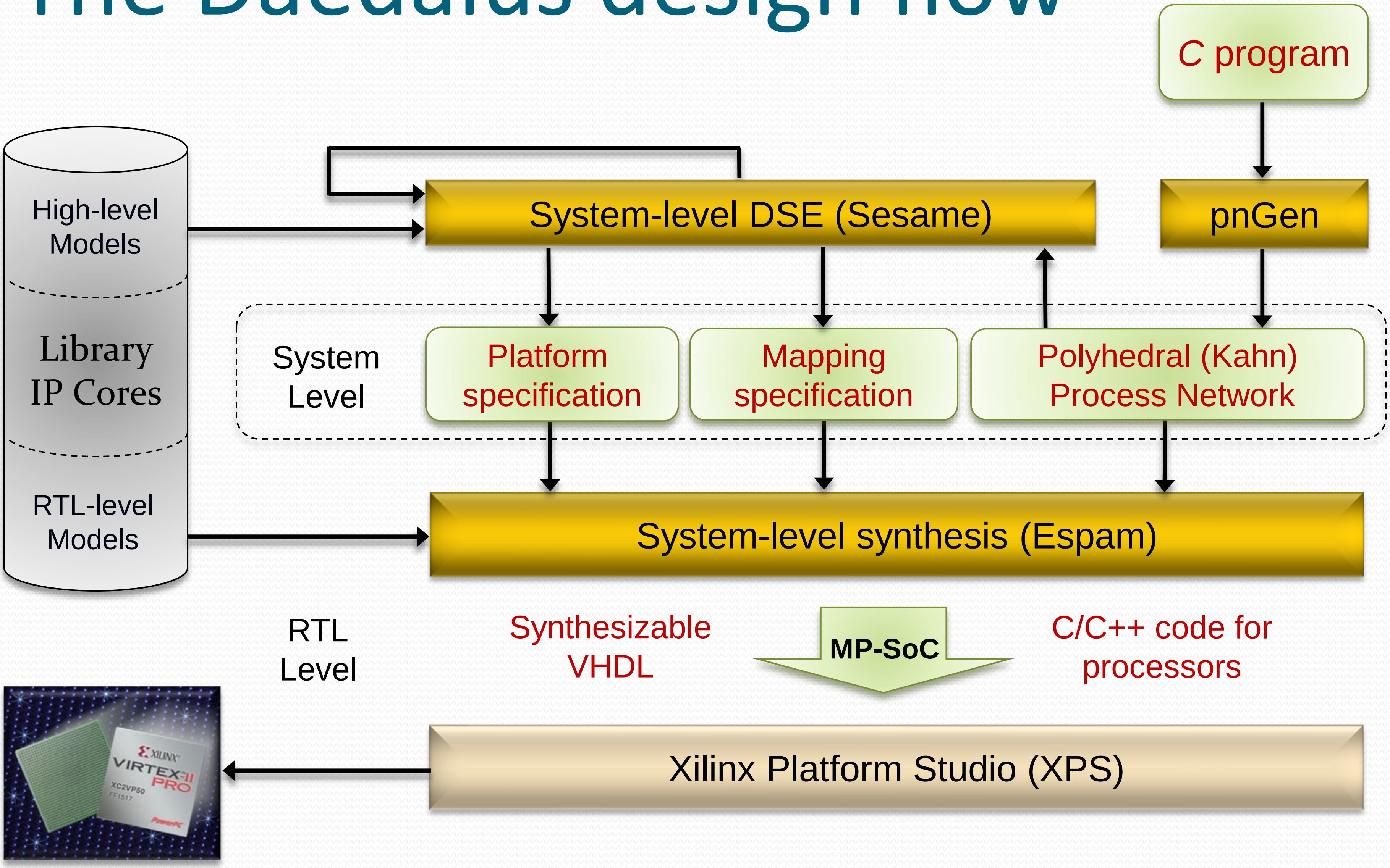
System-level MPSoC design with Daedalus

Lab assignments (ESS course)

Svetlana Minakova
s.minakova@liacs.leidenuniv.nl

LIACS
Leiden University
The Netherlands

The Daedalus design flow



C program

pnGen

High-level Models

Library IP Cores

RTL-level Models

System-level DSE (Sesame)

System Level

Platform specification

Mapping specification

Polyhedral (Kahn) Process Network

System-level synthesis (Espam)

RTL Level

Synthesizable VHDL

MP-SoC

C/C++ code for processors

Xilinx Platform Studio (XPS)

