# FPGA acceleration for HPC supercapacitor simulations

Charles Prouveur

Work done in collaboration with M. Haefele, T. Kenter & N. Voss

# What is an FPGA ?

- A chip historically used in embedded systems
- Can be found in accelerator cards nowadays


Xilinx U200


Intel Stratix10


Intel Agilex 7 (F)

# FPGA acceleration

- Why is it different ?

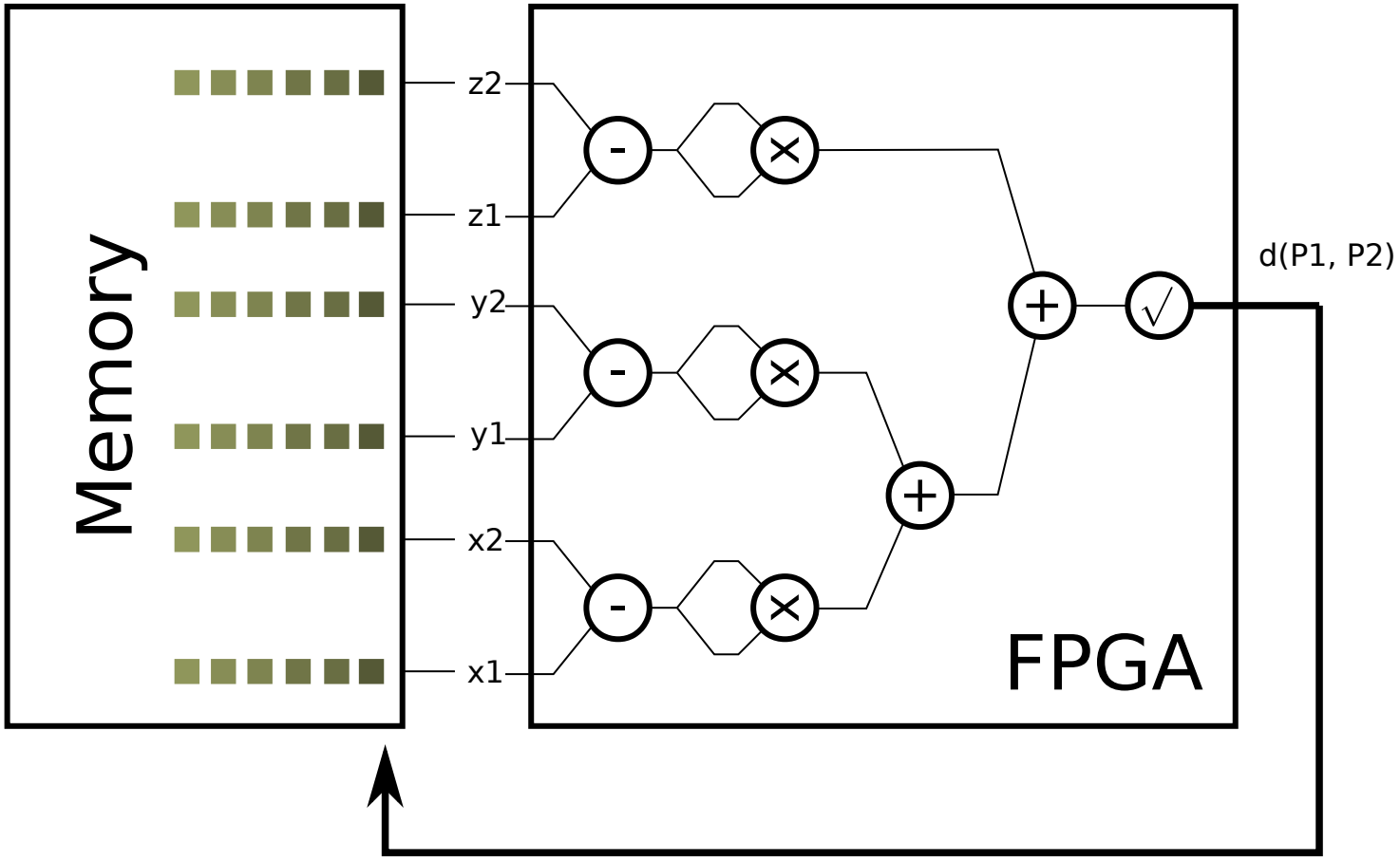
  Physical design vs instructions executions

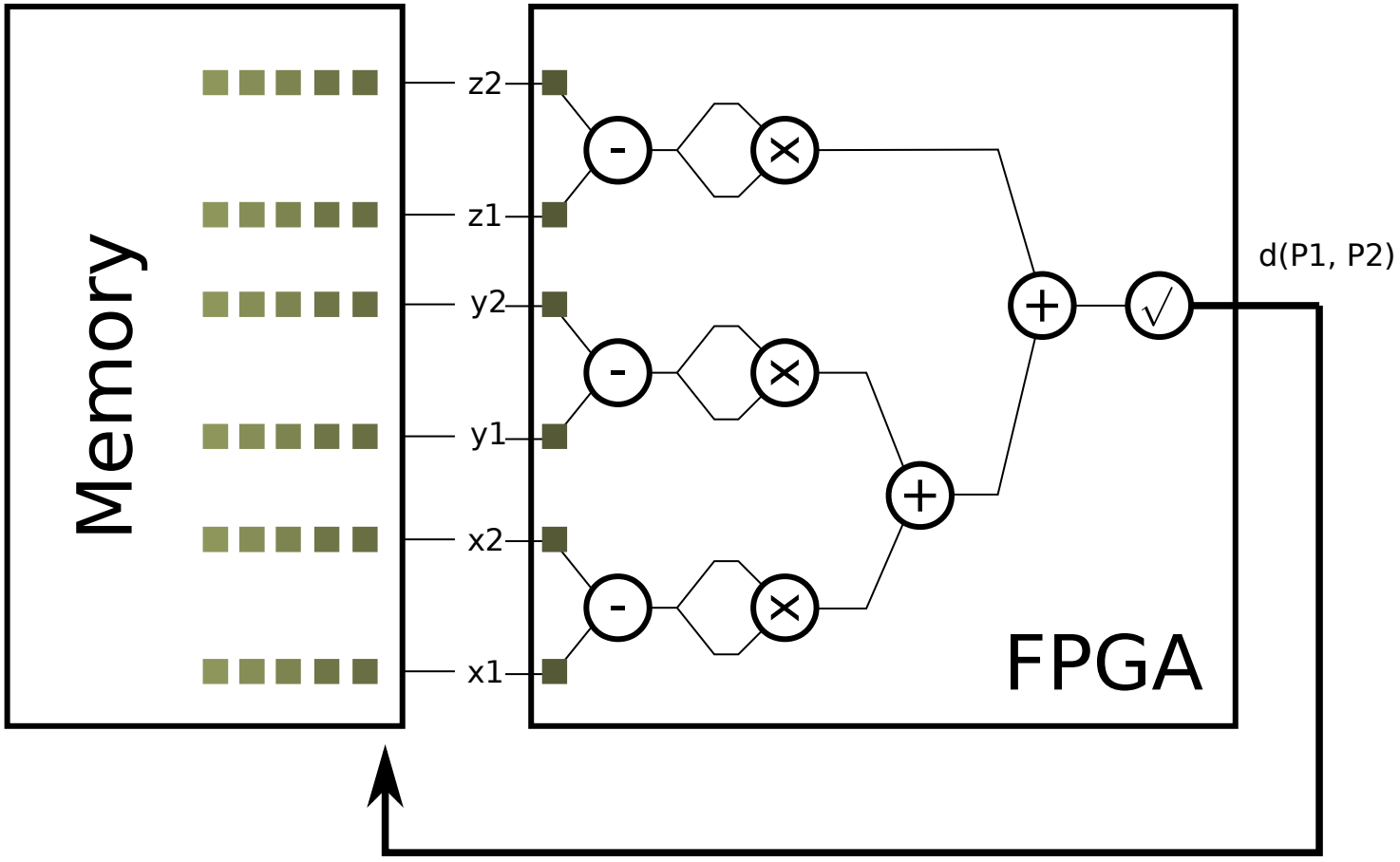
- How does it work ?

