE, through the convolution computation, and back to the CPU over the interconnect.

Development Tool Flow

The DFE part of an accelerated solution itself contains two components: one or more Kernels, responsible for the data computations; and a single Manager, which orchestrates global data movement for the CPUs, DFEs and Kernels+Memory inside. Hence, accelerating an application requires the user to write three program parts:

- Kernel(s)
- A Manager
- and a CPU application

The developer creates the Kernel and the Manager by writing programs in MaxJ: an extended form of Java adding operator overloading. Using MaxCompiler requires only minimal familiarity with Java. A developer executes a MaxCompiler-based program to produce a "`.max` file" containing the DFE configuration, meta-data and SLiC functions. The CPU application is compiled and linked with the `.max` file, SLiC and MaxelerOS, to create the application executable; this executable includes all the code necessary to utilize the DFE, such as configuring DFEs and setting up the dataflows (R)DMAs between CPUs and DFEs, and configurations for memory controllers for DFE memories. MaxelerOS (within Linux) and SLiC connect the software and DFE parts of an application and provide C/C++ or other skins interfaces to the user. MaxIDE, based on the Eclipse open source platform, provides a complete MaxCompiler development environment with MaxJ/Java syntactic code assistance and automated build and run tools; *Figure 4* shows a screenshot of MaxIDE. Both DFE and CPU code can be developed and run from MaxIDE.

Kernels

MaxCompiler kernels describe computations structurally (computing in space), rather than specifying a sequence of processor instructions (computing in time). A kernel's dataflow is described by a graph of computational cores:

Computation cores perform arithmetic and logic operations (e.g., `+`,`*`,`<`,`&`) as well as type casts to convert between floating point, fixed point and integer variables.

Value cores provide parameters which are either constant or set by the CPU application at runtime.

Stream 'offsets' allow access to elements at different positions in the dataflow.

Multiplexer cores are for making decisions.

Counter cores are for catching specific dataflow positions such as boundary conditions.

I/O cores connect the kernel to the manager and serve for streaming data in and out.

Figure 5 shows the kernel code for a 3-point moving average over N values with zero at the boundaries:

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

We define inputs and outputs and the computations to go from inputs to outputs. Dataflow

```
DFEType type = dfeFloat(8,24);
DFEVar x = io.input("x", type);

DFEVar x_prev = stream.offset(x, -1);
DFEVar x_next = stream.offset(x, +1);

DFEVar cnt = control.count.simpleCounter(32, N);
DFEVar valid = (cnt > 0) & (cnt < (N-1));

DFEVar y = valid ? (x_prev + x + x_next) / 3.0 : 0.0;
io.output("y", y, type);
```

Figure 5: Moving average kernel description.

streams are represented as `DFEVars` and used via regular Java expressions and function calls.

Figure 6 depicts the kernel graph for the moving average, split into a data part (right-hand side) and a logic part (left-hand side). Computation happens while the data flows through the network without any dynamic events. The computations are all dependent on each other, while still running fully in parallel. Consequently, we can estimate the performance of the Kernel by dividing the amount of data that needs to flow through the kernel with the expected data rate. Being able to accurately predict performance of implementation options makes it easy to explore optimizations in a spreadsheet and then implement the best one, which is hard to do on classical CPU systems.

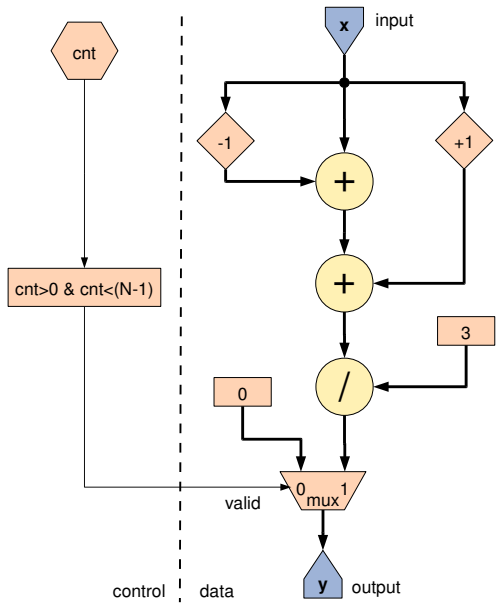

Figure 6: Graph for the moving average kernel.

```
Manager m = new Manager(params);
Kernel  k = new MovingAverageKernel(
                    m.makeKernelParameters());
m.setKernel(k);
m.setIO(link("x", CPU),
        link("y", CPU));
m.createSLiCInterface();
m.build();
```

Figure 7: Manager for moving average.

Manager

The manager wraps kernels and orchestrates their data choreography. Manager functions include user configurable I/O streams to CPUs (R)DMA, to other DFEs in the node, and to the off-chip Large Memory. The manager program gets "executed" at runtime by MaxelerOS, partly running on CPUs and partly running on the DFE itself.

