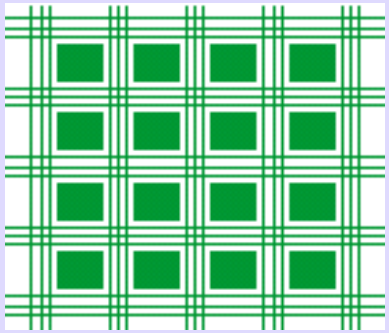
High-Performance Computing for Low-Power Systems

Ghent University, Belgium

Erik D'Hollander

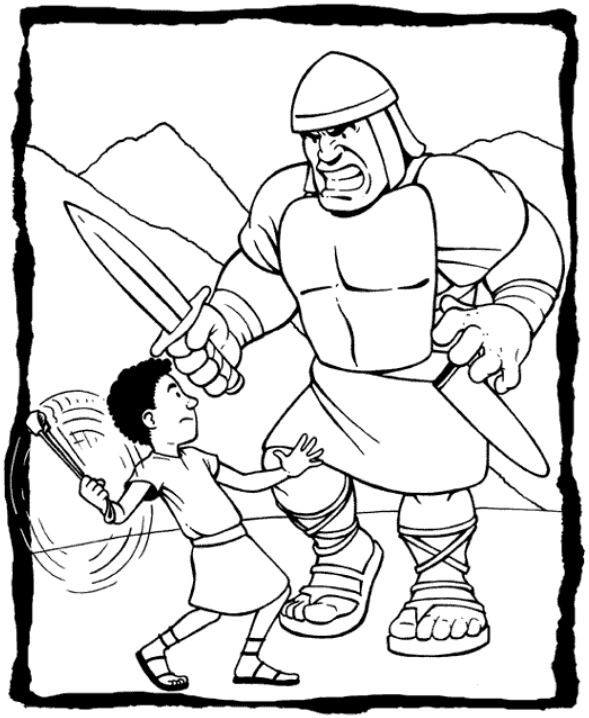
erik.dhollander@ugent.be



Advanced HPC Systems workshop
Cetraro, June 27-29, 2011

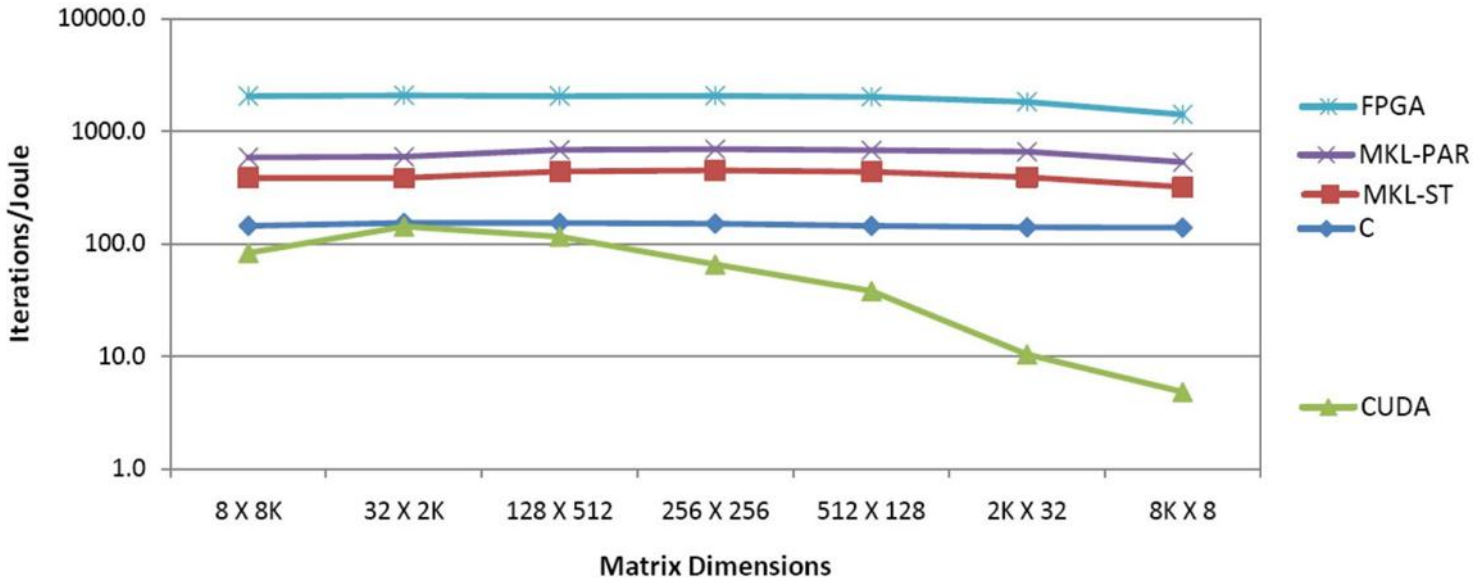
Overview

- GPU's vs. FPGA's
  - Power gain
  - Architecture
  - Programming
- Application
  - Wavelet CAD
  - GPU performance
  - FPGA simulation
- Conclusion



David & Goliath

Power gain: Miles per Gallon Iterations per Joule…



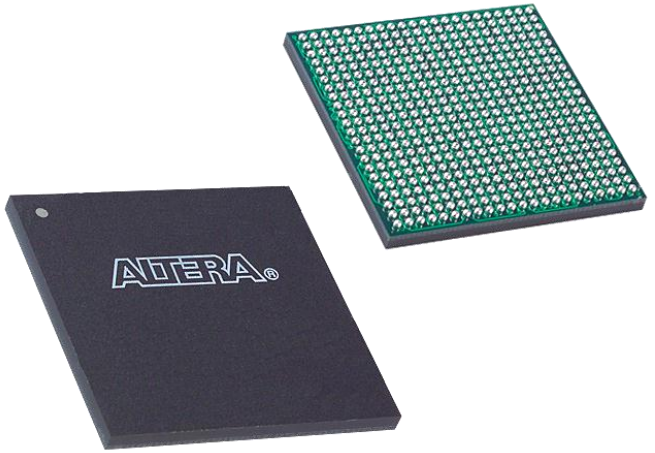
*Comparison of Gaxpy kernel on naive C, MKL-single thread, MKL-parallel, CUDA BLAS and FPGA

*Srinidhi Kestur, John D. Davis, Oliver Williams: BLAS Comparison on FPGA, CPU and GPU. ISVLSI 2010: 288-293

Architecture Commonalities

Common GPU and FPA characteristics

- Accelerators
- Parallel processing
- Streaming data

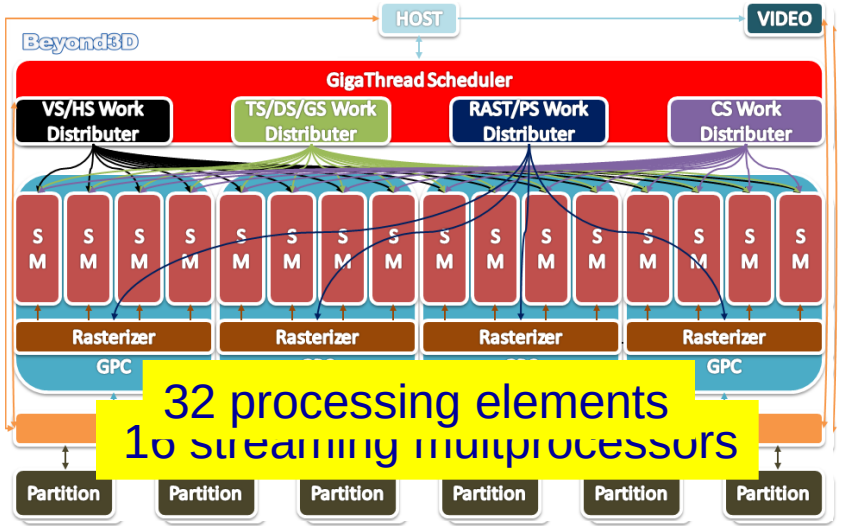


The weakest link…

Fast thread scheduler!