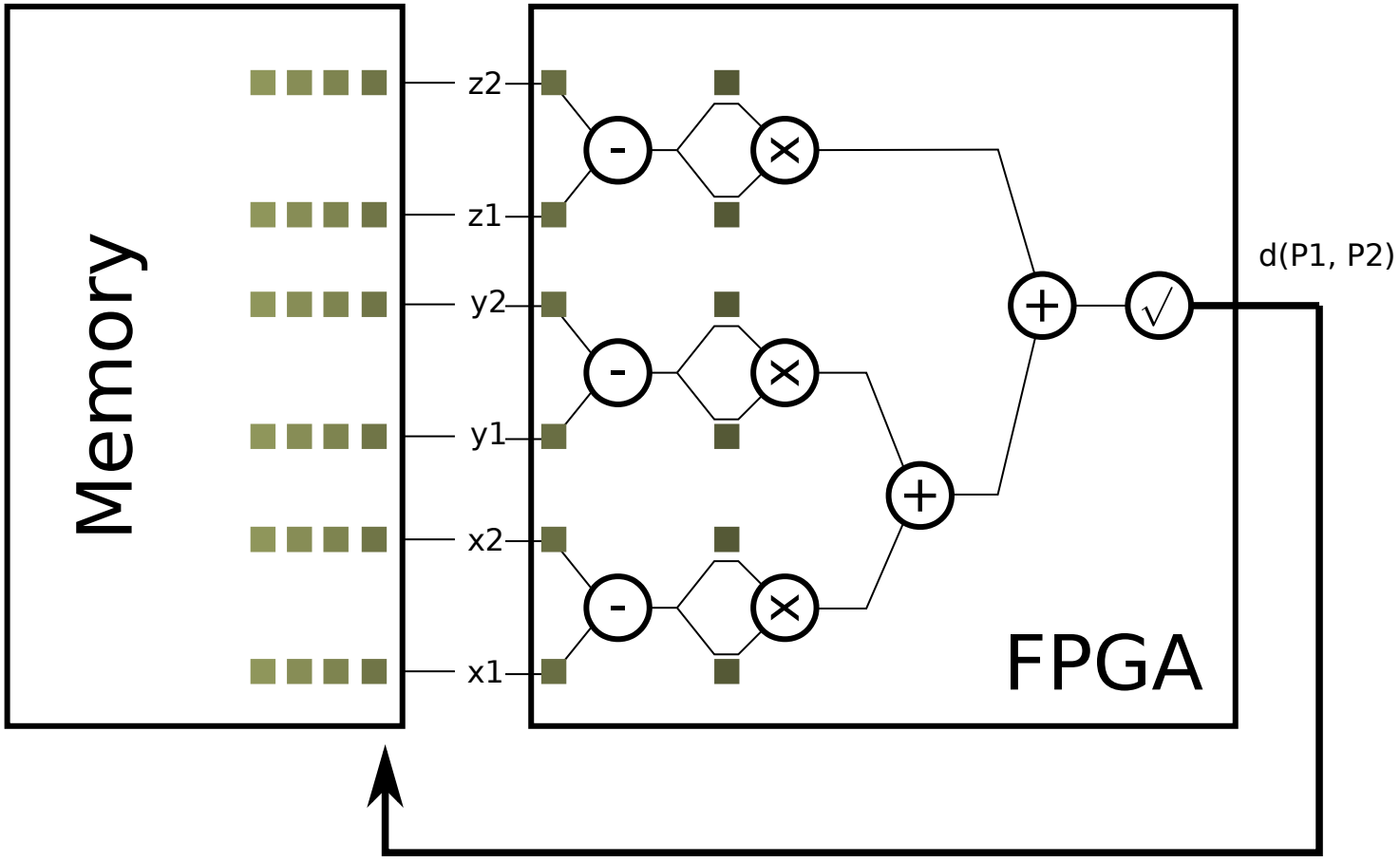
  Data is streamed from DRAM into pipelines on the chip

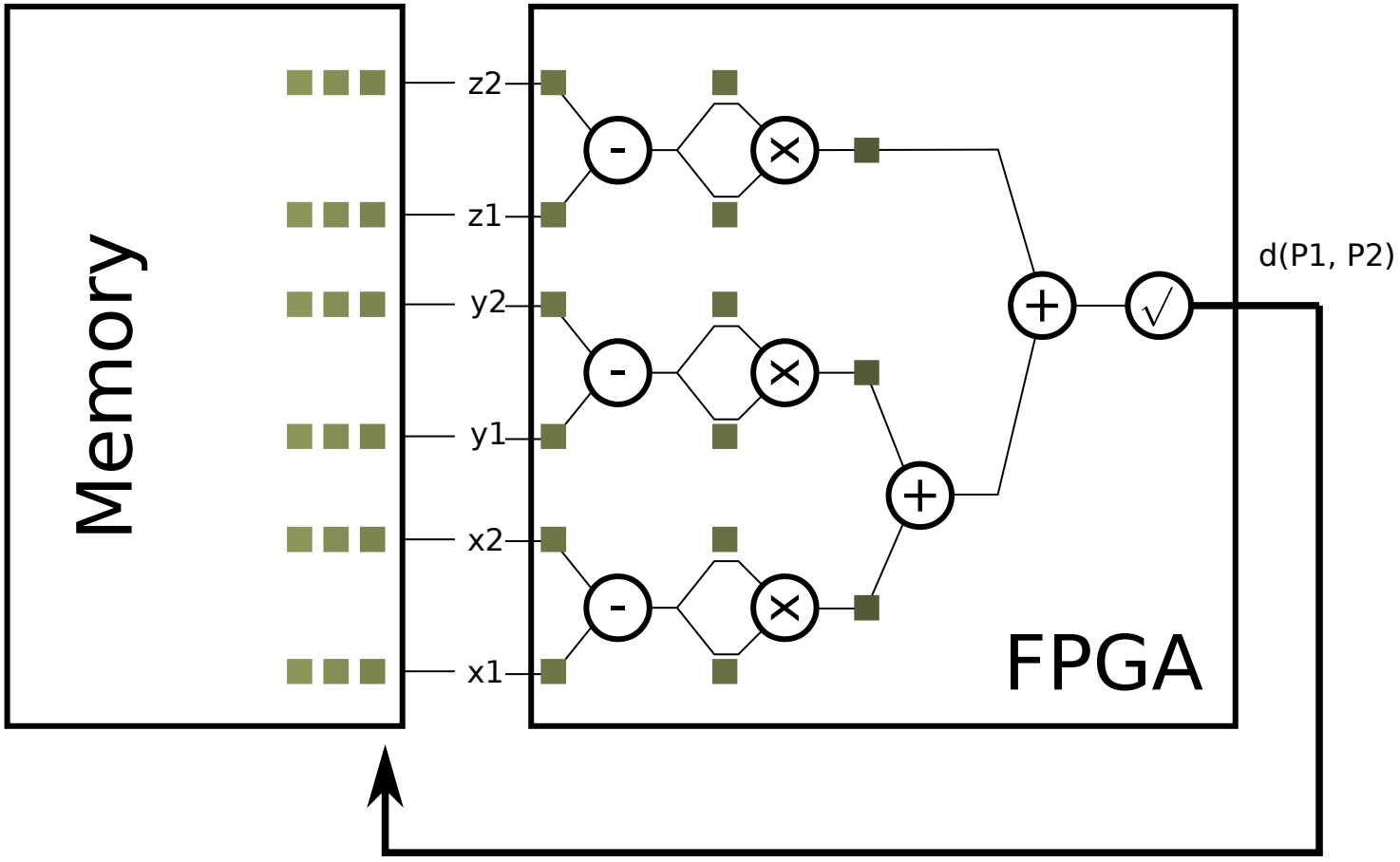
- What is the point ?

