Massive Parallel QCD Computing on FPGA Accelerator with Data-Flow Programming

Thomas Janson and Udo Kebschull

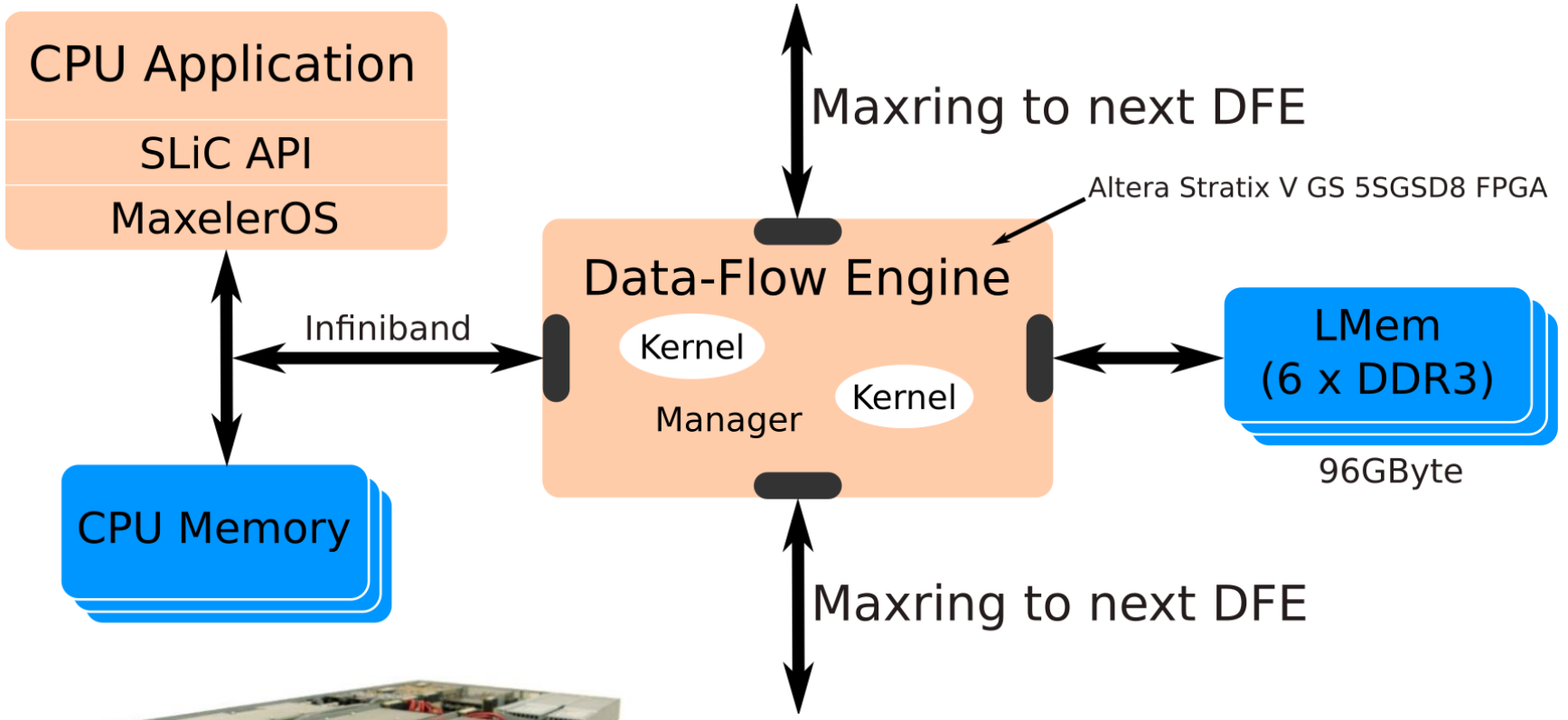
Infrastructure and Computer Systems in Data Processing (IRI)
Goethe University Frankfurt
Germany

ACAT 2017
University of Washington, Seattle

Introduction and Overview

- Motivation: fully enabled parallelism and next neighbor search by using data-flow programming
- Compute framework
- Concept of data-flow programming
- Wilson Dirac operator most intense algorithm
- Algorithm and neighbor indexing
- Dslash Operator as Dataflow Graph
- Implementation
- Results
- Conclusion