GPU architecture

- Fermi GPU



32 processing elements

16 streaming multiprocessors

x16 crossbar

Max. 512 running threads

GPU programming

- CUDA
- Massively parallel

Number of threads (MxN)

HOST

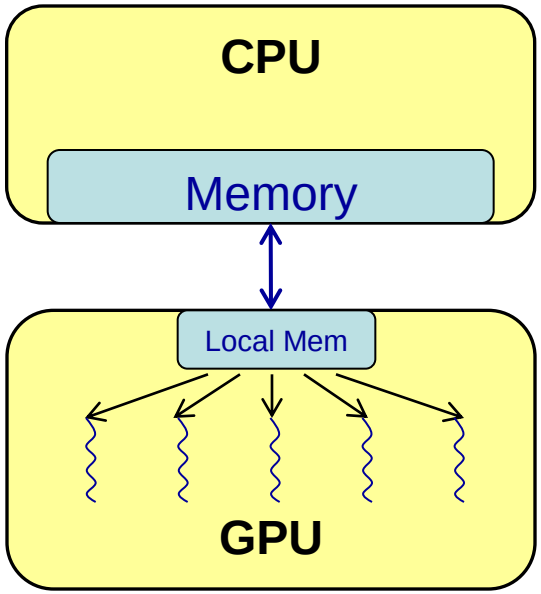
```
dim3 dimBlock(M, N);
dim3 dimGrid(1,1);
MatrixMulKernel<<<dimGrid, dimBlock>>>( M, N, P );
```

GPU

```
__global__ void MatrixMulKernel(float* M, float* N, float* P)
{
// Calculate the row index of P[Row][Col] and M[Row][k]
        int Row = blockId.y * N + threadId.y;
// Calculate the column index of P[Row][Col] and N[k][Col]
        Int Col = blockId.x * TILE_WIDTH + threadId.x;
        Pvalue = 0;
// each thread computes one element of result matrix
        for (int k = 0; k < N; ++k)
                Pvalue += M[Row][k] * N[k][Col];
        P[Row][Col] = Pvalue;
}
```

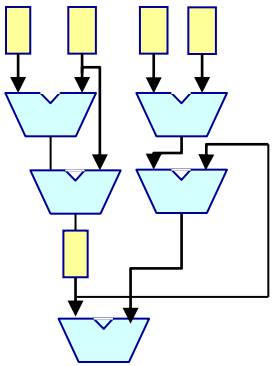
GPU as accelerator

- Send stream → Process → Receive stream
- Between CPU and GPU:
  - DMA
- Within GPU:
  - Parallel threads
  - SPMD

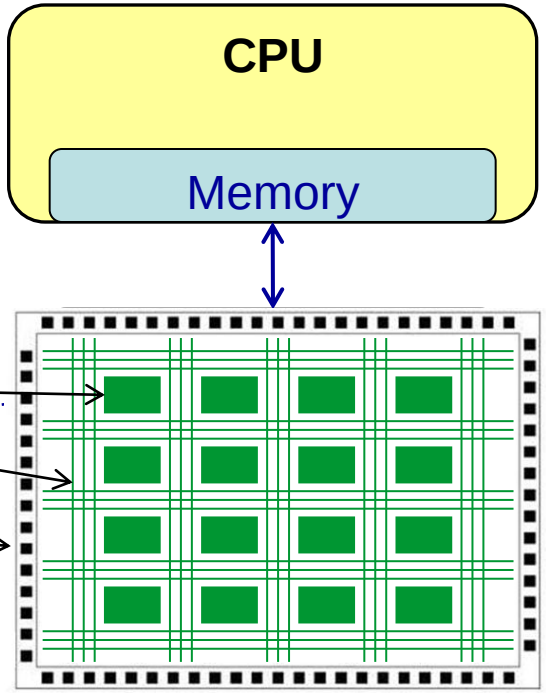


FPGA as accelerator

- Send stream → Process → Receive stream
- Between CPU and FPGA:
  - DMA
- Within FPGA:
  - *Pipelined datapaths*
  - *Parallel datapaths*



Very wide data memory ensures continuous dataflow to every datapath



GPU ↔ FPGA operation

- GPU = *Von Neumann*: instruction fetch, decode…
  IF ID OF EX WB
  - instructions and operands in memory
  - operand address calculation requires instructions
- FPGA = *Dataflow*: instructions (operations) pipelined in hardware
  ~~IF ID~~ OF EX WB
  - 40% gain
  - instructions in pipeline, operands in memory
  - operand address calculation requires *address generator* (AG)
  - loops require *loop controller (to feed data back into pipeline)*

GPU → FPGA ports?

Trials

- FCUDA* : CUDA port to FPGA
- PGI CUDA : CUDA port to x86 platforms

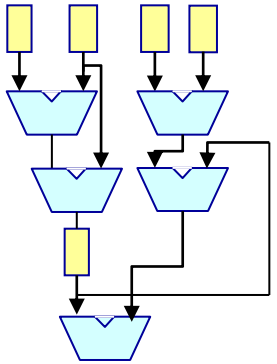
Problems

- CUDA cannot express pipelines
- CUDA requires C → VHDL compiler

*FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs, Alexandros Papakonstantinou et al., ASP, 2009

ROCCC*: a direct approach

- C→VHDL compiler
  - Creates pipelined datapaths
  - Creates parallel loop control
  - Supports streaming paradigm
  - Hierarchical design (modules, systems)
  - Eclipse programming environment
  - VHDL Testbench generation
  - Integrates with existing IP-cores and Xilinx toolchain
- Limitations with respect to FPGA environment
  - No use of pointers
  - Array index expressions limited to loop index + constant e.g. a[i+6]



*Riverside Optimizing Compiler for Configurable Circuits

ROCCC*: modules and systems

- **Modules**
  - Computational datapath
  - Translates into pipelined logic blocks
  - Scalar arguments, scalar results
  - Loops unrolled
  - E.g. FIR filter (5 taps);

```
void FIR(int A0, int A1, int A2, int A3, int A4, int& result)
 { const int T[5] = { 3, 5, 7, 9, 11 } ;
   result = A0*T[0] + A1*T[1] + A2*T[2] + A3*T[3] + A4*T[4];
 }
```