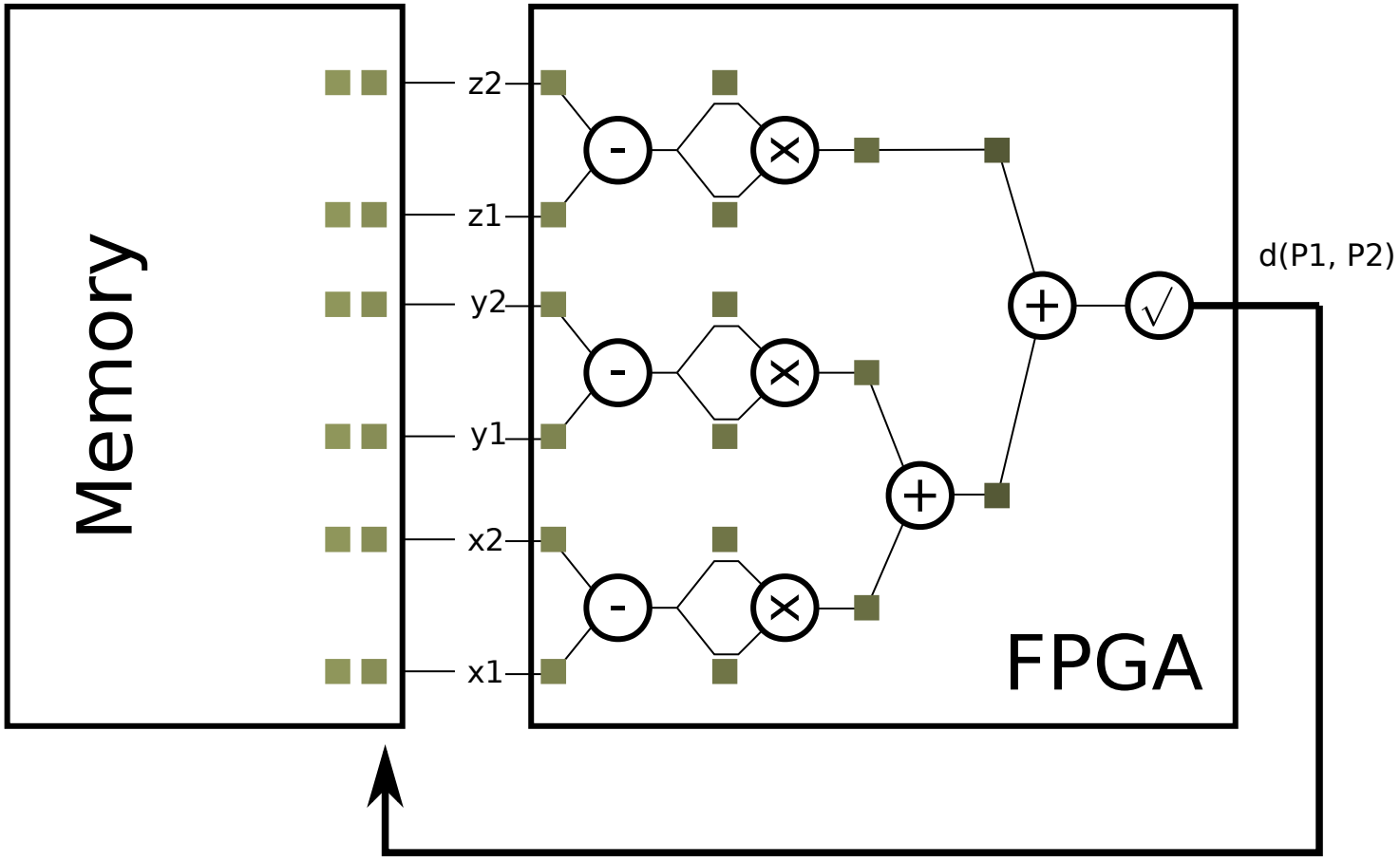
  FPGAs’ TDP’s ~ 5-6 x less than GPUs’