Framework – Maxeler System Architecture

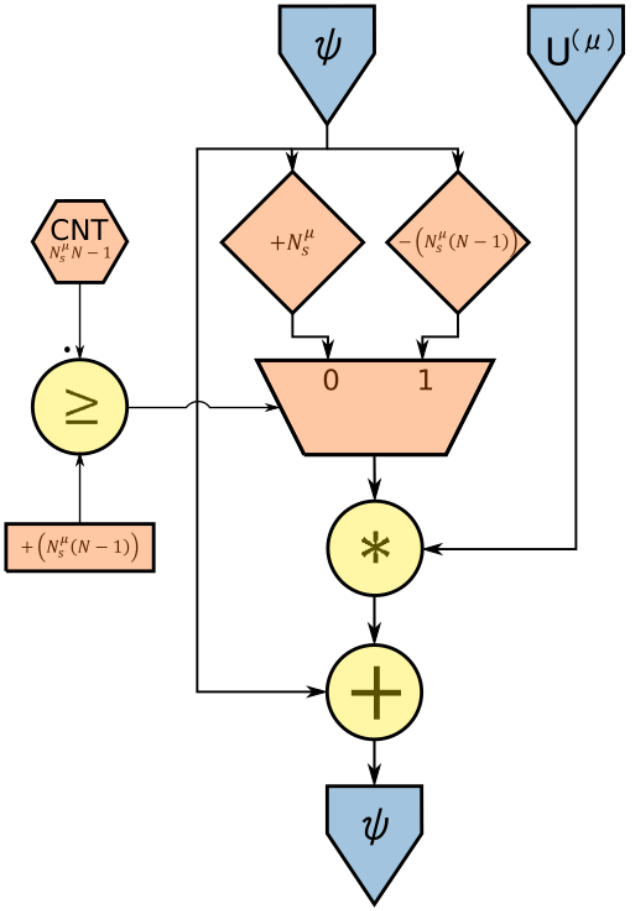


Picture from: Maxeler Technologies, "Programming MPC Systems", White Paper, June 2013

Concept of Data-Flow Programming

- Algorithm as data-flow graph
- Small set of node types
- All nodes are fully pipelined
- Arc can hold more than one data item
- All arithmetic operations are spatially distributed
- Control flow and data flow decoupled
- Offset operator picks up data items within the data stream

- Stream length known at compile time

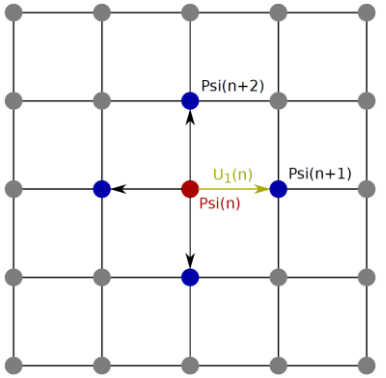


Wilson Dirac Operator

- *We compute:* $\Psi' = (1 - \kappa H)\Psi$
- *H collects all neighbor terms*

$$\mathrm{H} = \left(\sum_{\mu=1}^{4} (1 - \gamma_\mu) \mathrm{U}_\mu(\mathrm{n})\, \psi(\mathrm{n} + \mu) + (1 + \gamma_\mu) \mathrm{U}_\mu^{\mathrm{T}}(\mathrm{n} - \mu) \psi(\mathrm{n} - \mu) \right)$$

- Site Index: $n = (x, y, z, t)$
- Array Index: $n = x + y * N_s + z * {N_s}^2 + t * {N_s}^3$
- AoS data layout
  - for each site index n: $(\psi, U_1, U_2, U_3, U_4)$



- Index arithmetic to collect next neighbor
  - $n_{up} = x + \big((y + 1)\%N_s\big) * N_s + z * {N_s}^2 + t * {N_s}^3$
  - $n_{down} = x + \big((N_s + y - 1)\%N_s\big) * N_s + z * {N_s}^2 + t * {N_s}^3$