*Jason R. Villarreal, Adrian Park, Walid A. Najjar, Robert Halstead: Designing Modular Hardware Accelerators in C with ROCCC 2.0. FCCM 2010: 127-134

ROCCC*: modules and systems

- **Systems**
  - Loops on streaming data I/O → *loop controller*
  - Operates on streams of arrays → *address generator*
  - Compiler organizes reads/writes of data in each clock cycle
  - E.g. FIR array

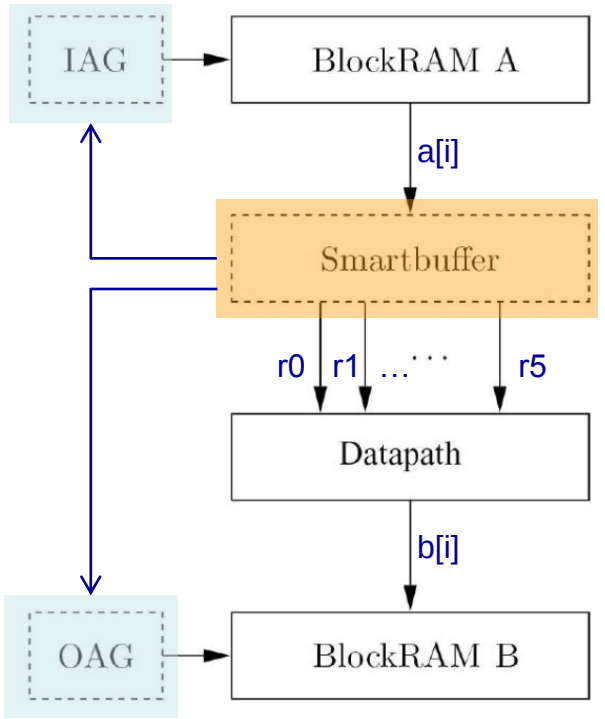
```
#include "roccc-library.h“   → modules/systems library
void FIRSystem(int* A, int* B)
{   for(int i = 0 ; i < 100 ; ++i)
        FIR(A[i], A[i+1], A[i+2], A[i+3], A[i+4], myTmp)
        B[i] = myTmp ;        → FIR module from modules library
}
```

*Jason R. Villarreal, Adrian Park, Walid A. Najjar, Robert Halstead: Designing Modular Hardware Accelerators in C with ROCCC 2.0. FCCM 2010: 127-134

ROCCC: Smart Buffer

- **Objective: minimize memory accesses to FPGA BlockRAM**
  - Fetched min. data in each cycle
  - Stores data in registers
  - Reuses data in registers
  - Optimizes bandwidth
  - Minimizes pipeline stalls
- E. g. FIR filter

  a[i] → r_(i mod 5)
  b[i] ← FIR(r0,r1,r2,r3,r4)
  b[i+1] ← FIR(r1,r2,r3,r4,r0)
  b[i+2] ← FIR(r2,r3,r4,r0,r1)
  b[i+3] ← FIR(r3,r4,r0,r1,r2)
  b[i+4] ← FIR(r4,r0,r1,r2,r3)



ROCCC: Smart Buffer

- **Extended to multiple streams, multiple windows**
- E. g. Edge detection

```
for (i=1 ; i<255 ; i++) {
  for (j=1 ; j<255 ; j++) {
    maskV = (A[i-1][j+1] - A[i-1][j-1])
          + (A[i][j+1] - A[i][j-1])
          + (A[i+1][j+1] - A[i+1][j-1]);

    maskH = (A[i+1][j-1] - A[i-1][j-1])
          + (A[i+1][j] - A[i-1][j])
          + (A[i+1][j+1] - A[i-1][j+1]);

    B[i-1][j-1] = maskV*maskV + maskH*maskH;
  }
}
```

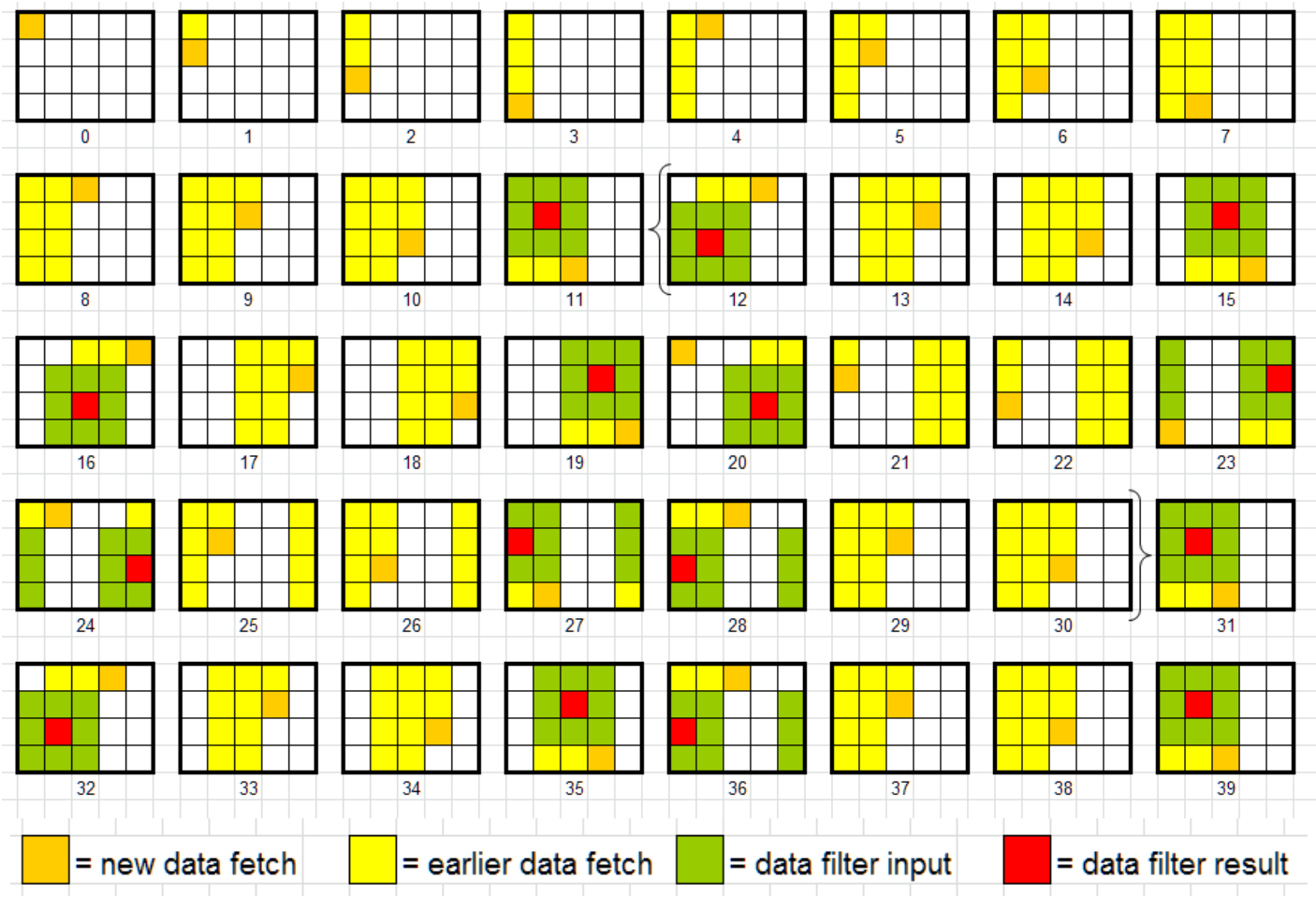
Replace array elements by smart buffer regs
Remove loop headers (replace by loop control)