Memory
z2
z1
y2
y1
x2
x1
-
-
-
×
×
×
+
+
√
d(P1, P2)
FPGA


Memory
z2
z1
y2
y1
x2
x1
-
-
-
×
×
×
+
+
√
d(P1, P2)
FPGA


Memory
z2
z1
y2
y1
x2
x1
-
-
-
×
×
×
+
+
√
d(P1, P2)
FPGA

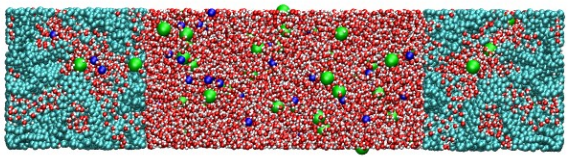

Memory
z2
z1
y2
y1
x2
x1
-
-
-
×
×
×
+
+
√
d(P1, P2)
FPGA


Memory
z2
z1
y2
y1
x2
x1
-
-
-
×
×
×
+
+
√
d(P1, P2)
FPGA

# FPGA parallelism

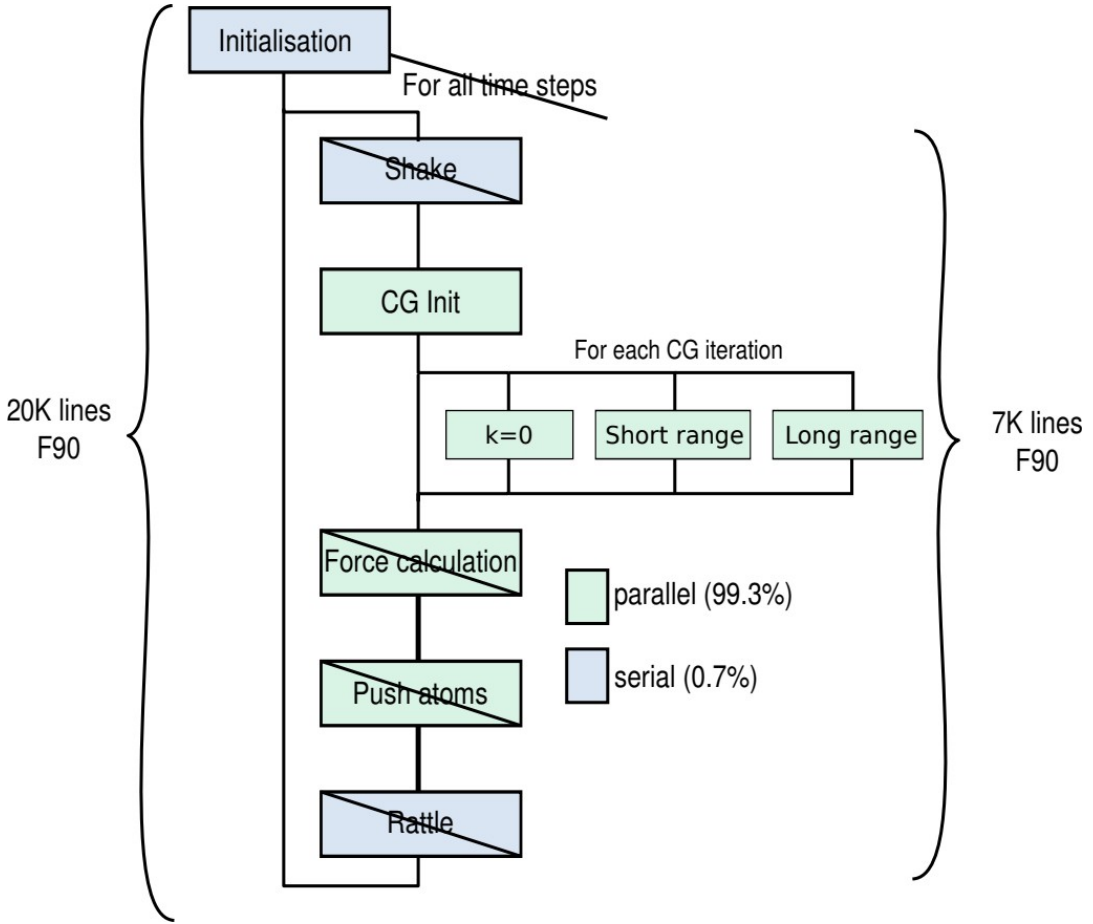
- Acceleration is proportional to the number of pipes
- BUT : also proportionnal to the length of the pipe(!)
- Computing time = number of operations / (frequency x number of pipes x length of the pipe) + latency

# MetalWalls

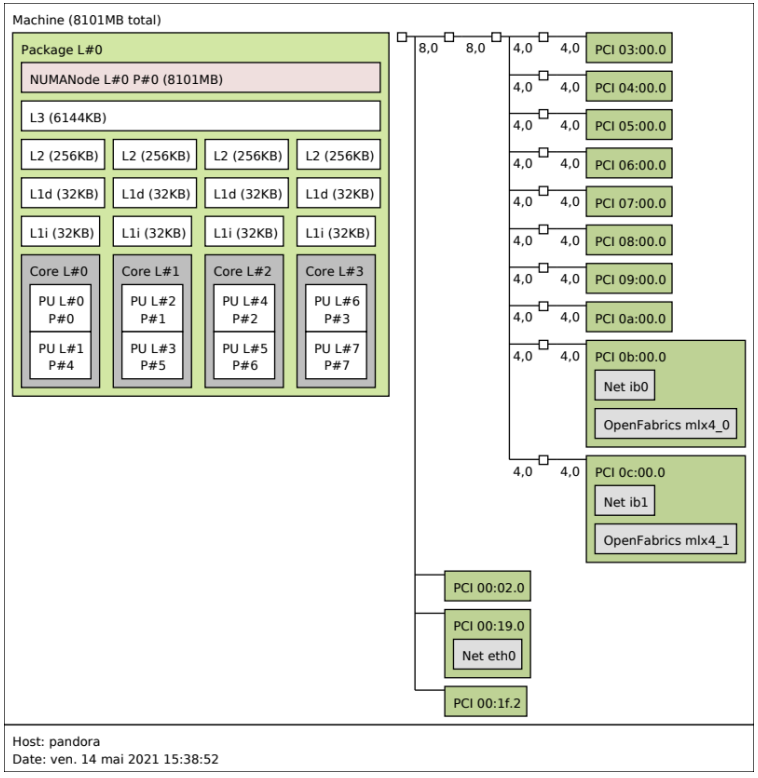


- Molecular dynamic production code used by Sorbonne university researchers to simulate electrochemical systems such as «supercapacitors»
- Fortran 90 base code, parallelised with MPI
- Most of the computing time = electrostatic potential computing
- Computing efficiency published : model running during several weeks on 512 cores while maintaining a parallel efficiciency above 75 %
- Currently using CPU and GPU implementations (OpenAcc)

# Miniapp extracted from the production code



# Target device on Jumax at Juelich supercomputing center



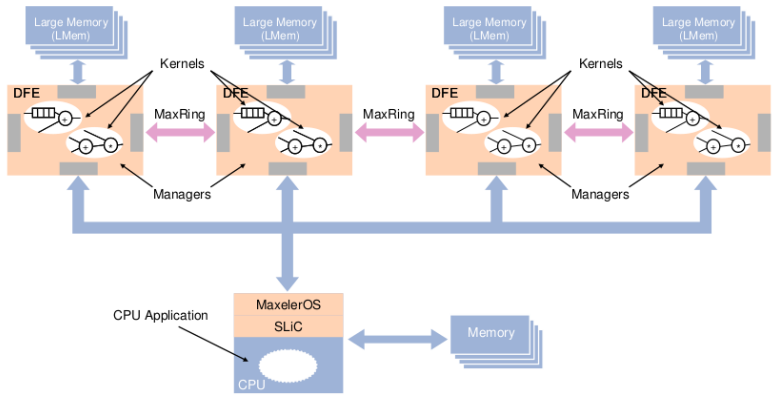




*Figure 1:* Overview of a Maxeler acceleration system

- CPU host : AMD EPYC 7601
- 8 nodes Xilinx VU9P on one blade as target devices
- 8x PCIe 2.0 lanes → 4.0 GB/s for all nodes
- Each node has three 16-GB DDR4 Dual In-Line Memory Modules (DIMMs), which provide a theoretical peak bandwidth of 15 GB/s each

# The FPGA used here

- Max5C is a U200 with one less DIMM
- 3 SLR : the chip is divided in 3 Super Logic Region connected by solders (very low bandwidth)
- SLR communication should be avoided as much as possible
- Design done using Maxj with Maxeler toolchain

| | MAX5C | Alveo U200 |
|---|---|---|
| FPGA | VU9P | VU9P |
| SLRs | 3 | 3 |
| LUTs (k) | 1,182 | 1,182 |
| FFs (k) | 2,364 | 2,364 |
| DSPs | 6,840 | 6,840 |
| BRAM18s | 4,320 | 4,320 |
| URAMs | 960 | 960 |
| On Chip Memory Capacity | 43.2 MByte | 43.2 MByte |
| DDR DIMMs | 3 | 4 |
| DDR Capacity per DIMM | 16 GB | 16 GB |
| Supported DDR Frequencies (MHz) | 933, 1066, 1200 | 1200 |
| DIMM to SLR Mapping | DIMM_0 ->SLR0<br>DIMM_1 ->SLR1<br>DIMM_2 ->SLR2 | DIMM_0 ->SLR0<br>DIMM_1 ->SLR1<br>DIMM_2 ->SLR1<br>DIMM_3 ->SLR2 |
| PCIe Placement | SLR1 | SLR0 |
| PCIe | PCIe Gen 2 x8 | PCIe Gen 2 x8 |
| Networking | 1 x 100 GBit/s | 2 x 100GBit/s |
| Networking Placement | SLR2 | SLR2 |