C program – Sobel Edge Detection

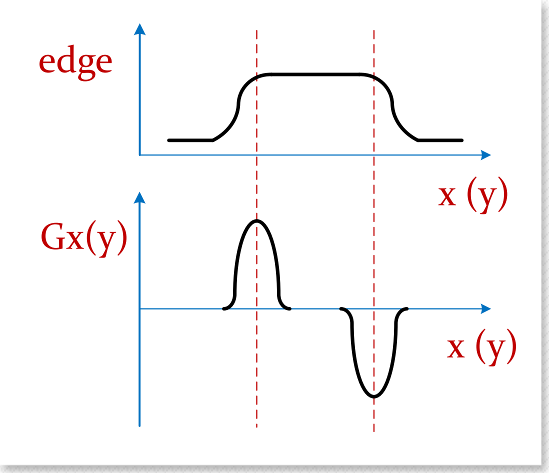
- Locates sharp changes (edges) in the intensity function (image).
- Edges are pixels where brightness changes significantly.
- Sobel edge detection calculates the gradient at each pixel of the image.
- The gradient is calculated as differences in a local neighborhood (3x3) of each pixel using convolution operation.

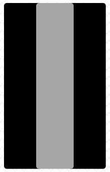
P1	P2	P3	P4	P5	P6
P7	P8	P9	P10	P11	P12
P13	P14	P15	P16	P17	P18
P19	P20	P21	P22	P23	P24
P24	P25	P26	P27	P29	P30

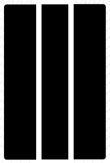
*

C1	C2	C3
C4	C5	C6
C7	C8	C9

Convolution mask







Convolution(P15) = P8C1 + P9C2 + P10C3 + P14C4 + P15C5 + P16C6 +P20C7 +P21C8 + P22C9

Applying Sobel edge detection…

- Gx: detects vertical edges

-1	0	1
-2	0	2
-1	0	1

- Gy: detects horizontal edges

1	2	1
0	0	0
-1	-2	-1

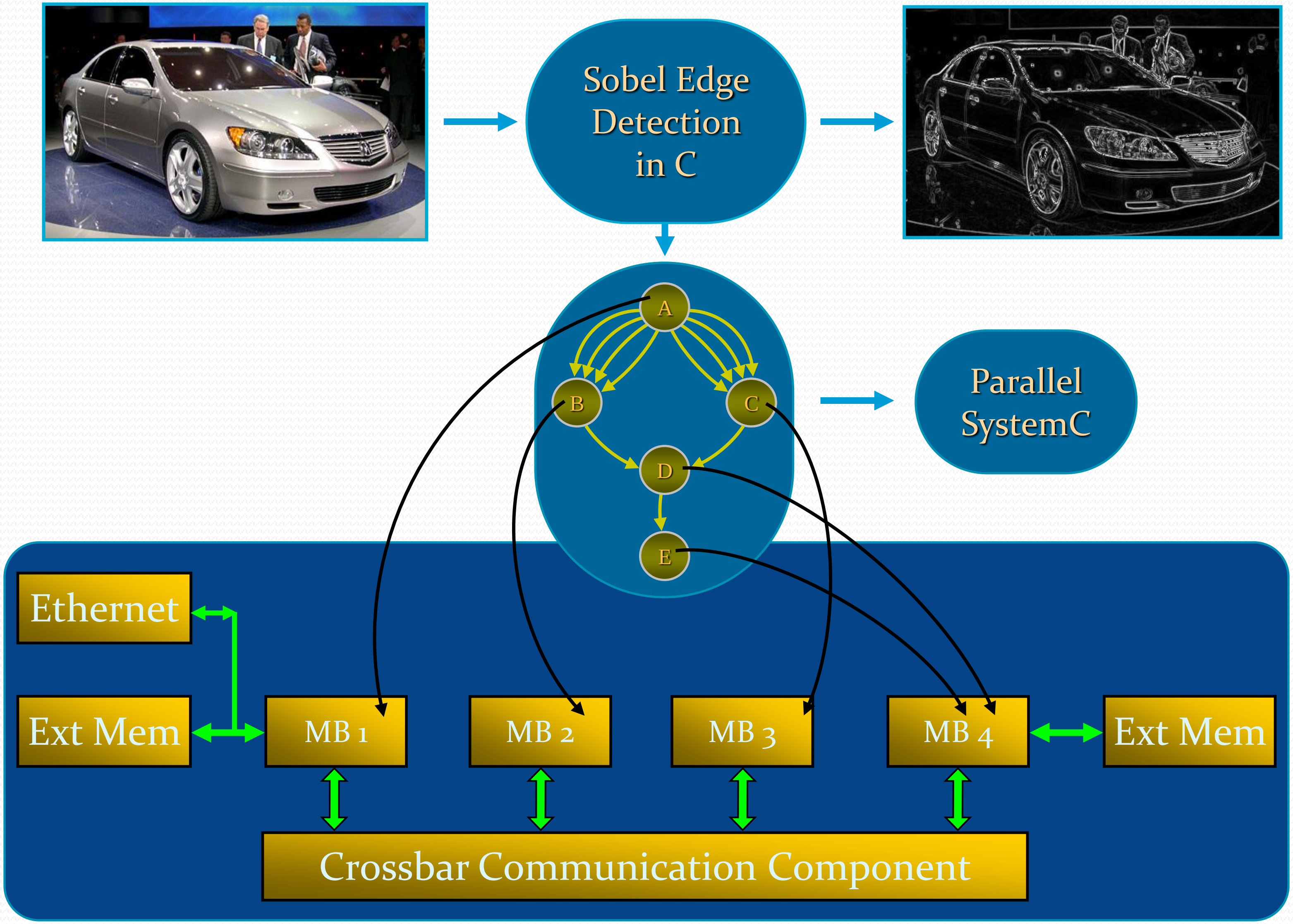
- To approximate the gradient’s magnitude:

  $|G|=|Gx|+|Gy|$

- To visualize the result, normalize the gradient:

  $G=G/4$

… with Daedalus



Sobel Edge Detection in C

Parallel SystemC

Ethernet

Ext Mem

MB 1

MB 2

MB 3

MB 4

Ext Mem

Crossbar Communication Component

Writing C programs compliant with the pnGen

- Restrictions to input top-level program code – *main*()
  - Static Affine Nested Loop Programs (SANLPs)
- No restriction for code in function calls

```
int N=10;
#pragma parameter N 5 100;

int main(void) {

   int i, j, k;
   MyType A[600];

   for (k=1; k<=6*N-3;k++)
      A[k] = Func1();

   for (j=1; i<=N; j++) {
      for (i=j; j<=3*j-2; i++) {
         if (i + j <= 4*N - 6)
            A[i] = Func2(A[2*i-1], A[2*i+1]);
         else
            A[i] = Func3(A[2*i-1], A[2*i+1]);
      }
   }

   return(0);
}
```

Parameters in SANLPs

- **Symbolic constants**, i.e., do not change at run-time
- Have default value
- Have range of values – the generated PPN is valid for every value of the parameters within the specified range

```
int N=10;
#pragma parameter N 5 100;

int main(void) {

   int i, j, k;
   MyType A[600];

   for (k=1; k<=6*N-3;k++)
      A[k] = Func1();

   for (j=1; i<=N; j++) {
      for (i=j; j<=3*j-2; i++) {
         if (i + j <= 4*N - 6)
            A[i] = Func2(A[2*i-1], A[2*i+1]);
         else
            A[i] = Func3(A[2*i-1], A[2*i+1]);
      }
   }

   return(0);
}
```

FOR-loops in SANLPs

- Bounds are affine functions of other loops’ indices and parameters
- No data dependent loop bounds

```
…
p = F(token);

for (i=1; i<=p; i++) { … }
...
```

```
int N=10;
#pragma parameter N 5 100;

int main(void) {

   int i, j, k;
   MyType A[600];

   for (k=1; k<=6*N-3; k++)
      A[k] = Func1();

   for (j=1; j<=N; j++) {
      for (i=j; i<=3*j-2; i++) {
         if (i + j <= 4*N - 6)
            A[i] = Func2(A[2*i-1], A[2*i+1]);
         else
            A[i] = Func3(A[2*i-1], A[2*i+1]);
      }
   }

   return(0);
}
```

if-statements in SANLPs

- Conditions are affine functions of loops’ indices and parameters
- No data dependent conditions

```
…
p = F(token);

if (p > 5) { … }
...            ✗
```

```
int N=10;
#pragma parameter N 5 100;

int main(void) {

   int i, j, k;
   MyType A[600];

   for (k=1; k<=6*N-3;k++)
      A[k] = Func1();

   for (j=1; i<=N; j++) {
      for (i=j; j<=3*j-2; i++) {
         if (i + j <= 4*N - 6)
            A[i] = Func2(A[2*i-1], A[2*i+1]);
         else
            A[i] = Func3(A[2*i-1], A[2*i+1]);
      }
   }

   return(0);
}
```

Scalars, arrays and pointers in SANLPs

- No pointers to data tokens
- Scalars and arrays with an arbitrary type
- Arrays are indexed with affine functions of loops' indices and parameters

```
…
int *a;
...
for (i=1; i<=N; i++) {
   x = F(a);
}
...
```