ROCCC: Smart Buffer

- **Extended to multiple streams, multiple windows**
- Resulting program: loop body = scalar operations on registers

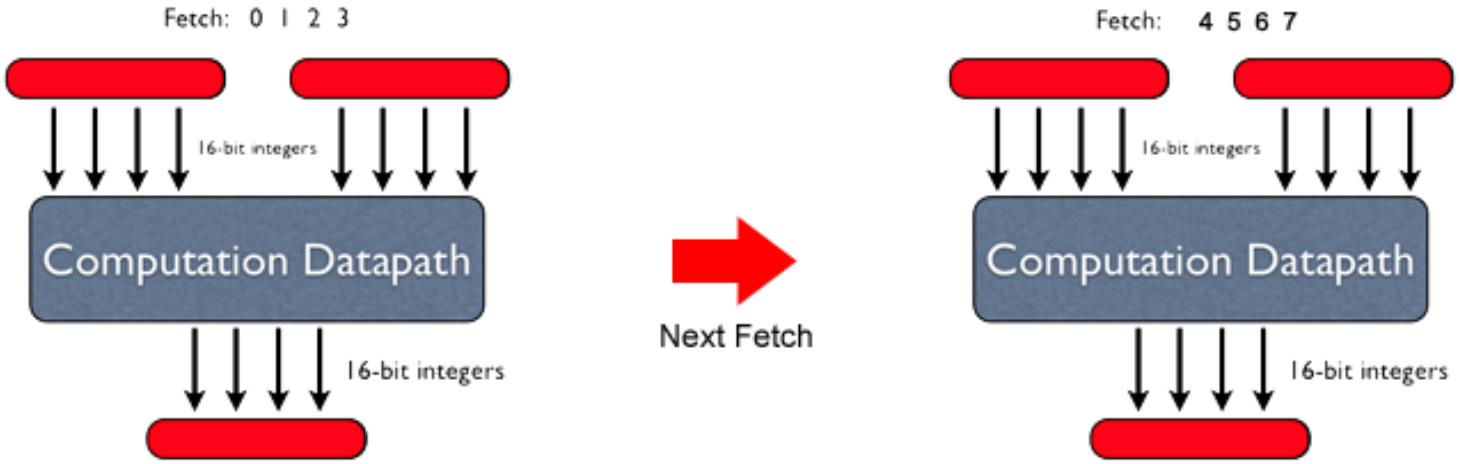
```
maskV = (dataIn_0_2 - dataIn_0_0)
      + (dataIn_1_2 - dataIn_1_0)
      + (dataIn_2_2 - dataIn_2_0);

maskH = (dataIn_2_0 - dataIn_0_0)
      + (dataIn_2_1 - dataIn_0_1)
      + (dataIn_2_2 - dataIn_0_2);

B_0_0 = maskV*maskV + maskH*maskH;
```

Smart buffer windows



ROCCC: Other optimizations

- *__Stream management__
- Exploit wide memory bus (e.g. 64 bit) to feed multiple // streams
- Expandable internally with very wide dual port BlockRAM

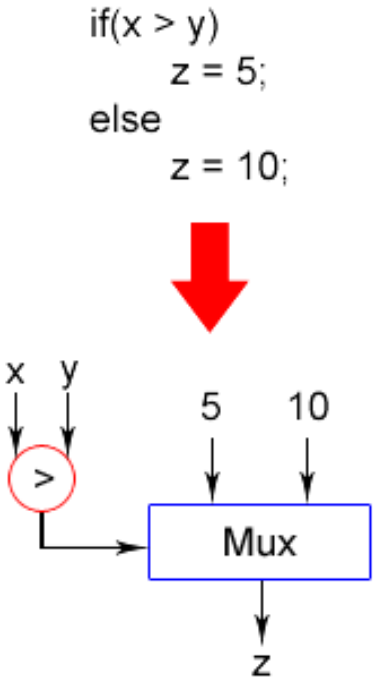


*ROCCC tutorials

ROCCC: Other optimizations

- Loop unrolling
- Loop tiling
- Loop interchange
- If predication

- Scalar replacement
- Sequential loop pipelining
- Systolic array wavefront generation
- Pipelining retiming



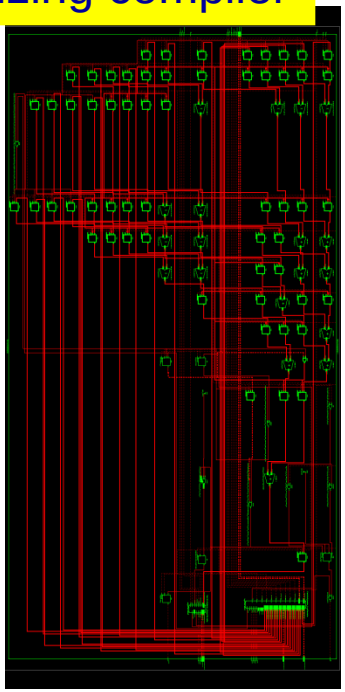
FPGA programming toolchain

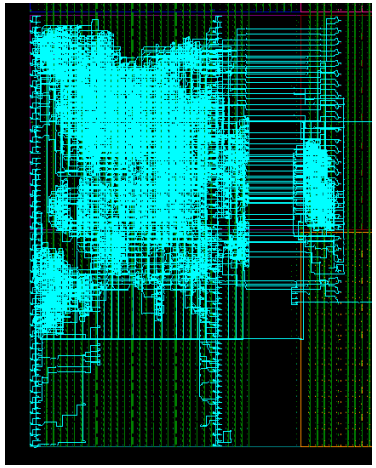
C → VHDL ← ROCCC
FPGA optimizing compiler

s

Generate circuit

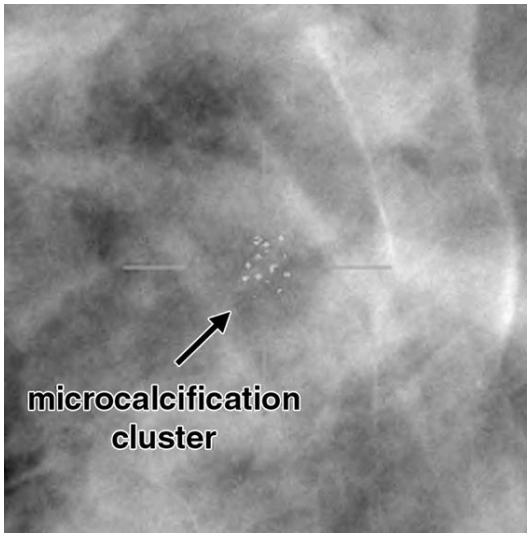
```
void MatrixMultiplication(int** A, int** B, int*** C0)
{
  int i ;
  int j ;
  int k ;
  int currentSum ;

  for (i = 0 ; i < 10 ; ++i)
  {
    for (j = 0 ; j < 10 ; ++j)
    {
      currentSum = 0 ;
      for (k = 0 ; k < 10 ; ++k)
      {
      currentSum += A[i][k] * B[k][j] ;
      }
      C0[i][j][k] = currentSum ;
    }
  }
}
```