# Designing all kernels on one FPGA

- Due to hardware contraints, each kernel is put into one SLR (Super Logic Region)
- For simplicity, the conjugate gradient is computed on the host
- We want the highest frequency possible
- We also want the biggest number of separate pipelines
- All kernels work synchronously
- Use as much as possible the device’s DRAM to reduce communications

# Challenges

- Limited ressources
- More logic available → higher design frequency likely
- More pipelines → less logic available
- Using DRAM makes meeting timings harder

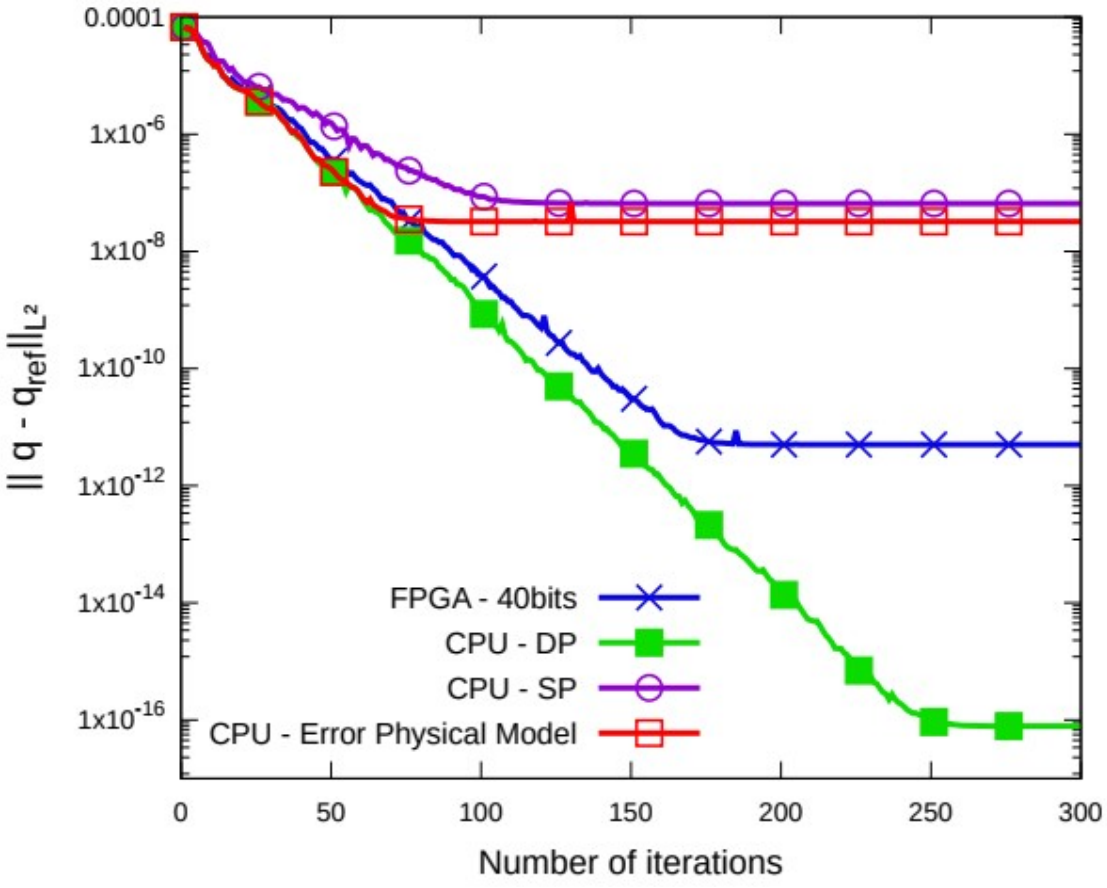
  → harder compilation, ie we are less likely to achieve high frequency with as many pipelines

# Balancing the kernels

- Need for synchronicity → balance needed
- Theoretical time for each sequential kernel is known : $N^2$ / freq , $N^2/2$ / freq and N x M / freq
- Balance is case dependent (i.e. : dependent on N and M)
- For the production test case considered here, the balance is (8,4,1) :

  8 pipes for the kernel in $N^2$, 4 pipes for the kernel in $N^2/2$ and one pipe for the kernel in N x M

# Numerical accuracy analysis




We can save ressources but there is a catch : the number of iterations to converge increases

# Ressources usage of the multiple kernels designs

| Design Name | 64 bits Design | 40 bits Design | Final design |
|---|---|---|---|
| Design frequency (MHz ) | 200 | 200 | 300 |
| Pipes $(U_{\mathrm{lr},0}, U_{\mathrm{sr}}, U_{\mathrm{lr},+})$ | (8,4,1) | (16,8,2) | (32,16,4) |
| Logic (LUTs & FFs) | 27.7% | 33.4% | 44.6 % |
| DSPs | 33.42% | 29.52% | 53.3% |
| On-chip Mem | 22.7% | 20.3% | 28.8% |