Figure 7 shows the configuration of a manager, for the moving average kernel, that connects its input and output streams to the CPU. The call to `createSLiCInterface()` generates a simple engine interface; more complex engine interfaces can be described by the programmer, and a DFE might have several engine interfaces to handle different usage models. When the Kernel(s) and Manager are compiled with MaxCompiler, software functions are generated for each engine interface and included in the `.max` file. These engine interface functions can be called from CPU code written in C/C++, or from other languages such as Python and MATLAB via the SLiC Skins interface.

MaxIDE allows the user to program the manager in the same environment as the application kernels. The manager program typically instantiates the kernel(s) and manager and configures them, then the Manager Compiler turns this configuration into a corresponding `.max` file.

CPU Application

The CPU application sits on top of SLiC and MaxelerOS (see *Figure 2*). MaxIDE can automatically

```
const int N = 80;
float x[80], y[80];
for (int i=0; i<N; ++i)
    x[i] = 10.0 * rand() / RAND_MAX;

// This SLiC function is generated automatically by MaxCompiler.
MovingAverage(N, x, y);
```

Figure 8: CPU application for the moving average example (fragment).

```
debug.printf(cnt < 8,
             "%d: sel=%d x=%.2f y=%.2f\n",
             cnt, sel, x, y);
```

Figure 9: Debug code for the moving average example.

manage a C framework for the CPU code, *Figure 8* shows a fragment of the C code for the moving average example with a call to the SLiC function `MovingAverage`. This figure and the Python example of *Figure 3* show how easily the DFE can be integrated into a CPU application.

Simulation and Debugging

Kernel designs can be rapidly developed by simulating first in software and then building DFE configurations to run at maximal speed. Maxeler's fast Simulator allows us to run DFE programs in software on the CPU.

A key feature for debugging is a DFE `printf` function, analogous to the C language version, which traces stream values during execution. DFE printf can be used in both simulation and on the DFE, and the amount of print output can be controlled by a DFEVar condition. *Figure 9* shows a code example for DFE printf in *Figure 5*.

Optimization

Each line in a Kernel program generates pieces of dataflow computing. Since we compute in space, each line creates certain resources. DFE resource annotation brings this information back into the source code on a line-by-line basis. *Figure 10* shows an example of this. For example, resources are conserved by preferring integer operations over floating point, and converting floating point values to fixed point where feasible, using less bits to represent numbers, and sharing operators. The balance of parallelism vs precision can be carefully adjusted using the capacity usage as a guide.

LUTs	FFs	BRAMs	DSPs	FilterKernel.maxj
665	847	9.5	2	resources used by this file
0.22%	0.14%	0.89%	0.10%	% of available
				`int len = 10000;`
				`int bits = MathUtils.bitsToAddress(len);`
				`DFEFloat type = dfeFloat(8, 24);`
9	32	0.0	0	`DFEVar x = io.input("x", type);`
1	32	0.0	0	`DFEVar y = io.input("y", type);`
80	46	9.5	0	`Memory<DFEVar> w = mem.alloc(type, len);`
				`w.mapToCPU("weights");`
39	45	0.0	0	`DFEVar addr = simpleCounter(bits,len);`
536	692	0.0	2	`DFEVar sum = x + y * w.read(addr);`
				`io.output("s", sum, dfeFloat(8, 24));`

Figure 10: Resource usage report. LUTs, FFs, DSPs (multiply/add units) are fundamental resources used to construct dataflow cores.

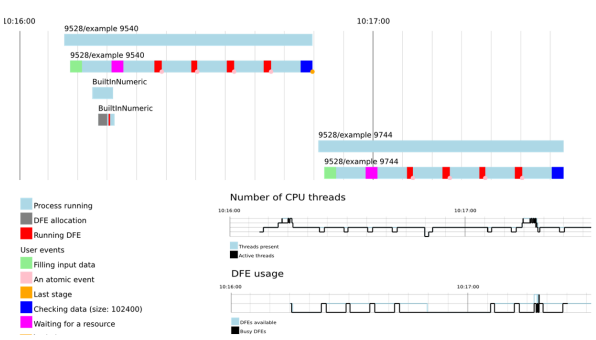

Figure 11: CPU and DFE Profiling.

Latency annotation reports have a similar layout to resource reports, but show line-by-line detail of how long (clock cycles or nanoseconds) it takes data to flow through the DFE. This is useful for network applications where nanoseconds can matter, and turnaround time between data arrival and departure is crucial.

Clock cycles can vary from Kernel to Kernel. Timing reports detail which clock frequency was achieved and where the current bottlenecks are to achieve higher clock frequencies. Bottlenecks arise from local crowding of signals and computation, e.g. by reading a DFEVar many times, creating many circular constructs, or just pushing towards the limits of the DFE space limit.

The interaction of the CPU and DFE is also important when optimizing an application: the CPU and DFE should run simultaneously in parallel for maximum performance, with no idle time on either side. SLiC includes tools which instrument the code and produce event flow graphs; *Figure 11* shows an example of a profiling graph. Visualization of how the runtime execution is spread between the CPU and DFE helps identify under-utilized compute bandwidth.

Training and Education

Maxeler Technologies provides MPC educational material and workshops, using MaxCompiler and also delivers complete Multiscale Dataflow solutions.