Application: DWT

- Discrete wavelet transformation for Computer Aided Diagnosis
- **Objective**: detect microcalcifications (small granularity) in X-ray
- **Solution:**
- DWT of image
- Dismiss low frequencies
- IDWT of image
- $\Rightarrow$ enhanced image



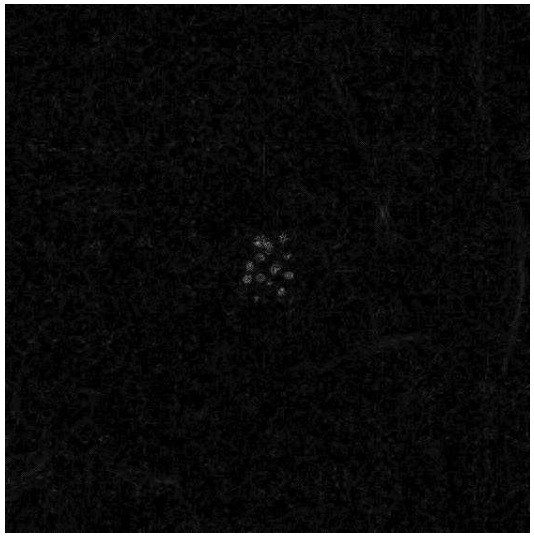
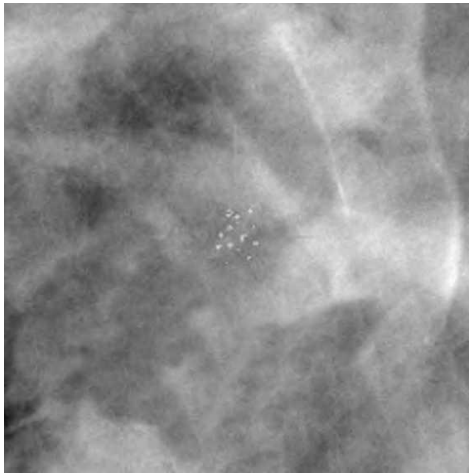
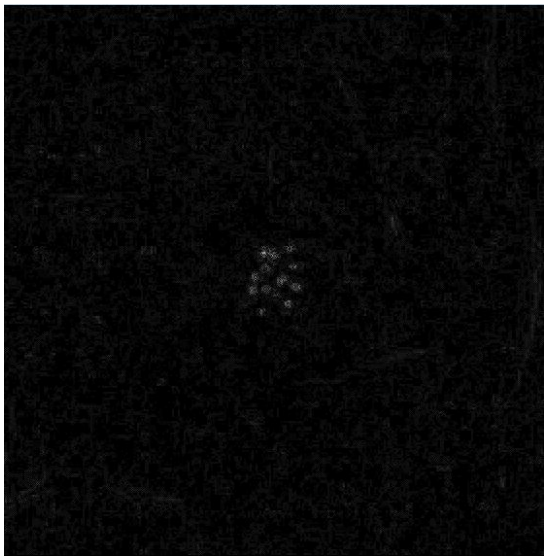