✗

```
int N=10;
#pragma parameter N 5 100;

int main(void) {

   int i, j, k;
   MyType A[600];

   for (k=1; k<=6*N-3;k++)
      A[k] = Func1();

   for (j=1; i<=N; j++) {
      for (i=j; j<=3*j-2; i++) {
         if (i + j <= 4*N - 6)
            A[i] = Func2(A[2*i-1], A[2*i+1]);
         else
            A[i] = Func3(A[2*i-1], A[2*i+1]);
      }
   }

   return(0);
}
```

Function arguments in SANLPs

- No argument, scalar or a pointer to a scalar

```
…
int a[5];
...
for (i=1; i<=N; i++) {
   x = F(a);
}
...
```

✗

```
…
int a[5];
...
for (i=1; i<=N; i++) {
   x = F(a[i-1], &a[i], a[i+1]);
}
...
```

✓

```
int N=10;
#pragma parameter N 5 100;

int main(void) {

   int i, j, k;
   MyType A[600];

   for (k=1; k<=6*N-3;k++)
      A[k] = Func1();

   for (j=1; i<=N; j++) {
      for (i=j; j<=3*j-2; i++) {
         if (i + j <= 4*N - 6)
            A[i] = Func2(A[2*i-1], A[2*i+1]);
         else
            A[i] = Func3(A[2*i-1], A[2*i+1]);
      }
   }

   return(0);
}
```

Input and output arguments of a function in SANLPs

- Clear separation between input and output arguments of a function

```
…
void F2(int *a);
…
int a;
...
for (i=1; i<=N; i++) {
   a = F1(&a);
   F2(&a);
   F3(a);
}
...
```

✗

```
...
int F2(int a);
…
int a;
...
for (i=1; i<=N; i++) {
   a = F1(&a);
   a = F2(a);
   F3(a);
}
...
```

✓

```
…
void F2(int a, int *a);
…
int a;
...
for (i=1; i<=N; i++) {
   a = F1(&a);
   F2(a, &a);
   F3(a);
}
...
```

✓

```
…
int F2(const int *a);
…
int a;
...
for (i=1; i<=N; i++) {
   a = F1(&a);
   a = F2(&a);
   F3(a);
}
...
```

✓

```
…
void F2(const int *a, int *a);
…
int a;
...
for (i=1; i<=N; i++) {
   a = F1(&a);
   F2(&a, &a);
   F3(a);
}
...
```

✓

Ordering of function arguments in SANLPs

- Input arguments followed by output arguments

```
...
int F1(const int *a, myType b);
void F2(const int *a, int *d, const myType *c)
…
int a;
...
for (i=1; i<=N; i++) {
   a = F1(&a, b);
   F2(&a, &d, &c);
}
...
```

✗

```
...
int F1(const int *a, myType b);
void F2(const int *a, const myType *c, int *d)
…
int a;
...
for (i=1; i<=N; i++) {
   a = F1(&a, b);
   F2(&a, &c, &d);
}
...
```

✓

Sharing the data in SANLPs

- Data between function calls is shared only through function arguments

```
…
int F1(int b) {
   int d;
   d = a*3 – sin(a) + b/4;
   return d;
}
...
int a;
…
int main(void) {
   int b;
   int c;
   ...
   for (i=1; i<=N; i++) {
   a = F1(b);
   F2(&a, &c);
}
...
```

✗

```
…
int F1(int a, int b) {
   int d;
   d = a*3 – sin(a) + b/4;
   return d;
}
...
int main(void) {
   int a;
   int b;
   int c;
   ...
   for (i=1; i<=N; i++) {
   a = F1(a, b);
   F2(&a, &c);
}
...
```

✓

```
…
void F1(int b, int d) {
   a = b*3 – sin(b) + d;
}
...
int a;
…
int main(void) {
   int b;
   int c;
   ...
   for (i=1; i<=N; i++) {
   F1(b, c);
   F2(&a, &c);
}
...
```

✗

```
…
void F1(int b, int d, int *a) {
   a = b*3 – sin(b) + d;
}
…
int main(void) {
   int a;
   int b;
   int c;
   ...
   for (i=1; i<=N; i++) {
   F1(b, c, &a);
   F2(&a, &c);
}
...
```

✓

Let’s start

- For description of Tasks and Deliverables go to:
  - **http://liacs.leidenuniv.nl/~stefanovtp/courses/ES/hands_on/Assignment_Tasks.pdf**

- For instructions related to using Daedalus and other tools go to:
  - **http://liacs.leidenuniv.nl/~stefanovtp/courses/ES/hands_on/Assignment_Instr.pdf**