Algorithm

Algorithm 1 Wilson Dirac Operator

for $t = 0$ to $N_t - 1$ **do**
  for $z = 0$ to $N_s - 1$ **do**
    for $y = 0$ to $N_s - 1$ **do**
      for $x = 0$ to $N_s - 1$ **do**
        MemIndex $n \leftarrow x + y * N_s + z * N_s^2 + t * N_s^3$
        Spinor result $\leftarrow 0$
        for $\mu = 1$ to 4 **do**
          MemIndex neighbourUp $\leftarrow n.\text{up}(\mu)$
          MemIndex neighbourDown $\leftarrow n.\text{down}(\mu)$
          result $\leftarrow$ result $+ ((1 - \gamma[\mu]) * (U[\mu, \text{neighbourUp}] * \Psi[\text{neighbourUp}]))$
          result $\leftarrow$ result $+ ((1 + \gamma[\mu]) * (U[\mu, n] * \Psi[\text{neighbourDown}]))$
        end for
        $\Psi'[n] \leftarrow \Psi[n] -$ result
      end for
    end for
  end for
end for

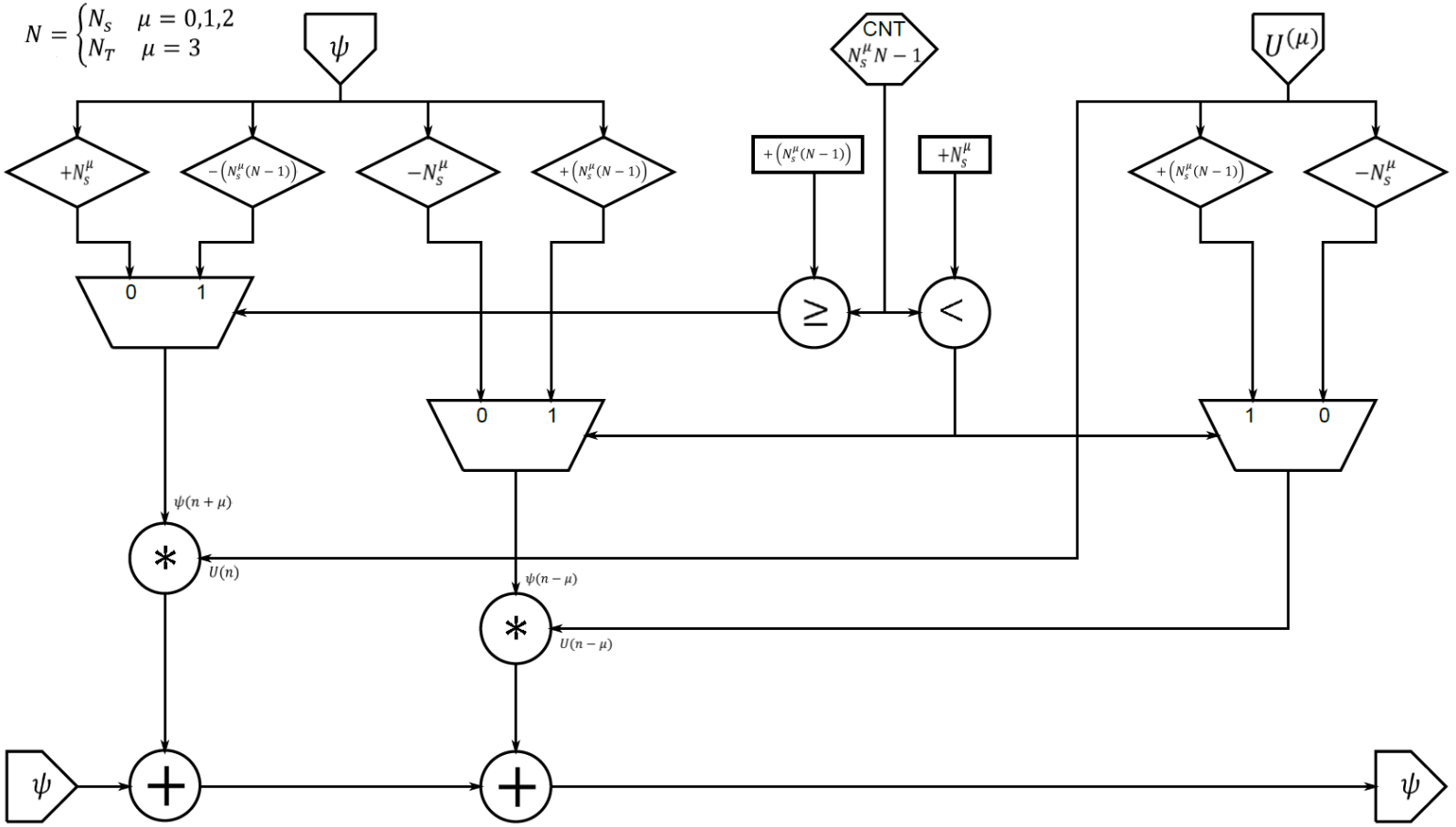
Data-Flow Graph in MaxJ

- The outer for loops (x,y,z,t) are unrolled (data stream)
- The inner loop describes the four graphs for each direction

```
for(int mu=0;mu<4;mu++) {
  int N = (mu<3) ? Ns : Nt;
  cnt[mu] <== control.count.simpleCounter(32, exp(Ns,mu)*N);