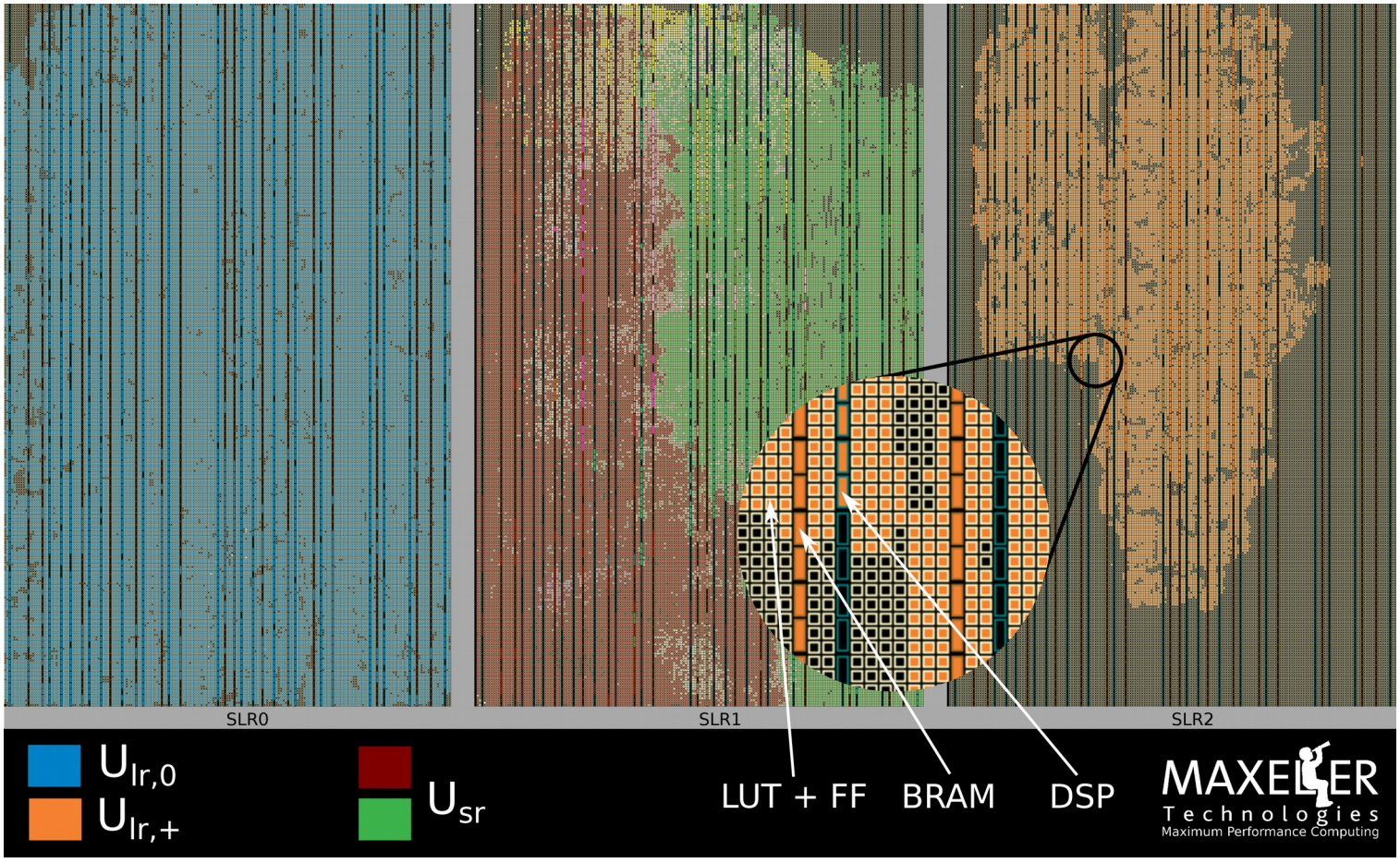
DSP limited in the U_lr,0 kernel (87 % DSPs used in its SLR)

# Design using the device’s DRAM

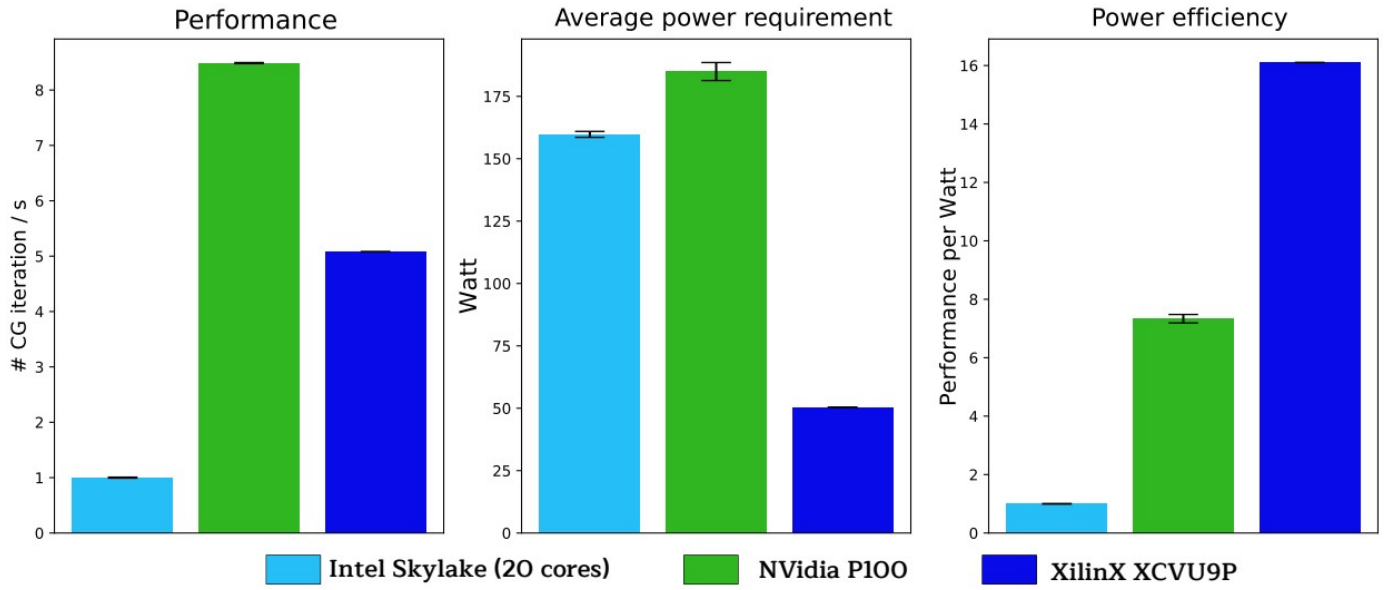
- The previous design does not use the DRAM
- Q has to be sent every iteration from and to the FPGA
- (x,y,z) they can be stored
- Using LMEM makes the design harder to compile

  → have to make concessions
- Best design is (24,12,3) with a frequency of 260 MHz

