Can FPGAs compete with GPUs?

John Romein, Bram Veenboer
Outline

- explain FPGA
  - hardware
- FPGA vs. GPU
  - programming models (OpenCL)
  - case studies
    - matrix multiplication
    - radio-astronomical imaging
  - lessons learned
- answer the question in the title

analyze performance & energy efficiency
What is a Field-Programmable Gate Array (FPGA)?

- configurable processor
- collection of
  - multipliers / adders
  - registers
  - memories
  - logic
  - transceivers
  - clocks
  - interconnect
What is an FPGA program?

- connect elements
  - performs fixed function
- data-flow engine
- Hardware Description Language (HDL)
  - Verilog, VHDL
  - difficult
FPGA vs GPU

• FPGA advantages
  – high energy efficiency
  • no instruction decoding etc.
  • use/move as few bits as possible
  – configurable I/O
  • high bandwidth
  • e.g., many times 100 GbE

• GPU advantages
  – easier to program
  – more flexible
  – short compilation times
New Intel (Altera) FPGA technologies

1) high-level programming language (OpenCL)
2) hard Floating-Point Units
3) tight integration with CPU cores

➔ simple tasks: less programming effort than HDL
➔ allows complex HPC applications
FPGA ≠ GPU

• hardware
  - GPU: use hardware
  - FPGA: create hardware

• execution model
  - GPU: instructions
  - FPGA: data flow
Common language: OpenCL

• OpenCL
  − similar to CUDA
  − C + explicit parallelism + synchronization + software-managed cache
  − offload to GPU/FPGA

• GPU programs not suitable for FPGAs
  − FPGA: data-flow engine
FPGA: data-flow pipeline example

• create hardware for complex multiply-add

\[
\begin{align*}
C.\text{real} &+ = A.\text{real} \times B.\text{real}; \\
C.\text{real} &+ = -A.\text{imag} \times B.\text{imag}; \\
C.\text{imag} &+ = A.\text{real} \times B.\text{imag}; \\
C.\text{imag} &+ = A.\text{imag} \times B.\text{real}; 
\end{align*}
\]

• needs four FPUs

• new input enters every cycle
Local memory

• GPU: use memory, FPGA: create memory
  
  ```c
  float tmp[128] __attribute__((…));
  ```

• properties:
  - registers or memory blocks
  - #banks
  - bank width
  - bank address selection bits
  - #read ports
  - #write ports
  - single/double pumped
  - with/without arbiter
  - replication factor

• well designed program: few resources, stall-free
Running multiple kernels

- GPU: consecutively
- FPGA: concurrently
  - channels (OpenCL extension)
  
    channel float2 my_channel __attribute__((depth(256)));
  - requires less memory bandwidth
  - all kernels resident → must fit
Parallel constructs in OpenCL

- work groups, work items (CUDA: thread blocks, threads)
  - GPU: parallel
  - FPGA: default: one work item/cycle; not useful
- FPGA:
  - compiler auto-parallelizes whole kernel
  - #pragma unroll
    - create hardware for concurrent loop iterations
    - replicate kernels

```c
#pragma unroll
for (int i = 0; i < 3; i++)
    a[i] = i;
```
More GPU/FPGA differences

• code outside critical loop:
  − GPU: not an issue
  − FPGA: can use many resources

• Shift registers
  − FPGA: single cycle
  − GPU: expensive

```c
init_once();
for (int i = 0; i < 100000; i ++)
  do_many_times();
```

```c
for (int i = 256; i > 0; i --)
  a[i] = a[i - 1];
```
Resource optimizations

- compiler feedback
  - after minutes, not hours
- HTML-based reports
- indispensable tool
Frequency optimization

- Compiler (place & route) determines $F_{\text{max}}$
  - Unlike GPUs
- Full FPGA: longest path limits $F_{\text{max}}$

- HDL: fine-grained control
- OpenCL: one clock for full design

- FPGA: 450 MHz; BSP: 400 Mhz; 1 DSP: 350 Mhz
- Little compiler feedback

- Recompile with random seeds
- Compile kernels in isolation to find frequency limiters
  - Tedious, but useful
Applications

- dense matrix multiplication
- radio-astronomical imager
- Signal processing (FIR filter, FFT, correlations, beam forming)
Dense matrix multiplication

• complex<float>
• horizontal & vertical memory accesses (→ coalesce)
• reuse of data (→ cache)
Matrix multiplication on FPGA