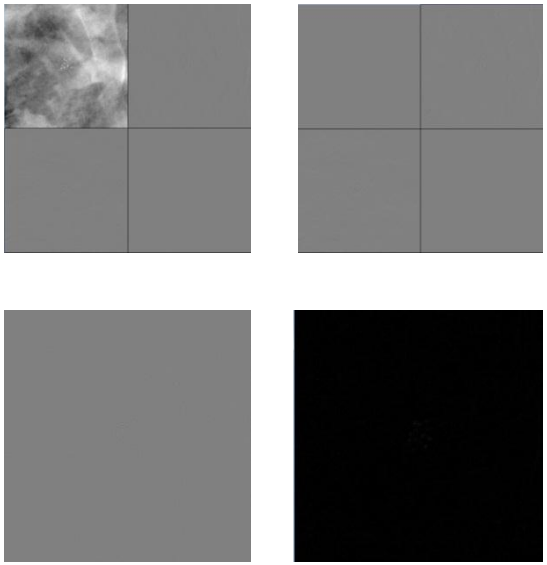
Image enhancement



Steps

1. Discrete Wavelet Transform
2. Remove LL part
3. Inverse DWT
4. Normalize
5. Stretch

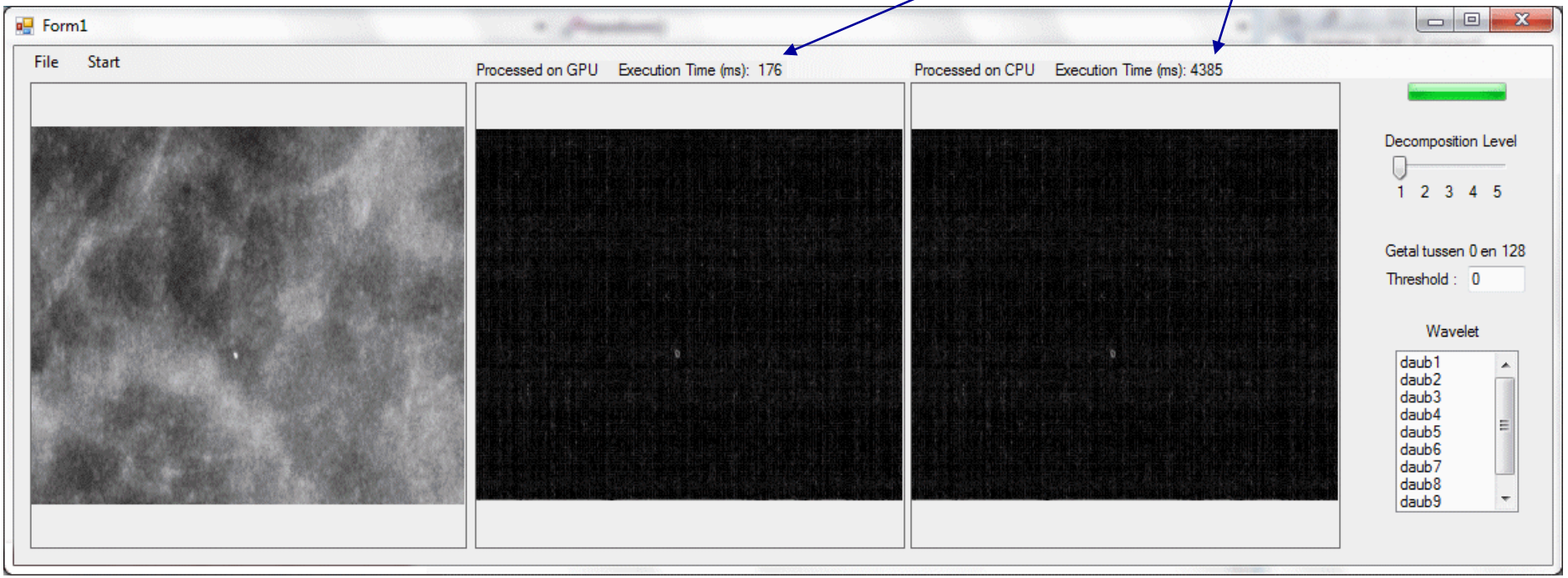




GPU speed-up

- Using CUDA, parallelizing 5 phases*
- GT-330 NVIDIA (desktop GPU)

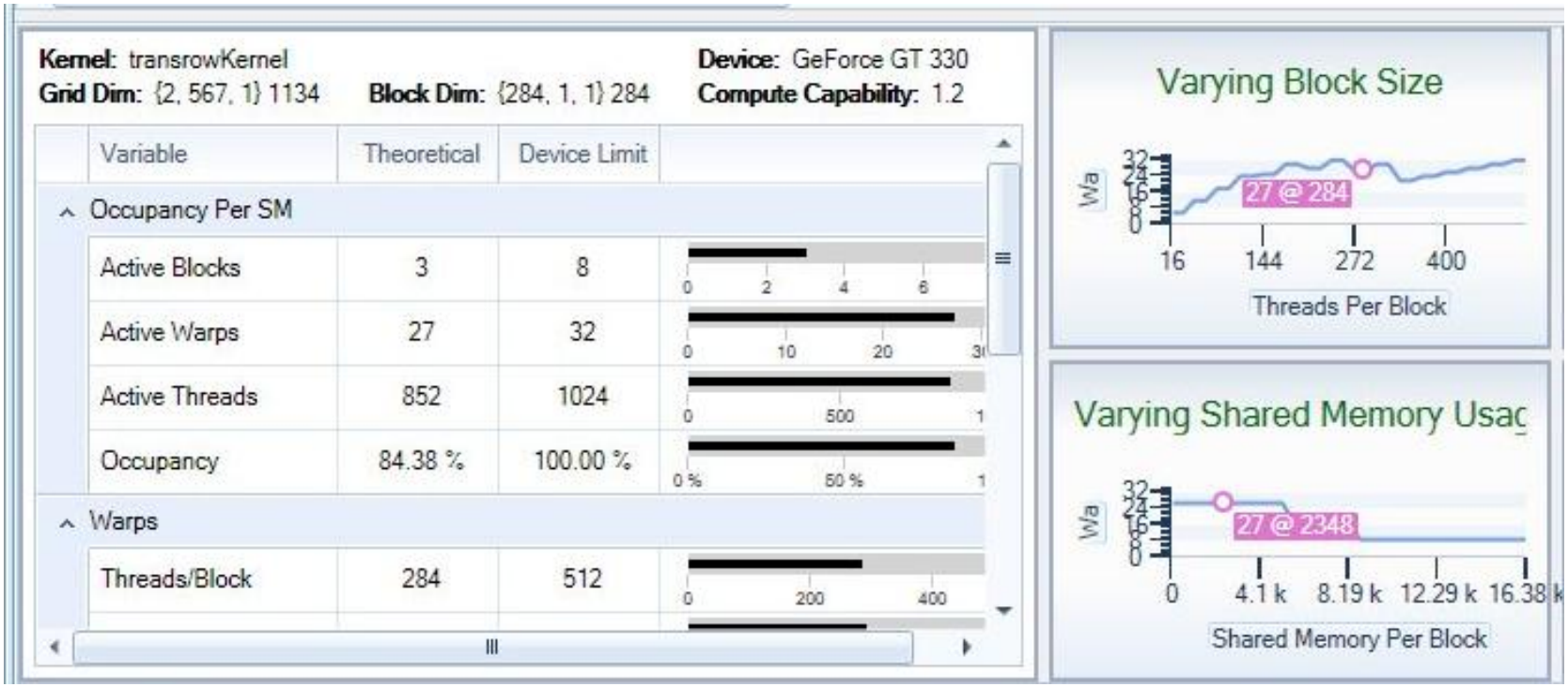
GPU 176 ms
CPU 4,385 ms
Sp = 24.9



*Dries Rosseel, Generating speed-up for the optimization of RX-images, Msc thesis, 2010