  up0[mu] <== stream.offset(spinorIn, exp(Ns,mu));
  up1[mu] <== stream.offset(spinorIn,-(exp(Ns,mu)*(N-1)));
  spinorUp[mu] <== ((exp(Ns,mu)*(N-1)) > cnt[mu]) ? up0[mu] : up1[mu];
  down0[mu] <== stream.offset(spinorIn,-(exp(Ns,mu)));
  down1[mu] <== stream.offset(spinorIn,(exp(Ns,mu)*(N-1)));
  spinorDown[mu] <== (cnt[mu] < exp(Ns,mu)) ? down1[mu] : down0[mu];
  link0[mu] <== stream.offset(linkIn[mu], -(exp(Ns,mu)));
  link1[mu] <== stream.offset(linkIn[mu], (exp(Ns,mu)*(N-1)));
  linkDown[mu] <== (cnt[mu] < exp(Ns,mu)) ? link1[mu] : link0[mu];
  spinorUpDashed[mu] <== dslashLocalForward(linkIn[mu],spinorUp[mu],mu);
  spinorDownDashed[mu] <== dslashLocalBackward(linkDown[mu],spinorDown[mu],mu);
  spinorPartialSum[mu] <== addSpinor(spinorUpDashed[mu], spinorDownDashed[mu]);
 }
DFEStruct spinorPartSumA = addSpinor(spinorPartialSum[0], spinorPartialSum[1]);
DFEStruct spinorPartSumB = addSpinor(spinorPartialSum[2], spinorPartialSum[3]);
DFEStruct spinorResult = mul(kappa,addSpinor(spinorPartSumA, spinorPartSumB);
```

Dslash Operator as Data-Flow Graph

- Loop transformation leads to:



Data Layout and Number Representation

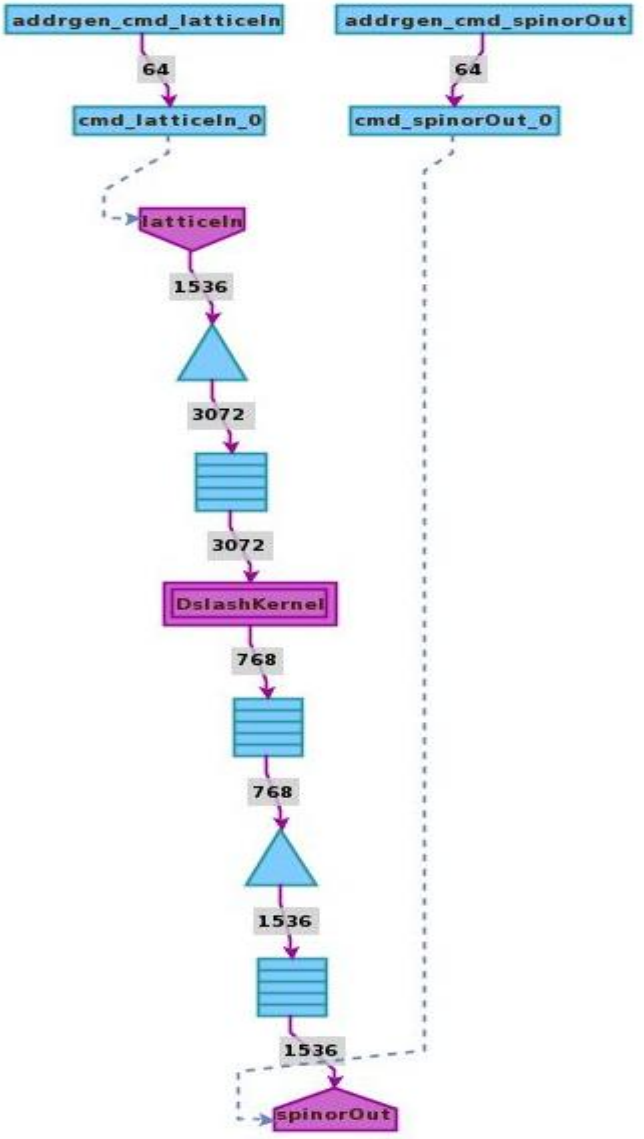
- We store the lattice in LMem
- AoS (array of structure) data layout
- We use 32 bit fixed-point number representation
  - 23 fractional bits, 7 bit mantissa (signed integer)
  - Advantage: much lower resource usage
  - Entries for the link variable smaller than one (unitary 3x3 matrices, determinant 1)
  - In case of spinor field, an estimation is done against a given data set (shows relative error of 10^-4% against exact values computed with Mathematica)
  - Same test against single precision floating point shows same relative error
  - We use this design with the option to test against more data sets

Kernel Throughput

- Kernel throughput depends on kernel clock rate (133 MHz max)
- We use 32 bit fixed-point number representation
- Input stream with 47 GByte/s
- Output stream with 12 GByte/s
- Theoretical equivalent peak performance 176 GFLOP/s
  - our kernel computes 1320 FLOPs each clock cycle

Memory Interface

- We stream from LMem, and write the result back to LMem
- 6 x DDR3 memory , 384 bit interface
- Burst rate 8 , 3072 bit intern
- 384 Byte in one kernel tick
- Memory Interface 1066 Mbps (hard to get timing closure)
- Memory Interface configured with 800 Mbps and kernel with 100 MHz
- The design is considered to be memory bound



Implementation Results

- Resource usage:
- Lattice size restricted
  - $16^3 \times 4$
- Kernel peak performance
  - with 100 MHz – 132 GFLOP/s
- Peak memory bandwidth
  - 35.76 GByte/s
- Measured with maxtop utility from the Maxeler framework
  - FPGA usage 49.6 %
  - Memory bus utilization 62.5 %
  - Power usage 35.5 W
  - Memory bandwidth 23.64 GByte/s
  - Performance 66 GFLOP/s
- Comparison against CL2QCD implementation

Resource Usage				
Logic utilization	114771	/	262400	(43.74%)
Primary FFs	203944	/	524800	(38.86%)
Secondary FFs	5810	/	524800	(1.11%)
Multipliers	2304	/	3926	(58.69%)
DSP blocks	1152	/	1963	(58.69%)
Block Memory	2489	/	2567	(96.96%)

Platform	Memory BW	Performance	Perf./BW ratio
OpenSPL FPGA (32 bit fixed-point)	23.64 GByte/s	66 GFLOP/s	2.79
AMD Radeon HD 5870 (DP) [10]	100 GByte/s	60 GFLOP/s	1.2
AMD Radeon HD 7970 (DP)[11]	200GByte/s	120 GFLOP/s	1.2

[10] M. Bach, V. Lindenstruth, O. Philipsen, C. Pinke, "Lattice QCD based on OpenCL", arXiv:1209.5942v1, 2012

[11] O. Philipsen, C. Pinke, A. Sciarra and M. Bach, "CL^{2}QCD - Lattice QCD based on OpenCL," PoS LATTICE 2014 (2014) 038 [arXiv:1411.5219 [hep-lat]].

Conclusion

- Dslash operator can be implemented in a highly parallel and efficient way on an FPGA using the data-flow approach
- Processes all arithmetic operations in parallel with 1320 FLOPs per clock cycle
- Measured memory bandwidth of 23.64 GByte/s
- Measured performance 66 GFLOP/s
- We reach a very good performance bandwidth ratio