- hierarchical approach
  - systolic array of Processing Elements (PE)
  - each PE computes 32x32 submatrix
Matrix multiplication performance

- Arria 10:
  - uses 89% of the DSPs, 40% on-chip memory
  - clock (288 MHz) at 64% of peak (450 Mhz)
  - nearly stall free

<table>
<thead>
<tr>
<th>Type</th>
<th>Device</th>
<th>Performance (TFlop/s)</th>
<th>Power (W)</th>
<th>Efficiency (GFlop/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Intel Arria 10</td>
<td>0.774</td>
<td>37</td>
<td>20.9</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Titan X (Pascal)</td>
<td>10.1</td>
<td>263</td>
<td>38.4</td>
</tr>
<tr>
<td>GPU</td>
<td>AMD Vega FE</td>
<td>9.73</td>
<td>265</td>
<td>36.7</td>
</tr>
</tbody>
</table>
Image-Domain Gridding for Radio-Astronomical Imaging

- see talk S9306 (Astronomical Imaging on GPUs)
Image-Domain Gridding algorithm

```c
#pragma parallel
for s = 1...S :
    complex<float> subgrid[P][N ×N ];
for i = 1...N ×N :
    float offset = compute_offset(s, i);
for t = 1...T :
    float index = compute_index(s, i, t);
for c = 1...C :
    float scale = scales[c];
    float phase = offset - (index × scale);
    complex<float> phasor = {cos(phase), sin(phase)};
    #pragma unroll
    for p = 1...P : // 4 polarizations
        complex<float> visibility = visibilities[t][c][p];
        subgrid[p][i] += cmul(phasor, visibility);

apply_aterm(subgrid);
apply_taper(subgrid);
apply_ifft(subgrid);
store(subgrid);
```
FPGA gridding design

- Create a dataflow network:
  - Use all available DSPs
  - Every DSP performs a useful computation every cycle
  - No device memory use (except kernel args)
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__attribute__((num_compute_units(NR_GRIDDERS)))
__kernel void gridder()
{
    int    gridder = get_compute_id(0);
    float8 subgrid[NR_PIXELS];

    for (unsigned short pixel = 0; pixel < NR_PIXELS; pixel++) {
        subgrid[pixel] = 0;
    }

    #pragma ivdep
    for (unsigned short vis_major = 0; vis_major < NR_VISIBILITIES; vis_major += UNROLL_FACTOR) {
        float8 visibilities[UNROLL_FACTOR] __attribute__((register));

        for (unsigned short vis_minor = 0; vis_minor < UNROLL_FACTOR; vis_minor++) {
            visibilities[vis_minor] = read_channel_intel(visibilities_channel[gridder]);
        }

        for (unsigned short pixel = 0; pixel < NR_PIXELS; pixel++) {
            float8 pixel_value = subgrid[pixel];
            float8 phasors     = read_channel_intel(phasors_channel[gridder]);  // { cos(phase), sin(phase) }

            #pragma unroll
            for (unsigned short vis_minor = 0; vis_minor < UNROLL_FACTOR; vis_minor++) {
                pixel_value.even += phasors[vis_minor] * visibilities[vis_minor].even + -phasors[vis_minor] * visibilities[vis_minor].odd;
                pixel_value.odd  += phasors[vis_minor] * visibilities[vis_minor].odd  +  phasors[vis_minor] * visibilities[vis_minor].even;
            }

            subgrid[pixel] = pixel_value;
        }
    }
}

for (unsigned short pixel = 0; pixel < NR_PIXELS; pixel++) {
    write_channel_intel(pixel_channel[gridder], subgrid[pixel]);
}
Sine/cosine optimization

- compiler-generated: 8 DSPs
- limited-precision lookup-table: 1 DSP
  - more DSPs available → replicate φ ± 4 20x
### FPGA resource usage + $F_{\text{max}}$

<table>
<thead>
<tr>
<th></th>
<th>ALUTs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
<th>MLABs</th>
<th>$\phi$</th>
<th>$F_{\text{max}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridding-ip</td>
<td>43%</td>
<td>31%</td>
<td>64%</td>
<td>1439 (95%)</td>
<td>71%</td>
<td>14</td>
<td>258</td>
</tr>
<tr>
<td>degridding-ip</td>
<td>47%</td>
<td>35%</td>
<td>72%</td>
<td>1441 (95%)</td>
<td>78%</td>
<td>14</td>
<td>254</td>
</tr>
<tr>
<td>gridding-lu</td>
<td>27%</td>
<td>32%</td>
<td>61%</td>
<td>1498 (99%)</td>
<td>57%</td>
<td>20</td>
<td>256</td>
</tr>
<tr>
<td>degridding-lu</td>
<td>33%</td>
<td>38%</td>
<td>73%</td>
<td>1503 (99%)</td>
<td>69%</td>
<td>20</td>
<td>253</td>
</tr>
</tbody>
</table>