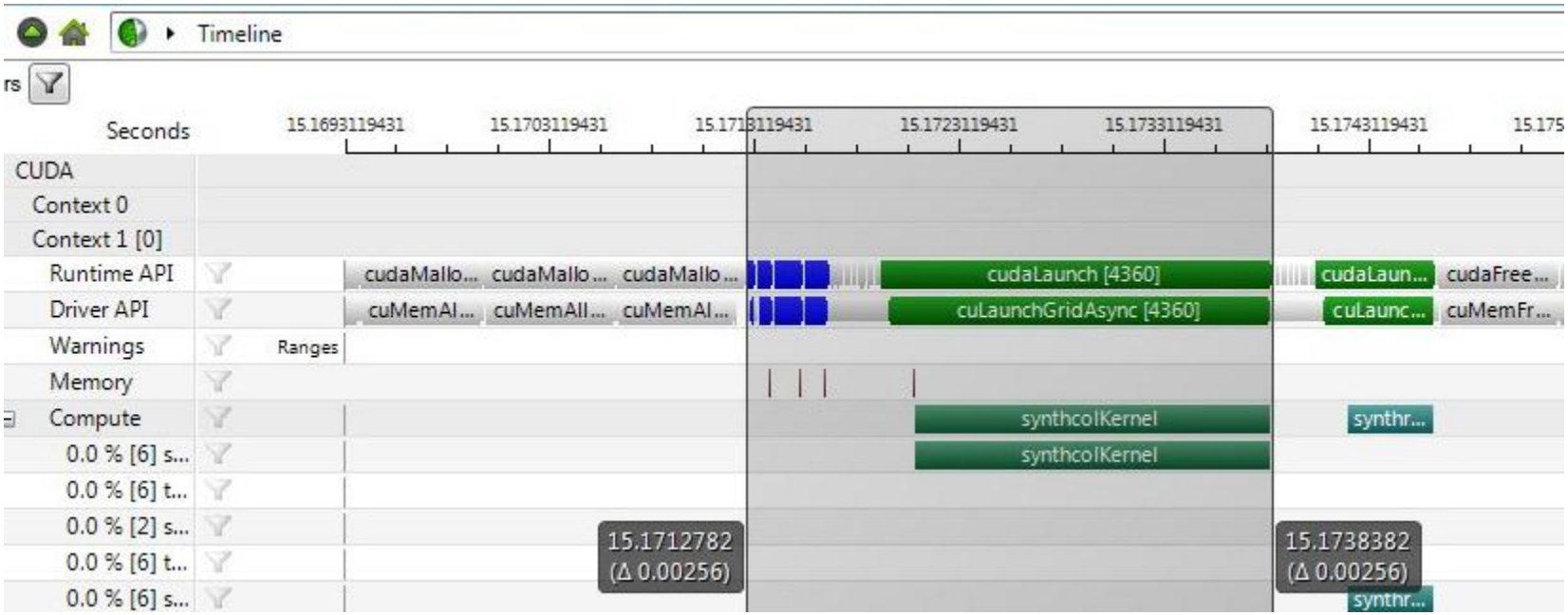
GPU – performance analysis

- Using NSight



GPU – performance analysis

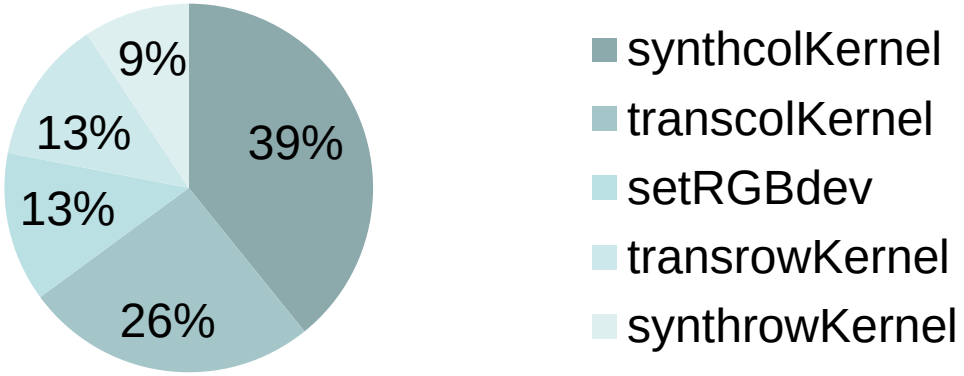
- Using NSight



GPU – performance analysis

- Using NSight

Top Device Functions by Total Time					
Launch Summary | All					
Name	Launches	Total (µs)	Min (µs)	Avg (µs)	Max (µs)
synthcolKernel	6	10,340.21	1,716.63	1,723.37	1,731.00
transcolKernel	6	6,785.76	1,122.59	1,130.96	1,137.85
setRGBdev	2	3,466.74	1,729.27	1,733.37	1,737.47
transrowKernel	6	3,331.95	551.677	555.325	559.166
synthrowKernel	6	2,458.23	404.029	409.705	420.319



FPGA – ROCCC code

- **DWT kernel extraction: transrow kernel**
- Modified in several ways:
- A) code
  - outer loop parallel
  - inner loop streamlined
  - innerst loop unrolled
  - bodies pipelined
- B) data
  - output “dest[][]” → 2 arrays “desty[]”

```
for (unsigned int y = 0; y < h; y++) {
  for (unsigned int k = 0; k < w2; k++) {
    s = 0.0f;
    d = 0.0f;
      for (int m = tH.first(); m <= tH.last(); m++) {
        n = 2 * k + m;
        if (n < 0) n = 0 - n;
        if (n >= (int)w) n -= 2 * (1 + n - w);
        s += tH[m] * float(sour[y][n]);
                        }
      for (int m = tG.first(); m <= tG.last(); m++) {
        n = 2 * k + m;
        if (n < 0) n = 0 - n;
        if (n >= (int)w) n -= 2 * (1 + n - w);
        d += tG[m] * float(sour[y][n]);
                        }
      dest[y][k] = mmxround(s);
      dest[y][k+w2] = mmxroundTH(d);
  }
}
```

ROCCC kernel

- DWT transrow kernel @ 12 mul, 10 add = 22 ops

```
/* bior24.flt --
    float tH[9] =  {0.0234375, -0.0468750,-0.1250000,
                    0.2968750,  0.7031250, 0.2968750,
                   -0.1250000, -0.0468750, 0.0234375}
    float tG[3] = {-0.250, 0.5000, -0.250}
*/

L:  for (k2 = 0; k2 < w2; k2+=1) {  // w2 = image width/2
      destyH[k2] = tH[0]*soury[k2]   + tH[1]*soury[k2+1] + tH[2]*soury[k2+2]
                 + tH[3]*soury[k2+3] + tH[4]*soury[k2+4] + tH[5]*soury[k2+5]
                 + tH[6]*soury[k2+6] + tH[7]*soury[k2+7] + tH[8]*soury[k2+8];
     destyG[k2] = tG[0]*soury[k2] + tG[1]*soury[k2+1] + tG[2]*soury[k2+2];
    }
}
```