# Airview of the compiled final design with all kernels on the FPGA



# Results



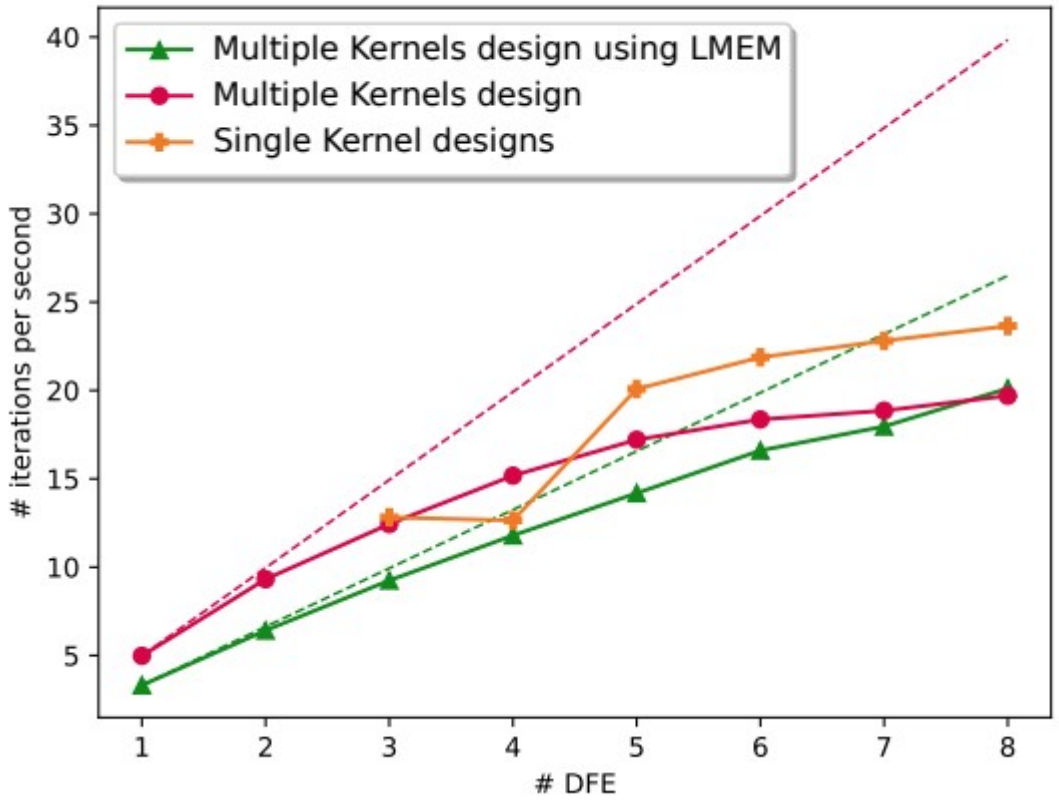

Measured FPGA power efficiency is twice the GPU's

# Ressources usage of the single kernel designs

| Design Name | Design $U_{\mathrm{lr},0}$ | Design $U_{\mathrm{sr}}$ | Design $U_{\mathrm{lr},+}$ |
|---|---|---|---|
| Design frequency (MHz ) | 300 | 300 | 300 |
| Total number of pipes | 96 | 48 | 42 |
| Logic (LUTs & FFs) | 51.9% | 63.3% | 62.8% |
| DSPs | 87.2% | 55.4% | 83.5% |
| On-chip Mem | 27.2% | 25.8% | 38.4% |

- DSP limited in two kernels and Logic limited in one kernel
- Adapting the balance to a number of FPGAs allocated per kernel

# Speedup using multiple FPGAs



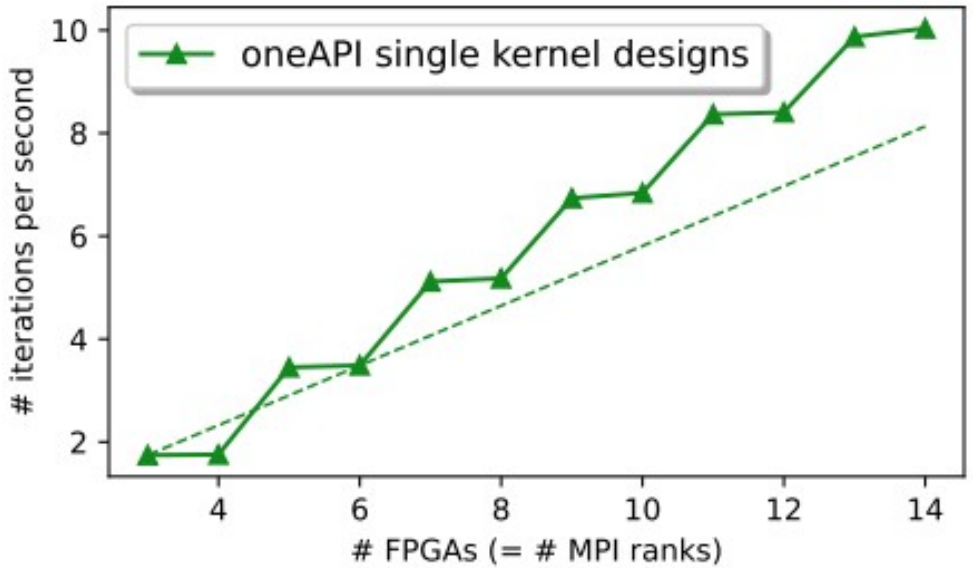
# FPGA targeting with OneAPI


- Development done in C++
- Easy to use and to make you first design targeting Intel FPGAs
- Another hardware and another toochain
- Long term support is more likely

# OneAPI implementation of Metalwalls

- Targeting Stratix 10 at Paderborn supercomputing center
- Poor resources usage (max 8 pipes @ 400 Mhz)
- Mixed precision did not save as many resources (if any)
- Obtaining proper throughput was a big challenge
- Single kernel designs scaling done up to 14 nodes (we could only use one out of two FPGAs on each node)
- Higher TDP with the Stratix 10

# Speedup at PC²




Good scaling, but ~10x worse than with maxeler design
(but 1 month of development vs ~2 years)

# Conclusion

- We Implemented Metalwalls kernels on FPGA using Maxeler tools
- Our FPGA implementation showed 2x better performance per watt than a GPU of similar silicon technology and even better against skylake CPU with maxeler design
- Multiple FPGAs performance bottlenecked by old interconnect technology but achieved nonetheless
- OneAPI implementation : promising technology, better development environment, not mature enough for HPC yet but developping fast