- almost all of 1518 DSPs available used
- $F_{\text{max}} < 350$ Mhz
Experimental setup

- compare Intel Arria 10 FPGA to comparable CPU and GPU
- CPU and GPU implementations are both optimized

<table>
<thead>
<tr>
<th>Type</th>
<th>Device</th>
<th>#FPUs</th>
<th>Peak</th>
<th>Bandwidth</th>
<th>TDP</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2697v3</td>
<td>224</td>
<td>1.39 TFlop/s</td>
<td>68 GB/s</td>
<td>145W</td>
<td>28nm (TSMC)</td>
</tr>
<tr>
<td>FPGA</td>
<td>Nallatech 385A</td>
<td>1518</td>
<td>1.37 TFlop/s</td>
<td>34 GB/s</td>
<td>75W</td>
<td>20nm (TSMC)</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA GTX 750 Ti</td>
<td>640</td>
<td>1.39 TFlop/s</td>
<td>88 GB/s</td>
<td>60W</td>
<td>28nm (TSMC)</td>
</tr>
</tbody>
</table>
Throughput and energy-efficiency comparison

- **throughput**: number of visibilities processed per second (Mvisibilities/s)
- **energy-efficiency**: number of visibilities processed per Joule (MVisibilities/J)

• **similar peak performance, what causes the performance differences?**
Performance analysis

- **CPU**: sin/cos in software takes 80% of total runtime
- **GPU**: sin/cos overlapped with other computations
- **FPGA**: sin/cos overlapped, but cannot use all DSPs for FMAs
FPGAs vs GPUs: lessons learned (1/3)

• dataflow vs. imperative
  - rethink your algorithm
  - different program code
• on FPGAs:
  - frequent use of OpenCL extensions
  - think about resource usage, occupancy, timing
  - less use of off-chip memory
• parallelism:
  - both: kernel replication and vectorization
  - FPGA: pipelining and loop unrolling
FPGAs vs GPUs: lessons learned (2/3)

- FPGA \neq GPU, yet: same optimizations …
  - exploit parallelism
  - maximize FPU utilization
    - hide latency
  - optimize memory performance
    - hide latency \rightarrow prefetch
    - maximize bandwidth \rightarrow avoid bank conflicts, unit-stride access (coalescing)
    - reuse data (caching)
- … but achieved in very different ways!
- + architecture-specific optimizations
FPGAs vs GPUs: lessons learned (3/3)

• much simpler than HDL
  - FPGAs accessible to wider audience
    • long learning curve
  - small performance penalty

• yet not as easy as GPUs
  - distribute resources in complex dataflow
  - optimize for high clock

• tools still maturing
Current/future work

- Stratix 10 versus Volta/Turing
  - Stratix 10 imager port not trivial ← different optimizations for new routing architecture

- 100 GbE FPGA support
  - send/receive UDP packets in OpenCL
  - BSP firmware development

- Streaming data applications
  - Signal processing (filtering, correlations, beam forming, ...)

- OpenCL FFT library
Conclusions

• complex applications on FPGAs now possible
  – high-level language
  – hard FPUs
  – tight integration with CPUs

• GPUs vs FPGAs
  – 1 language; no (performance) portability
  – very different architectures
  – yet many optimizations similar
So can FPGAs compete with GPUs?

- depends on application
  - compute → GPUs
  - energy efficiency → GPUs
  - I/O → FPGA
  - flexibility → CPU/GPU
  - programming ease → CPU, then GPU, then FPGA

FPGA not far behind; CPU way behind
Acknowledgements

• This work was funded by
  - Netherlands eScience Center (Triple-A 2)
  - EU H2020 FETHPC (DEEP-EST, Grant Agreement nr. 754304)
  - NWO Netherlands Foundation for Scientific Research (DAS-5)

• Other people
  - Suleyman Demirsoy (Intel), Johan Hiddink (NLeSC), Atze v.d. Ploeg (NLeSC), Daniël v.d. Schuur (ASTRON), Merijn Verstraaten (NLeSC), Ben v. Werkhoven (NLeSC)