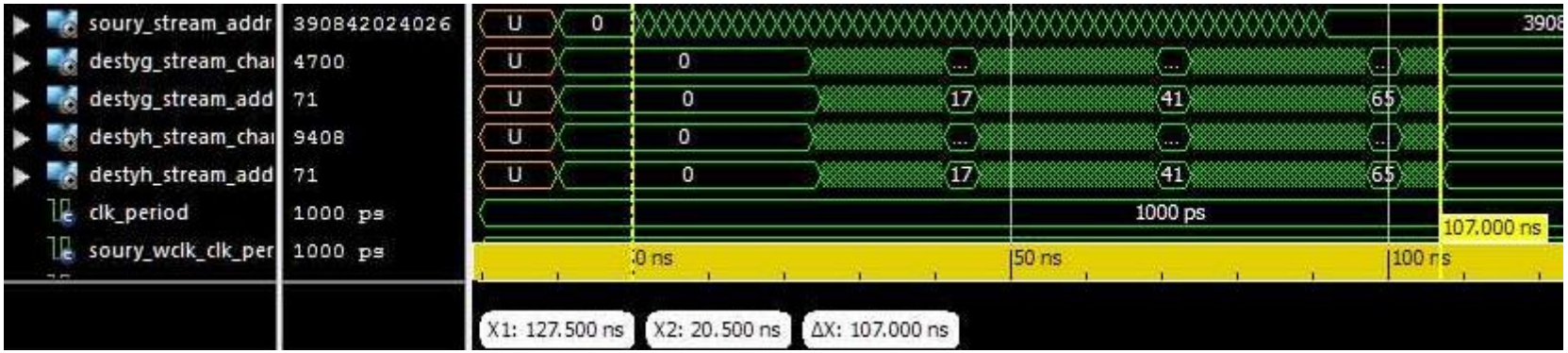
- Kernel → VHDL → testbench

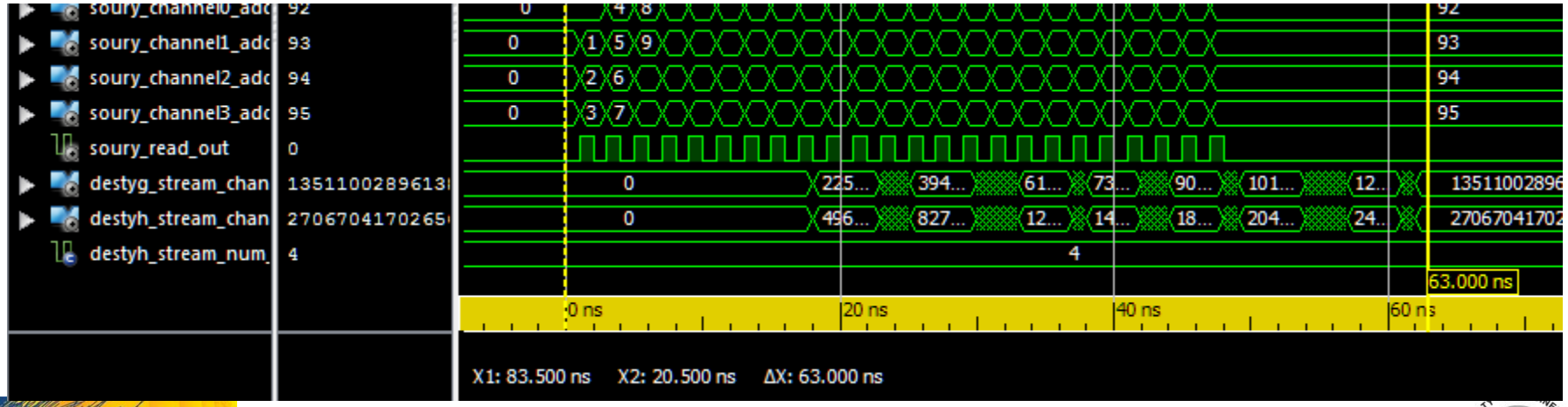
FPGA – ROCCC performance analysis

- Single stream: 194 cycles ⇒ 22 ops / (194 * 8 ns) = 1.42 Gop/s

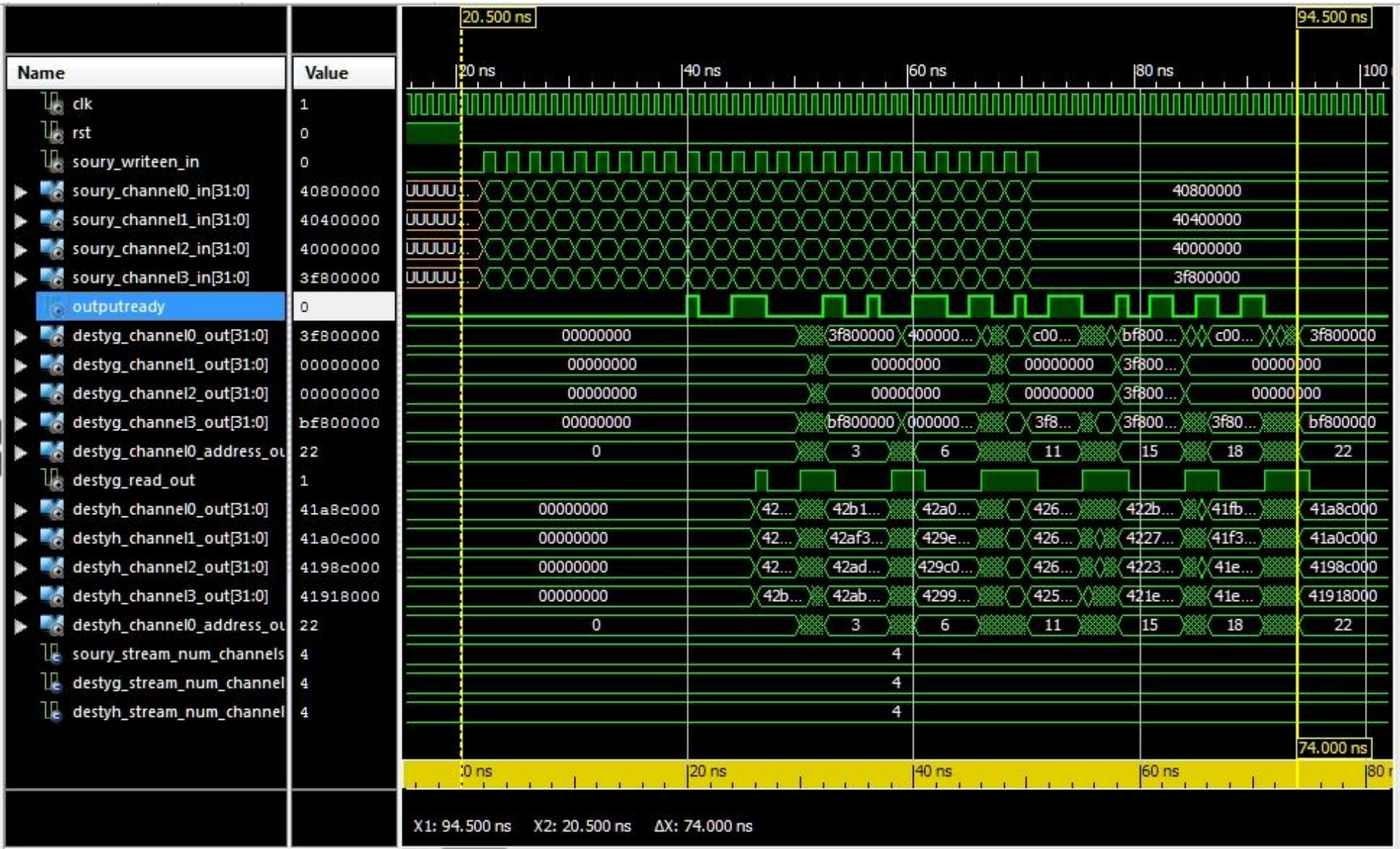
FPGA – ROCCC performance analysis

- 2 streams: 107 cycles; 4 streams: 63 cycles





4 streams: 74 cy@ 74MHz = 2.2 GFlops
latency = 30cy@13ns = 390 ns



ROCCC ↔ Handmade ↔ GPU

- **Power of C→ VHDL synthesis**
- ROCCC image kernel: 2.2 GFlops @ 74 MHz, Virtex 4
- Manual BLAS* 2 fpgas: 7.8 GFlops @ 140 MHz, StratixII 60
- GPU image kernel: 5.4 Gflops @ 1300 MHz, GT330

- Improvements: use a pipelined multiplier → lower cycle time

^S. Rousseaux, et al. A High Performance FPGA-based accelerator for BLAS library Implementation, Proc. Reconfigurable Systems Summer Institute, 2009.

Conclusion

- ROCCC toolchain greatly facilitates HPC on FPGA
- Good performance, low power
- Areas for improvement:
  - Enhanced Address Generator to account for more complex index expressions*
  - Host-FPGA bandwidth
- Areas of application?
  - Recognize LHF of hardware kernels
  - Optimize data locality using reuse distance analysis
  - Think spatial, parallel and pipelined

*D'Hollander, Beyls: High Performance Computing with FPGAs, High Speed and Large Scale Scientific Computing, 2009

?