Thank you for your attention

# Annexes

# Original code vs Maxcompiler

```
!$acc data present(xyz_atoms(:,3), q_elec(:), V(:))

! Blocks on the diagonal compute only half of the pair interactions
do itype = 1, num_elec_types
   count_n = electrodes(itype)%count
   offset_n = electrodes(itype)%offset
   do jtype = 1, itype-1
      count_m = electrodes(jtype)%count
      offset_m = electrodes(jtype)%offset

      ! Number of blocks with full interactions (below the diagonal
      num_block_full = localwork%pair_atom2atom_num_block_full_local(itype,jtype)
      iblock_offset = localwork%pair_atom2atom_full_iblock_offset(itype,jtype)
      do iblock = 1, num_block_full
         call update_other_block_boundaries(iblock_offset+iblock, &
              offset_n, count_n, offset_m, count_m, &
              istart, iend, jstart, jend)
        !$acc parallel loop  private(zij, zijsq, pot_ij, vi)
        do i = istart, iend
           vi = 0.0_wp
           !$acc loop reduction(+:vi)
           do j = jstart, jend
              zij = xyz_atoms(j,3) - xyz_atoms(i,3)
              zijsq = zij*zij
              pot_ij = volfactor * (sqrpialpha*exp(-zijsq*alphasq) + pi*zij*erf(zij*alpha))
              !$acc atomic update
              V(j) = V(j) - q_elec(i) * pot_ij
              vi = vi + q_elec(j) * pot_ij
           end do
           !$acc end loop
           V(i) = V(i) - vi
        end do
        !$acc end parallel loop
      end do
   end do

   num_block_diag = localwork%pair_atom2atom_num_block_diag_local(itype)
   iblock_offset = localwork%pair_atom2atom_diag_iblock_offset(itype)

   do iblock = 1, num_block_diag
      call update_diag_block_boundaries(iblock_offset+iblock, offset_n, count_n, istart, iend)
      !$acc parallel loop  private(zij, zijsq, pot_ij, vi)
      do i = istart, iend
         vi = 0.0_wp
         !$acc loop reduction(+:vi)
         do j = istart, iend
            zij = xyz_atoms(j,3) - xyz_atoms(i,3)
            zijsq = zij*zij
            pot_ij = volfactor * (sqrpialpha*exp(-zijsq*alphasq) + pi*zij*erf(zij*alpha))
            vi = vi + q_elec(j) * pot_ij
         end do
         !$acc end loop
         V(i) = V(i) - vi
      end do
      !$acc end loop
   end do

   ! Number of blocks with full interactions (below the diagonal
   num_block_full = localwork%pair_atom2atom_num_block_full_local(itype,itype)
```

```
    DFEVector<DFEStruct> xyz = io.input("xyz", xyztypeVec,load);

    DFEVectorType<DFEVar> vectorType = new DFEVectorType<DFEVar>(dfeFloat(11, 53), numPipes);
    DFEVector<DFEVar> q = io.input("q", vectorType,load);

    ArrayList<Memory<DFEVar>> zRam = new ArrayList<Memory<DFEVar>>();
    ArrayList<Memory<DFEVar>> qRam = new ArrayList<Memory<DFEVar>>();
    for(int i=0; i<numPipes; i++) {
        zRam.add(mem.alloc(scalar, maxParticlesPerPipe));
        qRam.add(mem.alloc(scalar, maxParticlesPerPipe));
    }

    for(int i=0; i<numPipes; i++) {
        DFEVar Address_0 = counter_atoms.getCount().cast(dfeUInt(MathUtils.bitsToAddress(maxParticlesPerPipe)));
        DFEVar q_j = q[i].cast(scalar);
        DFEVar z_j = (DFEVar) xyz[i]["z"].cast(scalar);
        zRam[i].write(Address_0, z_j, load);
        qRam[i].write(Address_0, q_j, load);
    }

    DFEVar[] k0sum = new DFEVar[numPipes];
    // Parallel execution
    for(int i=0; i<numPipes; i++) {
        DFEVar jCount = i + counter_atoms.getCount()*numPipes;
        DFEVar Address = counter_atoms.getCount().cast(dfeUInt(MathUtils.bitsToAddress(maxParticlesPerPipe)));
        DFEVar zj = load ? (DFEVar) xyz[i]["z"].cast(scalar) : zRam[i].read(Address);
        DFEVar qj = load ? q[i].cast(scalar) : qRam[i].read(Address);

        // Compute the potential
        DFEVar zij = zj - zi;
        DFEVar zijSq = zij * zij;
        DFEVar erfZij = ErrorFunction.erf(alpha * zij);
        DFEVar expZij = KernelMath.exp(-alphasq * zijSq);
        DFEVar vij = counter_zi.getCount().cast(dfeUInt(32)) < numWall_without_padding &
        jCount < numWall_without_padding ?
        qj * ((sqrpialpha * expZij) + (pi*zij*erfZij)) :
        constant.var(0).cast(scalar) ;

        k0sum[i] = FloatingPointAccumulator.accumulateFixedLength(vij.cast(dfeFloat(11, 53)), constant.var(true), numWallPerPipe.cast(dfeUInt(32)), true);
    }

    // sum contributions from each pipe
    DFEVar k0pot =  k0PotFactor * FloatingPointMultiAdder.add(k0sum);

    // Output
    io.output("k0pot", k0pot, dfeFloat(11, 53),counter_atoms.getWrap());
}
```

# OneAPI :

```
void kernel_k0( int num_atoms, double alpha, double alphasq, double sqrpialpha, double k0PotFactor,
        double *z, double *q, double *k0 ){

        //"first_index_i", (num_start_slr + num_slr_in_fpga) * Nperslr);

        double q_loc[num_max];
        double z_loc[num_max];
        for (int i = 0; i < num_atoms; i++) {
            z_loc[i] = z[i];
            q_loc[i] = q[i];
        }

        for (int i = 0; i < num_atoms; i++) { // MPI division here
            double zi = z_loc[i];
            double s = 0.0;
            for (int j = 0; j < num_atoms; j++) { // pipe parallelization here
                double zij = z_loc[j] - zi;
                double zijsq = zij*zij;
                double pot_ij = sqrpialpha*exp(-zijsq*alphasq) + M_PI*zij*erf(zij*alpha);
                s += - q_loc[j] * pot_ij;
            }
            k0[i] = s * k0PotFactor;
        }
}
```

home > mdlsuser > prog > gitlab > OneAPI > tobias_branch > MetalWalls_One > Kernel_k0.cpp > ...

```
#ifndef K0_INNER_U
#define K0_INNER_U 16
#endif

#ifndef K0_OUTER_U
#define K0_OUTER_U 2
#endif

#ifndef K0_READ_U
#define K0_READ_U 16
#endif

const int num_max = 43008; //8*1024;
class k0Kernel;

/// k0 contribution
/// -----------------------
void kernel_k0(const int num_atoms, const double alpha,
        const double alphasq, const double sqrpialpha,
        const double k0PotFactor,
        double *z, double *q, double *k0 ){

        Real q_loc[num_max];
        Real z_loc[num_max];
        #pragma unroll K0_READ_U
        for (int i = 0; i < num_atoms; ++i) {
            z_loc[i] = z[i];//fpga_reg(z[i]);
            q_loc[i] = q[i];//fpga_reg(q[i]);
        }
        for (int ib = 0; ib < num_atoms; ib+=K0_OUTER_U) { // MPI division here
            Real zi[K0_OUTER_U];
            double s[K0_OUTER_U];
            #pragma unroll
            for (int ii = 0; ii < K0_OUTER_U; ii++) {
                zi[ii] = z_loc[ib+ii];//fpga_reg(z_loc[ib+ii]);
                s[ii] = 0.0;
            }
            [[intel::initiation_interval(12)]] // 12 seems perfect
            for (int jb = 0; jb < num_atoms; jb+=K0_INNER_U) {
                #pragma unroll
                for (int ii = 0; ii < K0_OUTER_U; ii++) {
                    Real s_block = 0.0;
                    #pragma unroll
                    for (int jj = 0; jj < K0_INNER_U; jj++) {
                        Real zij = z_loc[jb+jj]-zi[ii];//fpga_reg(z_loc[jb+jj]) - fpga_reg(zi[ii]);
                        Real zijsq = zij*zij;
                        //Real c = fpga_reg(q_loc[jb+jj]) * (sqrpialpha*my_exp(-zijsq*(Real)alphasq) + M_PI*zij*my_erf(
                        Real c = q_loc[jb+jj] * (sqrpialpha*my_exp(-zijsq*(Real)alphasq) + M_PI*zij*my_erf(zij*alpha));
                        s_block += c;
                    }
                    s[ii] -= (double) s_block;
                }
            }
            #pragma unroll
            for (int ii = 0; ii < K0_OUTER_U; ii++) {
                k0[ib+ii] = s[ii] * k0PotFactor;
            }
        }
}
```

**Table 4: [LUTs, DSPs] usage of basic arithmetic functions based on oneAPI reports and Maxeler documentation.**

| Tool/Target | oneAPI/Stratix 10 | | Maxeler/VU9P | |
|---|---|---|---|---|
| Data Type | double | ap_float<10,35> | double | float<8,32> |
| Multiplication | [3253, 14] | [ 504, 1 ] | [132 , 7 ] | [ 427, 0 ] |
| Addition | [3484, 18] | [1696, 5 ] | [582 , 3 ] | [ 95, 4 ] |
| Division | [6034, 50] | [1864, 21 ] | [3135, 0 ] | [1247, 0 ] |
| exp() | [5504, 20] | [2218, 9 ] | [1290, 38] | [ 555, 16] |
| sqrt() | [2613, 28] | [1029, 6.5] | [1653, 0 ] | [ 673, 0 ] |
| fused sin/cos | [7548, 43] | [7548, 43 ]? | | |
| sin() | | | [1261, 28] | [ 771, 10] |